UNIVERSITY OF TWENTE.

Formal Methods and Tools
Faculty of Electrical Engineering, Mathematics and Computer Science

Master Thesis

Conflict Detection and Analysis for Single-Pushout High-Level Replacement

Author:
Ruud Welling

Committee:
prof.dr.ir. Arend Rensink
dr. Peter van Rossum
dr.ir. Jan Kuper

February 2014

Abstract

In many graph transformation systems (local) confluence is a desired property. For instance, if graph transformation is used for model transformation, then confluence and termination ensure that the order in which rules are applied does not matter, in other words, the final result will always be the same. In this research we investigate sufficient conditions for local confluence of high-level replacement (HLR) systems. Using the single-pushout high-level replacement approach we define critical pairs (minimal conflicting situations). Our sufficient condition for local confluence is based on the analysis of critical pairs. We provide our proofs on a categorical level, therefore our theory cannot only be applied to the category of graphs: we formulate requirements for the categories for which our theorems hold, and we show that our theory is also applicable to attributed graphs.

We also present some of the foundations for confluence analysis for HLR systems with negative application conditions (NACs). In order to show local confluence for a HLR system with NACs we will show that stricter conditions must hold for the critical pairs with NACs (compared to the situation without NACs). We show that HLR systems without NACs are locally confluent. Further analysis of HLR systems with NACs is future work.

Our theory has been implemented in the graph transformation tool GROOVE. Using our implementation we can find all critical pairs for a graph transformation system, and analyse if these pairs are locally confluent. Using this implementation we have performed some experiments on existing graph transformation systems.

Contents

1 Introduction **1**
- 1.1 What is Graph Transformation 1
- 1.2 Approaches to Graph Transformation 2
- 1.3 Conflicts and Confluence 4
- 1.4 Graph Transformation Tools 5
- 1.5 Related Work 6
- 1.6 Research Questions 6
- 1.7 A Last Minute Result 8
- 1.8 Structure 8

2 High-Level Replacement and Graph Transformation Systems **9**
- 2.1 Graphs and Morphisms 9
- 2.2 Category Theory Basics 11
- 2.3 Transformation by Pushout 15
- 2.4 Single-Pushout High-Level Replacement Systems 19

3 Confluence, Conflicts, Critical Pairs **23**
- 3.1 Embedding of Transformations 24
- 3.2 Critical Pairs 30
- 3.3 Sufficient Condition for Local Confluence 32

4 Negative Application Conditions **37**
- 4.1 Rules with NACs 38
- 4.2 Converting Right NACs to Left NACs 39
- 4.3 Derived Rules and Embeddings 43
- 4.4 Critical Pairs and Confluence 46

5 Attributed Graphs **53**
- 5.1 Algebraic Signatures and Algebras 53
- 5.2 Attributed Graphs as an SPO Category 55

6 Efficient Confluence Detection **59**
- 6.1 Essential Critical Pairs 59
- 6.2 Subsumption of Critical Pairs 62

7 Implementation and Experiments **65**
- 7.1 Critical Pair Detection Algorithm 65
- 7.2 Complexity of the Algorithm 66
- 7.3 Confluence Analysis 68

7.4 Graph Transformation Systems in Practice 69
7.5 Results 74
7.6 General Conclusion 81

8 Conclusion **83**
8.1 Achievements 83
8.2 Evaluation 84
8.3 A Last Minute Result 86
8.4 Future work 86

A Proofs **89**
A.1 Pushout Construction in $\boldsymbol{Graph^P}$ 89
A.2 Incorrectness of Pushout Construction in $\boldsymbol{Graph^P}$ 93

Chapter 1

Introduction

In today's world, models exist to help us represent and understand many things. For example, maps or blueprints can model the world or a building. In computer science, graphs are used as models almost everywhere; for example, data and control flow diagrams, UML diagrams, Petri nets, and hardware and software architectures are usually visualized with graphs. Many software development approaches use graphs to model software under development [21, 28]. In this area of model-based software development processes, we are witnessing a paradigm shift, models are no longer mere (passive) documentation, but they are used for code generation, analysis, and simulation as well [18]. Because so many models in software engineering can be represented as graphs, we can use graph transformation as a formalism to specify model transformation.

Graph grammars and graph transformations originated in the early 1970's [4, 7, 15, 29, 30, 34]. Since then the list of areas where graph grammars and transformations have been useful has grown impressively: pattern recognition [37], compiler construction [1], software specification and development [16], database design [20], modelling of concurrent systems [17], visual languages [3] and many others [36]. This wide applicability is due to the fact that graphs are a very natural way of explaining complex situations on an intuitive level. Graph transformation allows us to model the dynamics in these descriptions, since it can describe the evolution of graphical structures [10]. Graph transformation can also be useful if no (system) dynamics are involved, for example in case graph transformations transform one model into another. For model transformations, we can use the theory of graph transformation to verify the correctness of these transformations [11, 12].

1.1 What is Graph Transformation

Graph transformation means changing a graph into another graph. In order to give the graphs being transformed more meaning, we use graphs where every edge has a label. This definition will be later be extended to *attributed graphs*. Attributed graphs have special vertices that represent data such as booleans, strings or integers; these data-vertices may only be the target of an edge and the data-vertices be modified by transformations.

The changes in graph transformations are guided by rules (also called production rules). Rules specify changes in a relatively small substructure of a graph; these changes are modifications to that substructure, such as adding or removing certain parts from the substructure.

The most important requirement for a rule to apply is the occurrence of the substructure in the host graph. We call this substructure the *left-hand side* (LHS) of the rule. An occurrence of the LHS in a host graph is called a *match*. A rule also has a *right-hand side* (RHS) which (partially) overlaps with the LHS. All vertices and edges that are both in the LHS and RHS are preserved after applying the rule. The LHS can have vertices and edges which are not in the RHS, these are the vertices and edges that are deleted by the rule. The RHS can also have vertices and edges that are not in the LHS, these vertices and edges are created after applying rule.

It is possible for a rule to have *negative application conditions* (NACs), which can restrict applicability of a rule. A NAC is a graph that partially overlaps with the LHS of the rule. When a match for a rule exists, the rule is a candidate for application, however the rule may not be applied if one of the NACs of the rule is not satisfied. A NAC is satisfied when there does not exist an occurrence of the NAC in the host graph such that the overlapping part of the NAC and the LHS also overlap in the occurrence of the LHS (the match) and the occurrence of the NAC in the host graph

Figure 1.1 shows a rule with a NAC. This rule models a philosopher (`Phil`) which grabs a `Fork` on his left. The rule is applicable to some host graph G if G contains a `Phil`-vertex with has a `hungry`-self-edge and a `left`-edge to a `Fork` vertex, however there may not exist a `Phil`-vertex which has a `hold` edge to the same `Fork`. If there does exist such a vertex, the NAC is not satisfied.

In Figure 1.2 we show the same rule without the NAC, the graph L is the left-hand side of the rule, and R is the right-hand side. The rule can be applied to the graph G, the occurrence of L in G is shown by the dotted lines. We can see that the NAC is satisfied because there exists no `holds`-edge which targets the bottom fork in G.

The application of the rule results in the graph H. We can see that the rule deletes the `hungry`-edge of the right philosopher. The rule also adds two edges, namely an `hasLeft`-self-edge, and a `hold` edge. We can see that the top `Fork` and the left `Phil` were not needed by the rule. These vertices are left untouched in the resulting graph H.

A *graph transformation system* (GTS) consists of a set of rules. When a rule is applicable to a graph G, and application of this rule gives the graph H, then we say that G can be *directly transformed* to H. A transformation can consist of multiple steps: if G can be (directly) transformed to H and H can be (directly) transformed to X, then we say that G can be *transformed* to X.

1.2 Approaches to Graph Transformation

There exist many approaches to transform graphs, generally these approaches can be divided into two categories: the *algorithmic* (or set theoretic) approaches and the *algebraic* approaches. An overview and comparison between some of these algorithmic and algebraic approaches can be found in [36].

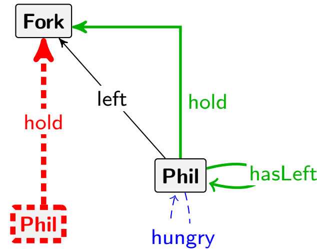


Figure 1.1: A Rule with a NAC. The LHS, RHS and the NAC are shown as one graph: the thin and dashed (blue) edge (hungry) is in the LHS and NAC, but not in the RHS; the wide (green) edges are in the RHS but not in the LHS or NAC; the thick dashed (red) edges and vertices are in the NAC, but not in the LHS or RHS. All other vertices and edges are in the NAC, LHS and RHS. The texts `Phil` and `Fork` on the vertices can be considered self-edges with the text as a label.

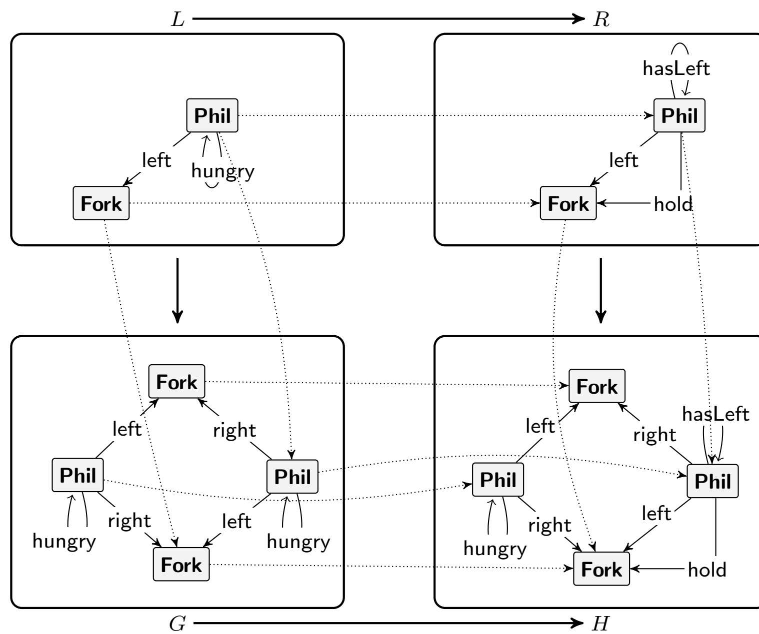


Figure 1.2: Application of a rule. The overlappings of the graphs (morphisms) are specified by the dotted lines.

Two common algebraic approaches to graph transformation are the double-pushout (DPO) approach and the single-pushout (SPO) approach. There are two cases where these approaches yield different results.

- When a rule has a match, but straightforward application of the rule would leave a dangling edge (an edge from which either the source or target vertex has been deleted), the result is no longer a proper graph (since every edge needs a source and a target vertex). Both approaches solve this differently.
- When a rule is applied via a non-injective match (i.e., two vertices v_1 and v_2 of the LHS are matched onto one vertex v of the host graph), and the rule deletes one of these vertices, but keeps the other, a complicated situation arises where the two approaches choose a different solution.

For both cases the rule would not be applicable in the DPO approach, whereas the SPO approach prioritizes the deletion: in the first case the dangling edge is deleted, and in the second case the vertex v of the host graph will be deleted (with all incident edges). A detailed presentation and comparison of the SPO and DPO approaches can be found in [13].

High-Level Replacement

The SPO approach (as well as other algebraic approaches) relies heavily on category theory. This approach cannot only be used to transform graphs, but also other structures. The transformation of these structures is also called *high-level replacement*. A transformation system using high-level replacement is called a *high-level replacement system* (HLR system). A graph transformation system is an example of an HLR system.

We will prove our theorems on a categorical level; in order to do so we define SPO categories, SPO categories fulfil several requirements which allow us to prove our theorems. We will show that there exists an SPO category for transformation of graphs, and we will show that our results can also be applied to attributed graphs. It is very likely that our work can be extended to allow transformation of other structures such as typed graphs, hypergraphs or Petri nets.

1.3 Conflicts and Confluence

Graph transformation is commonly used for model transformation. When transforming models it is important that there exists exactly one final result. Given an HLR system, where the object G (such an object could be a graph) can be transformed to some object X, we say that X is a *normal form* of G if no rules in the HLR system can be applied to X. If every object has a unique normal form, then the order in which rules are applied does not matter.

The property which we are looking for is *confluence*. The notion of confluence is not only used for graph transformation (or high-level replacement), but it is a general property of (abstract) rewrite systems [8]. An HLR system is called confluent when every diverging pair of transfigurations can be joined again, e.g., if G can be transformed to both X and Y and the HLR system is confluent,

then there must exist an object Z and transformations form X to Z and from Y to Z. Confluence ensures that there exists at most one normal form [8].

A weaker version of confluence is called *local confluence*. An HLR system is locally confluent if for all pairs of direct (single-step) transformations from an object G to H_1 and from G to H_2, there is an object X such that both H_1 and H_2 can be transformed to X after applying any number of rules.

We say that an HLR system is terminating when there do not exist infinite transformation sequences. If an HLR system is locally confluent and terminating, then the HLR system is confluent [8]. The termination also ensures the existence of a normal form. Therefore a locally confluent and terminating HLR system has a unique normal form for every object.

It is a known fact that (local) confluence is an undecidable problem [31, 33]. It is also known that termination of an HLR system is undecidable [32]. Some termination criteria for graph transformation systems have been shown in [9].

In this research we will investigate sufficient conditions for when an HLR system is locally confluent. In order to investigate whether an HLR system is locally confluent, we need to study conflicts: given a pair of rules with matches into the same host object, the transformations induced by two rules with matches are in *conflict* when after application of one of the rules, the other can no longer be applied with the same match.

Conflict detection is an important part of our research. We want to find out which pairs of rule applications can be used independently and which cannot. We can distinguish two different kinds of conflicts.

- The most basic kind of conflict is the *delete-use conflict*. This type of conflict can occur in any HLR system. A pair of rules is in delete-use conflict when one rule deletes something that was in the match of the other rule. In other words, a part of the host object that was needed by the second rule was deleted by the first rule, and therefore the second rule can no longer be applied via the same match.

- When we have rules with NACs in our HLR system, a conflict may arise where one rule adds some structure to the host object that is forbidden by the NAC of the other rule. We call this conflict a *produce-forbid conflict*.

The problem of conflict detection has been studied in some depth for the DPO approach [19, 24, 25, 26]. In this thesis we want to extend the existing results for DPO high-level replacement to the SPO setting.

In order to analyse conflicts efficiently, we search for conflict situations where the host object is as small as possible. We call this a *critical pair*.

Critical pairs are useful because (as we prove) there exists a critical pair for every conflict. Therefore critical pairs can be used to reason about all conflicts in an HLR system. We will show that an HLR system is locally confluent when all critical pairs fulfil a slightly stricter definition of local confluence.

1.4 Graph Transformation Tools

There exist many graph transformation tools. This section will discuss AGG and GROOVE. Groove is important to us because have implemented our algorithms

in GROOVE. AGG is an interesting tool because it already supports conflict detection, however it can only detect conflicts for DPO graph transformation.

The two tools are similar in the sense that they can both model graph transformation systems. Both tools allow users to create graph transformation rules and apply these to a host graph.

AGG (the Attributed Graph Grammar system) is a general development environment for algebraic graph transformation systems. AGG supports several kinds of validations. The AGG graph parser is able to check if a graph belongs to a certain graph language determined by a graph grammar. AGG can check consistency conditions that describe basic properties of graphs such as the existence of certain elements. AGG can also perform critical pair analysis, for each pair of rules AGG can find a minimal examples representing all conflicting situations. [38]

GROOVE (GRaph based Object-Oriented VErification) is a graph transformation tool for software model checking of object-oriented systems. GROOVE is able to generate the state space given a start graph and a set of rules. This allows GROOVE perform LTL and CTL model checking: GROOVE can check whether certain properties hold for all states. [35]

1.5 Related Work

A lot of research on conflict detection using DPO graph transformation has already been done: an efficient method to compute the set of all critical pairs for a graph transformation system has been proposed in [25]. This work is continued in [26] where the author defines essential critical pairs as a subset of critical pairs. Critical pair detection for graph transformation rules with NACs has been discussed in [23, 24]. A method for conflict detection for typed attributed graph transformation system is proposed in [19].

Many important concepts for the SPO transformation have been described by Ehrig et al. [13]; for instance, the concept of parallel independence, but also an embedding theorem, which we need later to show that local confluence of all critical pairs implies local confluence of the graph transformation system. SPO graph transformation for attributed graphs is also discussed.

Some work on SPO high-level replacement systems has been done by Ehrig et al. [14]. Ehrig defines some requirements for categories which can be used for SPO high-level replacement. Furthermore, parallel independence is defined, and local confluence of parallel independent transformations is shown.

For SPO graph transformation, Löwe [27] proves that a GTS is locally confluent if every critical pair is strictly confluent (also called transformation confluent). Strict confluence is stronger than local confluence, the author explains why local confluence of all critical pairs is not sufficient to show that a GTS is locally confluent. This paper does not show completeness of critical pairs, and the proof for the local confluence theorem is not completely clear.

1.6 Research Questions

As explained before, the existing methods for conflict detection are only defined and proven for the double-pushout approach. The aim of this research is to

extend these methods so they can be used in the single-pushout approach. Our final goal is to implement an algorithm for critical pair detection and confluence analysis in GROOVE. To reach this goal, we have broken it down into the following sub-questions:

1. How can the existing theory on finding critical pairs using the double-pushout approach be modified for the single-pushout approach?
2. How can we implement an algorithm to find critical pairs?
3. For which cases can we determine if a critical pair is locally confluent?

1.6.1 Modifying existing theory

The first important step is to come up with a definition for critical pairs and prove that the local confluence of all critical pairs implies the local confluence of an HLR system.

Our proofs are on an abstract categorical level: we define SPO categories, and prove our results for these SPO categories. Therefore our results are not only applicable for transformation of graphs. We will show that we can also apply our results to attributed graphs. And it is very likely that our results can also be applied to other structures.

Validation After every step, we will provide formal proofs to show that our results are correct. We will show completeness of critical pairs and we will show that the (strict) confluence of all critical pairs implies local confluence of the graph transformation system.

1.6.2 Algorithm for Critical Pair Detection

Our second research question goes well together with our first question. Based on the definition we have given for critical pairs, it was easy to come up with a method to compute all critical pairs for a graph transformation system. We have implemented this in an algorithm for GROOVE.

Validation In order to test whether our algorithm is correct, we have reused some existing test cases for critical pair detection which have been implemented in AGG. Since it is unclear whether the critical pair test cases for AGG are only for DPO critical pairs, we will have to build some SPO-specific test cases.

1.6.3 Determining local confluence

When we have shown that local confluence of all critical pairs implies the local confluence of a graph transformation system, we will perform an analysis to find out if a critical pair is locally confluent. To perform this local confluence analysis, we will do a state-space search (limited to a certain depth) in GROOVE in order to find out if two diverging transformations can be joined again. When this is the case then this critical pair is locally confluent.

Validation We cannot decide local confluence for all cases. However we can test our implementation on some of the many graph transformation systems that are (supposedly) confluent that have already been implemented in GROOVE. We can then see how well we can decide confluence in practice.

1.7 A Last Minute Result

Just before finishing this thesis, we have discovered that pushouts (transformations) do not always exist in the category of simple graphs. The counterexample (which we present in Appendix A.2) shows that a situation in which our pushout construction (which defines how a graph is transformed) is not correct in all situations.

Because of this some of our propositions about graphs are no longer valid. Our main proofs, however, are proven using abstract categories, for which we have defined the properties required for the results to hold; the fact that the category of simple graphs turns out not to satisfy those properties in no way invalidates the results. For those propositions and proofs which are not valid, we have added footnotes which state that this is the case. A more careful analysis of the situation can be found in Section 8.3.

1.8 Structure

Chapter 2 will explain the basics of our work. We will formally define graphs, provide an introduction to category theory, and we will show how objects (such as graphs) can be transformed to other objects using high-level replacement.

In Chapter 3 we will define the concept of critical pairs and prove that the (strict) local confluence of all critical pairs implies the local confluence of the high-level replacement system. Most of the concepts we introduce in this section already existed (for example for the DPO approach or for SPO graph transformation), however the proofs in this section are all new work.

Chapter 4 will introduce rules with NACs. We will extend the definition for critical pairs to allow rules with NACs, and we will show a sufficient condition for when a high-level replacement system is locally confluent, however we believe a better sufficient condition may be found in the future. The concept of NACs has already been defined by Ehrig et al. [13], however to our knowledge no research has been done on critical pair analysis for SPO high-level replacement (or graph transformation) with NACs.

In Chapter 5 we will define attributed graphs. This definition is based on the existing definition given by Ehrig et al. [10]. We will also show that the theory we have developed in the previous chapters is applicable to attributed graphs.

In Chapter 6 we will show that the local confluence of one critical pair can imply the local confluence of a second critical pair. This can be used as an alternative method to decide local confluence of a critical pair.

We end our thesis with a case study. In Chapter 7 we analyse some graph transformation systems which have been implemented in GROOVE, to find out how well critical pair detection and local confluence analysis works in practice.

Chapter 2

High-Level Replacement and Graph Transformation Systems

This chapter will introduce what graph transformation is and how it works. We start by defining what graphs and graph morphisms are in Section 2.1. Graph transformation relies heavily on category theory. In fact, the single-pushout approach does not only allow transformation of graphs, but also other kinds of objects, such as sets and attributed graphs. Because of this we will explain how to transform objects in any category (for which certain requirements must hold). We will cover the basics of category theory in Section 2.2. In Section 2.3 we will explain how we can apply a rule to an object, to transform it into another object. In Section 2.4 we will define transformation systems for any kind of object (called high-level replacement systems), we will also state the requirements for categories which can be used in SPO high-level replacement systems.

2.1 Graphs and Morphisms

The theory that we develop in this research will be applied in the context of graphs. Even though our proofs are generalized so they can also be applied to different objects, we will be using graphs as examples to make it clear how our theory can be applied to graphs. We start by formally defining what a graph is. We will be using graphs with labelled edges; these labels are elements of *Lab* which is the universe of all possible labels.

Definition 2.1.1 (Graph and Subgraph)**.** A *graph* is a tuple $G = (V_G, E_G)$ consisting of a set V_G of vertices, and a set $E_G \subseteq V_G \times \mathit{Lab} \times V_G$ of (directed) edges. Given an edge $e = (v_1, a, v_2) \in E_G$, the source and target mappings $s_G, t_G : E_G \to V_G$ and label mapping $l_G : E_G \to \mathit{Lab}$ are defined by $s_G(e) = v_1$, $t_G(e) = v_2$, and $l_G(e) = a$. We say that v_1 (resp. v_2, a) is the source vertex (resp. target vertex, label) of e. A *subgraph* S of G, written $S \subseteq G$, is a graph where $V_S \subseteq V_G$ and $E_S \subseteq E_G$.

Remark. Using this definition of graphs, we ensure that every pair of vertices can be connected by at most one (directed) edge with a given label.

Now given two graphs, we want a way to specify how one graph relates to the other. For this we use mappings between the sets of vertices and edges. We will first define mappings for sets.

Definition 2.1.2 (Mappings between Sets)**.** A *partial mapping* f from a set A to a set B, denoted $f : A \to B$, maps elements of A to elements of B.

- We say that f is *injective* if for all $a, a' \in A$, $f(a) = f(a')$ implies $a = a'$,
- We say that f is *surjective* if for all $b \in B$ there is an a such that $f(a) = b$
- We say that f is *total* if f is defined for all $a \in A$
- Every mapping $f : A \to B$ is a total mapping from some subset $\text{dom}(f)$ of A to B. We call $\text{dom}(f)$ the *domain* of f.
- We will sometimes write $f(A) = C$, where $C = \{f(a) \mid a \in \text{dom}(f)\}$.
- A pair of mappings (e_1, e_2) with $e_i : A_i \to B$ for $(i = 1, 2)$ is called *jointly surjective* when $e_1(A_1) \cup e_2(A_2) = B$.
- Two mappings $f : A \to B$ and $g : B \to C$ can be *composed* which results in a mapping $g \circ f : A \to C$, composition is defined as follows:

$$(g \circ f)(x) = \begin{cases} g(f(x)) & \text{if } x \in \text{dom}(f) \text{ and } f(x) \in \text{dom}(g) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Remark. From this definition we can conclude that composition of mappings is associative i.e., $h \circ (g \circ f) = (h \circ g) \circ f$, for any $f : A \to B$, $g : B \to C$ and $h : C \to D$.

A mapping between two graphs, called a graph morphism, allows us to show that a graph contains a substructure of another graph.

Definition 2.1.3 (Graph Morphism)**.** A *partial graph morphism* (also called graph morphism) $f : G \to H$ between two graphs G, H is a pair $f = (f_V : V_G \to V_H, f_E : E_G \to E_H)$ of partial mappings, such that the sources, targets and labels are preserved for every edge, i.e., $f_V \circ s_G = s_H \circ f_E$, $f_V \circ t_G = t_H \circ f_E$ and $l_G = l_H \circ f_E$.

We say that a partial graph morphism f is total (resp. injective, surjective) if both f_V and f_E are total (resp. injective, surjective). Two graph morphisms $f : L \to G$ and $g : H \to G$ are *jointly surjective* if both (f_V, g_V) and (f_E, g_E) are jointly surjective. For a partial graph morphism g we say that $\text{dom}(g) = (\text{dom}(g_V), \text{dom}(g_E))$ is the domain of g. Graph morphisms can be composed: given $f : X \to Y$, $g : Y \to Z$ and $h : X \to Z$ we have $g \circ f = (g_V \circ f_V, g_E \circ f_E)$ (using the associativity of function composition and the fact that g and f preserve sources, targets and labels, we know that $g \circ f$ is a graph morphism as well).

Remark. We will often use the word morphism to denote partial graph morphisms, we will also often use f instead of explicitly using f_V and/or f_E.

The next proposition directly follows from the fact that graph morphisms preserve sources and targets.

Proposition 2.1.4. *Let $g : G \to H$ be a partial graph morphism. Then* $\mathrm{dom}(g) = (\mathrm{dom}(g_V), \mathrm{dom}(g_E))$ *is a subgraph of G.*

Proof. We know that $\mathrm{dom}(g)$ is a graph if $\mathrm{dom}(g_E) \subseteq \mathrm{dom}(g_V) \times Lab \times \mathrm{dom}(g_V)$, this is the case because $g_V \circ s_G = s_H \circ g_E$, $g_V \circ t_G = t_H \circ g_E$. We know by definition that $\mathrm{dom}(g_V) \subseteq G_V$ and $\mathrm{dom}(g_E) \subseteq G_E$, therefore $\mathrm{dom}(g)$ is a subgraph of G. □

Many concepts for algebraic graph transformation rely on category theory. The next section will cover many basics of category theory using sets with mappings and graphs with graph morphisms as examples.

2.2 Category Theory Basics

Before diving into the detail of SPO High-Level Replacement, we introduce some basic definitions which are needed for our proofs. We will define categorical generalizations of injective, surjective and jointly surjective morphisms. These generalisations allow us to prove many of our theorems using category theory.

A large part of the definitions in this section is based on the definitions in [2, 10]. We start by defining categories.

Definition 2.2.1 (Category)**.** A *category* $\boldsymbol{C} = (Ob_{\boldsymbol{C}}, Mor_{\boldsymbol{C}}, \circ, id)$ is defined by

- a class $Ob_{\boldsymbol{C}}$ of *objects*
- for each pair of objects $A, B \in Ob_{\boldsymbol{C}}$, a set $Mor_{\boldsymbol{C}}(A, B)$ of *morphisms*
- for all objects $A, B, C \in Ob_{\boldsymbol{C}}$, a *composition* operation $\circ_{(A,B,C)}$: $Mor_{\boldsymbol{C}}(B, C) \times Mor_{\boldsymbol{C}}(A, B) \to Mor_{\boldsymbol{C}}(A, C)$
- for each object $A \in Ob_{\boldsymbol{C}}$, an *identity* morphism $id_A \in Mor_{\boldsymbol{C}}(A, A)$

such that the following conditions hold:

1. Associativity. For all objects $A, B, C, D \in Ob_{\boldsymbol{C}}$ and morphisms $f : A \to B, g : B \to C, h : C \to D$ it holds that $(h \circ g) \circ f = h \circ (g \circ f)$,
2. Identity. For all objects $A, B \in Ob_{\boldsymbol{C}}$ and morphisms $f : A \to B$, it holds that $f \circ id_A = f$ and $id_B \circ f = f$.

Remark. Instead of $f \in Mor_{\boldsymbol{C}}(A, B)$, we write $f : A \to B$. We also leave out the index for the composition operation, since it is clear which one to use.

In the next two definition we define two categories, one is based on sets and the other is based on graphs. In both categories the morphisms are partial mappings.

Definition 2.2.2 (Category $\boldsymbol{Set^P}$)**.** $\boldsymbol{Set^P}$ is the category with the class of all sets as objects and all (partial) mappings $f : A \to B$ as morphisms. Composition is defined as in Definition 2.1.2. The identity is the identical mapping $id_A = x \mapsto x$.

Definition 2.2.3 (Category $\boldsymbol{Graph^P}$). The category $\boldsymbol{Graph^P}$ consists of the class of all graphs (as defined in Definition 2.1.1) as objects with all possible graph morphisms (see Definition 2.1.3). Composition has been defined in Definition 2.1.3, and identities are the pairwise identities on the sets of vertices and edges.

Next we define monomorphisms and epimorphisms. A monomorphism is a categorical generalization of a total injective morphism, similarly an epimorphism is a categorical generalization of a surjective morphism.

Definition 2.2.4 (Monomorphism and Epimorphism). Given a category $\boldsymbol{C}$, a morphism $m : B \to C$ is called a *monomorphism* if, for all morphisms $f, g : A \to B$ it holds that $m \circ f = m \circ g$ implies $f = g$.

A morphism $e : X \to A$ is called an *epimorphism* if, for all morphisms $f, g : A \to B$, it holds that $f \circ e = g \circ e$ implies $f = g$.

$$A \underset{g}{\overset{f}{\rightrightarrows}} B \xrightarrow{m} C \qquad\qquad X \xrightarrow{e} A \underset{g}{\overset{f}{\rightrightarrows}} B$$

Remark. We will often say that a morphism f is mono (resp. epi) to denote that f is a monomorphism (resp. epimorphism).

In the following lemma and propositions, we will show for $\boldsymbol{Set^P}$, that the monomorphisms are the injective and total mappings, and that the epimorphisms are the surjective mappings. Later, in Proposition 2.2.13 we show that this is also true for $\boldsymbol{Graph^P}$.

Lemma 2.2.5. *Every monomorphism in* $\boldsymbol{Set^P}$ *is total.*

Proof. Let $m : A \to B$ be a mapping which is not total. Then there exists an $x \in A$ such that $m(x)$ is undefined. Let $id_A : A \to A$ be the identity morphism of A, and construct $f : A \to A$ as follows:

$$f(a) = \begin{cases} a & \text{if } a \neq x \\ \text{undefined} & \text{otherwise} \end{cases}$$

It is clear that $f \neq id_A$, we also know that $m \circ id_A = m$, and by choice of f we know that $m \circ f = m$ (since $m(x)$ and $f(x)$ are both undefined). Therefore we have $m \circ f = m \circ id_A$ but $f \neq id_A$, this means that m is not a monomorphism. □

Now we can prove that a $\boldsymbol{Set^P}$-monomorphism is equivalent to an injective and total set mapping.

Proposition 2.2.6. *A* $\boldsymbol{Set^P}$*-morphism* $f : A \to B$ *is a monomorphism if and only if it is total and injective.*

Our proof is based on a similar proof in [2].

Proof. ($\Rightarrow$) Let $f : A \to B$ be a monomorphism, let $a, a' \in A$ such that $a \neq a'$, and let $\{x\}$ be a one-element set. Consider the mappings $g, g' : \{x\} \to A$ where $g(x) = a$ and $g'(x) = a'$. Since $g \neq g'$ it follows, since f is a monomorphism, that $f \circ g \neq f \circ g'$, therefore $f(a) = (f \circ g)(x) \neq (f \circ g')(x) = f(a')$. Therefore f is injective. Lemma 2.2.5 states that f must also be total.

($\Leftarrow$) Conversely suppose that f is total and injective, and $g, h : C \to A$ are mappings such that $g \neq h$ and for some $c \in C$ we have $g(c) \neq h(c)$. Since f is injective, it follows that $f(g(c)) \neq f(h(c))$, therefore $f \circ g \neq f \circ h$, which means f is a monomorphism. □

Similarly we show for $\boldsymbol{Set^P}$ that epimorphisms coincide with surjective set mappings.

Proposition 2.2.7. *A $\boldsymbol{Set^P}$-morphism $f : A \to B$ is an epimorphism if and only if it is surjective.*

We repeat the proof given in [2].

Proof. ($\Rightarrow$) Suppose $f : A \to B$ is not surjective. Then there is a $b \in B$ such that $\forall a \in A : f(a) \neq b$. Define $g_1, g_2 : B \to \{0, 1\}$ as follows: $g_1(x) = 0$ for all $x \in B$, and $g_2(x) = 1$ if $x = b$ and 0 otherwise. We have $g_1 \circ f = g_2 \circ f$ but $g_1 \neq g_2$ therefore f is not an epimorphism.

($\Leftarrow$) Conversely, suppose $f : A \to B$ is surjective, and let $g_1, g_2 : B \to C$ be mappings such that $g_1 \neq g_2$. Then there is a b such that $g_1(b) \neq g_2(b)$. Because f is surjective, b has a preimage in $a \in A$ such that $f(a) = b$. Therefore $g_1(f(a)) \neq g_2(f(a))$ and $g_1 \circ f \neq g_2 \circ f$. It follows that f is an epimorphism. □

Definition 2.2.8 (Isomorphism)**.** A morphism $i : A \to B$ is called an *isomorphism* if there exists a morphism $i^{-1} : B \to A$ such that $i \circ i^{-1} = id_B$ and $i^{-1} \circ i = id_A$

$$A \underset{i^{-1}}{\overset{i}{\rightleftarrows}} B$$

Two objects A and B are isomorphic, written $A \cong B$, if there is an isomorphism $i : A \to B$.

Remark. All the theory that is explained in this paper is supposed to work modulo isomorphism; i.e., we want to consider isomorphic objects equal.

In the category $\boldsymbol{Graph^P}$ it is well known that isomorphisms are injective, total and surjective graph morphisms. We show that injective, total and surjective graph morphisms are indeed isomorphisms in Proposition 2.2.13. First we will show that for any category, an isomorphism is both a mono and an epi.

Proposition 2.2.9. *Every isomorphism is both a monomorphism and an epimorphism.*

Proof. See [2]. □

The converse does not hold for every category [2], however we will show for $\boldsymbol{Set^P}$ that an isomorphism is equivalent to a morphism which is both mono and epi. We already know (Propositions 2.2.6 and 2.2.7) that every $\boldsymbol{Set^P}$-morphism which is both mono and epi must be injective, total and surjective.

Proposition 2.2.10. *A surjective, injective and total $\boldsymbol{Set^P}$-morphism $f : A \to B$ is an isomorphism.*

Proof. Because f is surjective we know that for every $b \in B$ there exists an $a \in A$ such that $f(a) = b$, because f is injective we know that a is unique, and because f is total we know that every element of f has an image in B. We define the morphism $f^{-1} : B \to A$ as follows for all $b \in B$: $f^{-1}(b) = a$ where $a \in A$ such that $f(a) = b$. Now we have $f^{-1}(f(a)) = a$ for any $a \in A$, and $f(f^{-1}(b)) = b$ for any $b \in B$. Therefore $f \circ f^{-1} = id_B$ and $f^{-1} \circ f = id_A$, which means f is an isomorphism. □

The next definition defines when a pair of morphisms with the same target is called jointly epimorphic. This definition is taken from [10].

Definition 2.2.11 (Jointly Epimorphic)**.** A morphism pair (e_1, e_2) with $e_i : A_i \to B$ for $(i = 1, 2)$ is called *jointly epimorphic* if, for all $g, h : B \to C$ with $g \circ e_i = h \circ e_i$ for $i = 1, 2$ we have $g = h$:

$$\begin{array}{l} A_1 \xrightarrow{e_1} \\ A_2 \xrightarrow{e_2} \end{array} B \underset{h}{\overset{g}{\rightrightarrows}} C$$

Next we show that this constitutes a categorical generalization of a pair of jointly surjective morphisms. Remember that we call a pair of $\boldsymbol{Set^P}$ morphisms (e_1, e_2) with $e_i : A_i \to B$ for $(i = 1, 2)$ jointly surjective when $e_1(A_1) \cup e_2(A_2) = B$.

Proposition 2.2.12. *A pair of* $\boldsymbol{Set^P}$*-morphisms* (e_1, e_2) *with* $e_i : A_i \to B$ *for* $(i = 1, 2)$ *is jointly epimorphic if and only if it is jointly surjective.*

Our proof is fairly similar to the proof of Proposition 2.2.7 which originates from [2].

Proof. ($\Rightarrow$) Suppose (e_1, e_2) are not jointly surjective. Then there is a $b \in B$ such that $\forall a \in A_1 : e_1(a) \neq b$, and $\forall a \in A_2 : e_2(a) \neq b$. Define $g, h : B \to \{0, 1\}$ as follows: $g(x) = 0$ for all $x \in B$, and $h(x) = 1$ if $x = b$ and 0 otherwise. We have $g \circ e_i = h \circ e_i$ for $i = 1, 2$ but $g \neq h$ therefore (e_1, e_2) is not jointly epimorphic.

($\Leftarrow$) Conversely, suppose (e_1, e_2) is jointly surjective, and let $g, h : B \to C$ be mappings such that $g \neq h$. Then there is a b such that $g(b) \neq h(b)$. Because e_1 and e_2 are jointly surjective, b must have a preimage in e_1 or e_2. Assume (the other case is analogous) b has a preimage in e_1 i.e., we have an $a \in A_1$ such that $e_1(a) = b$. Then we know $g(e_1(a)) \neq h(e_1(a))$ and $g \circ e_1 \neq h \circ e_1$, therefore e_1 and e_1 are jointly epimorphic. ☐

Proposition 2.2.13. *For any morphism* $f : A \to B$ *in either,* $\boldsymbol{Set^P}$*, or* $\boldsymbol{Graph^P}$ *we have:*

1. *f is a monomorphism if and only if it is total and injective.*
2. *f is an epimorphism if and only if it is surjective.*
3. *f is an isomorphism if and only if it is total, injective and surjective.*
4. *Let $g : A' \to B$ be a morphism in the same category as f, then the pair (f, g) is jointly epimorphic if and only if it is jointly surjective.*

Proof. We have already shown these properties for $\boldsymbol{Set^P}$-morphisms in Propositions 2.2.6, 2.2.7, 2.2.9, 2.2.10 and 2.2.12.

Recall (Definition 2.1.3) that a $\boldsymbol{Graph^P}$-morphism f is total (resp. injective, surjective) if f_V and f_E are total (resp. injective, surjective). A pair of $\boldsymbol{Graph^P}$-morphisms f, f' with the same target is jointly surjective if both pairs (f_V, f'_V) and (f_E, f'_E) are jointly surjective. Now we can conclude that all properties of this proposition hold for $\boldsymbol{Graph^P}$ because they hold for $\boldsymbol{Set^P}$. ☐

2.3 Transformation by Pushout

In high-level replacement systems (HLR systems) objects are transformed to other objects. The changes in these transformations are guided by rules. Every rule has a morphism (the rule morphism) which belongs to a special subclass $\mathcal{R}$ of rule morphisms. In the category $\boldsymbol{Graph^P}$ we can use any partial graph morphism as a rule morphism, i.e., $\mathcal{R}$ contains all $\boldsymbol{Graph^P}$-morphisms. For some categories however, not all morphisms are allowed to be rule morphisms: in Chapter 5 we will see that we cannot use every $\boldsymbol{AGraph^P}$-morphism to transform $\boldsymbol{AGraph^P}$ objects.

The definition presented here is applicable to any category $\boldsymbol{C}$. Later on, when we define SPO HLR systems in Section 2.4, we will specify what conditions must hold the category $\boldsymbol{C}$ and the class of morphisms $\mathcal{R}$.

Definition 2.3.1 (Rule)**.** Given a category $\boldsymbol{C}$ with a class $\mathcal{R}$ of morphisms, a *rule* (also called production) $p = (L \xrightarrow{r} R)$ is an $\mathcal{R}$-morphism, called the rule morphism. The objects L and R are called the *left-hand side* (LHS) and *right-hand side* (RHS) of p, respectively.

Remark. The rules we have defined here are simple rules. For now we consider only these simple rules. Later, in Chapter 4 we will cover rules with negative application conditions (NACs).

In order to be able to apply a rule $p = (L \xrightarrow{r} R)$ to a object G, we need an occurrence of the left-hand side of the rule in G. This occurrence is defined by a morphism $m : L \to G$ such that m belongs to a class of match-morphisms $\mathcal{M}$.

The exact requirements on the class of morphisms $\mathcal{M}$ will be specified later in Definition 2.4.1. For $\boldsymbol{Graph^P}$ the class $\mathcal{M}$ is the class of total $\boldsymbol{Graph^P}$ morphisms. We do not allow all partial morphisms as matches because an $\mathcal{M}$-morphism $m : L \to G$ should model an occurrence of the structure of L in G, if m would be not be total, then it would only model the occurrence of a part of the structure of L in G.

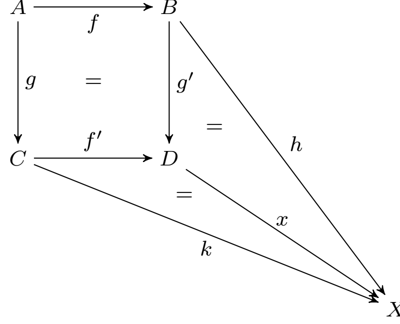
Definition 2.3.2 (Match)**.** Given a category $\boldsymbol{C}$ with a morphism class $\mathcal{M}$, a *match* for a rule $p = (L \xrightarrow{r} R)$ is an object G with a morphism $m : L \to G$ such that $m \in \mathcal{M}$.

In order to apply a rule $p = (L \xrightarrow{r} R)$ to a graph G via the match $m : L \to G$, we need a technique to glue the graphs G and R together over a common substructure. Ideally we would take the common substructure and add all other vertices and edges from G and R. The concept of a pushout formally defines how this glueing construction works. It has been taken from category theory, therefore pushouts can also be used to transform other kinds of objects.

Definition 2.3.3 (Pushout)**.** Given a span of morphisms $C \xleftarrow{g} A \xrightarrow{f} B$ in a category $\boldsymbol{C}$, a pushout $C \xrightarrow{f'} D \xleftarrow{g'} B$ over f and g is defined by

- a pushout object D
- a cospan $C \xrightarrow{f'} D \xleftarrow{g'} B$ with $f' \circ g = g' \circ f$

such that the following universal property is fulfilled: for all objects X and morphisms $h : B \to X$ and $k : C \to X$ with $k \circ g = h \circ f$, there is a unique morphism $x : D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$.



Remark. In the figure above the = sign means that the surrounding morphisms commute, e.g., the = in the square means that $f' \circ g = g' \circ f$.

Pushouts may not exist for all spans, some categories (such as $\boldsymbol{Graph^P}$) have pushouts over all spans (we show this[1] in Construction 2.3.4 and Proposition 2.3.5). For some categories, a pushout over $f : A \to B$ and $g : A \to C$ only exists if f and g belong to a certain subclass of morphisms. For a category $\boldsymbol{C}$ with morphism classes $\mathcal{R}$ for rules, and $\mathcal{M}$ for matches, we know that the category can be used for SPO high-level replacement if the pushout over f and g always exists when $f \in \mathcal{R}$ and $g \in \mathcal{M}$ (or vice versa).

Next we will show a construction[1] for pushouts in $\boldsymbol{Graph^P}$. Throughout this construction we write $f(a) = \perp$ if f is not defined for a.

Construction 2.3.4 (Pushout in $\boldsymbol{Graph^P}$)**.** Let A, B, and C be graphs, let $f : A \to B$, and $g : A \to C$ be (partial) morphisms. We can construct the pushout $B \xrightarrow{g'} D \xleftarrow{f'} C$ as follows:

Define the relation $\sim$ on the disjoint union $U = V_B \dot\cup V_C \dot\cup E_B \dot\cup E_C$ as follows: for all $a \in (V_A \dot\cup E_A)$ we have $f(a) \sim g(a)$ if $f(a) \neq \perp$ and $g(a) \neq \perp$.

Let $[x] = \{y \in U \mid x \equiv y\}$ where $\equiv$ is the equivalence relation generated by $\sim$.

Now we can construct V_D as follows:

$$\begin{aligned} V_D =&\{[x] \mid x \in V_B \dot\cup V_C \\ &\wedge \nexists a \in V_A : ((f(a) = \perp \wedge g(a) \equiv x) \vee (f(a) \equiv x \wedge g(a) = \perp))\} \end{aligned}$$

Before we can construct the set of edges, we first construct three sets: $E_{D,\text{add}}$ (edges that are being added), $E_{D,\text{del}}$ (edges that are being removed) and $E_{D,\text{all}} \subseteq (V_D \times \mathit{Lab} \times V_D)$ (all edges where the source and target exist in V_D).

$$E_{D,\text{add}} = \{([s], l, [t]) | (s, l, t) \in (E_B \dot\cup E_C) \setminus (f_E(E_A) \dot\cup g_E(E_A))\}$$

$$\begin{aligned} E_{D,\text{del}} =&\{([s], l, [t]) | (s, l, t) \in (E_B \dot\cup E_C) \wedge \exists a \in E_A : (f(a) \equiv (s, l, t) \\ &\wedge g(a) = \perp) \vee (g(a) \equiv (s, l, t) \wedge f(a) = \perp)\} \end{aligned}$$

$$E_{D,\text{all}} = \{([s], l, [t]) | (s, l, t) \in (E_B \dot\cup E_C) \wedge [s] \in V_D \wedge [t] \in V_D\}$$

[1] Actually this construction is incorrect, in Appendix A.2 we show that pushouts do not exist over all spans.

The set E_D is constructed as follows:

$$E_D = E_{D,\text{all}} \setminus (E_{D,\text{del}} \setminus E_{D,\text{add}})$$

The morphisms $f'_V : V_C \to V_D$ and $f'_E : E_C \to E_D$) are defined as follows: (g'_V and g'_E are defined analogously)

$$f'_V(c) = \begin{cases} [c] & \text{if } [c] \in V_D \\ \bot & \text{otherwise} \end{cases}$$

$$f'_E((s,l,t)) = \begin{cases} ([s],l,[t]) & \text{if } ([s],l,[t]) \in E_D \\ & \quad \wedge(([s],l,[t]) \in E_{D,\text{del}} \Rightarrow (s,l,t) \notin g_E(E_A)) \\ \bot & \text{otherwise} \end{cases}$$

Proposition 2.3.5. $B \xrightarrow{f'} D \xleftarrow{g'} C$ *as defined in Construction 2.3.4 is a pushout.*[2]

Proof. The proof[3] can be found in Appendix A.1 □

Next we state some important well known properties of pushouts.

Proposition 2.3.6 (Uniqueness, Composition and Decomposition of Pushouts)**.** *Given a category* **C**, *we have the following:*

1. *The pushout object D is unique up to isomorphism.*
2. *The composition and decomposition of pushouts results again in a pushout, i.e., given the following commutative diagram, the statements below are valid:*

$$\begin{array}{ccccc} A & \longrightarrow & B & \longrightarrow & E \\ \downarrow & (1) & \downarrow & (2) & \downarrow \\ C & \longrightarrow & D & \longrightarrow & F \end{array}$$

   - *Pushout composition: if* (1) *and* (2) *are pushouts, then* (1) + (2) *is also a pushout.*
   - *Pushout decomposition: if* (1) *and* (1) + (2) *are pushouts, then* (2) *is also a pushout.*

Proof. See [10]. □

Proposition 2.3.7. *Given a category* ***C*** *and a morphism $f : A \to B$ and an isomorphism $i : A \to C$, the pushout over f and i is $B \xrightarrow{id_B} C \xleftarrow{f \circ i^{-1}} A$.*

Proof. Consider the diagram below:

$$\begin{array}{ccc} A & \xrightarrow{\quad f \quad} & B \\ \downarrow i & = & \downarrow id_B \\ C & \xrightarrow{f \circ i^{-1}} & B \end{array}$$

[2] Actually this construction is incorrect, in Appendix A.2 we show that pushouts do not exist over all spans.

[3] Unfortunately this proof in incorrect, in Appendix A.2 we show a counterexample

We need to show three things: (1) the pushout must commute, (2) for all objects X and morphisms $h : C \to X$ and $k : A \to X$ with $k \circ i = h \circ f$, there is a morphism $x : B \to X$ such that $x \circ id_B = h$ and $x \circ f \circ i^{-1} = k$, and (3) the morphism x is unique.

1. The diagram clearly commutes since $f \circ i^{-1} \circ i = f = id_B \circ f$.
2. Let $x = h$, then we have $x \circ id_B = x = h$ and $x \circ f \circ i^{-1} = h \circ f \circ i^{-1} = k$.
3. We know that x is unique because id_B is epimorphic. □

As mentioned before, the pushout construction can be used to apply a rule via a given match. We call the transformation of a graph G to a graph H using a rule $p : L \xrightarrow{r} R$ and a match $m : L \to G$ a direct transformation.

Definition 2.3.8 (Transformation)**.** Given a rule $p = L \xrightarrow{r} R$ and a match $m : L \to G$ for p, the *direct transformation* from G with rule p and match m, written $G \overset{p,m}{\Longrightarrow} H$, is the pushout over r and m in $\boldsymbol{Graph^P}$. A sequence of direct transformations of the form $\varrho = (G_0 \overset{p_1,m_1}{\Longrightarrow} \dots \overset{p_k,m_k}{\Longrightarrow} G_k)$ constitutes a *transformation* from G_0 to G_k by $p_1 \dots p_k$, abbreviated to $G_0 \overset{*}{\Longrightarrow} G_k$. Given a set of rules P, we say that a transformation $G_0 \overset{*}{\Longrightarrow} G_k$ is *terminating* when there is no rule in P that can be applied to G_k.

Example 2.3.9. Figure 2.1 shows an example of a direct transformation. We will explain how H was constructed by following Construction 2.3.4. In this explanation we name edges and vertices by the numbers written next to them. To avoid ambiguity, we will explicitly state in which graph the vertex or edge is. For example by $(1,3)_{\in G}$ we mean the top left vertex in G.

The first step of our construction is to define the relation $\sim$ as follows:

$$\begin{aligned}(1,3)_{\in G} &\sim 1_{\in R}\\ (1,3)_{\in G} &\sim 3_{\in R}\\ (2,4)_{\in G} &\sim 2_{\in R}\\ (2,4)_{\in G} &\sim 4_{\in R}\\ (5,6)_{\in G} &\sim 6_{\in R}\end{aligned}$$

Using $\sim$ we can form the following equivalence classes:

$$\begin{aligned}[(1,3)_{\in G}] = [1_{\in R}] = [3_{\in R}] &= \{(1,3)_{\in G}, 1_{\in R}, 3_{\in R}\} && = (1,3)_{\in H}\\ [(2,4)_{\in G}] = [2_{\in R}] = [4_{\in R}] &= \{(2,4)_{\in G}, 2_{\in R}, 4_{\in R}\} && = (2,4)_{\in H}\\ [(5,6)_{\in G}] = [6_{\in R}] &= \{(5,6)_{\in G}, 6_{\in R}\} &&\\ [7_{\in R}] &= \{7_{\in R}\} && = 7_{\in H}\\ [a_{\in G}] &= \{a_{\in G}\} &&\\ [b_{\in R}] &= \{b_{\in R}\} &&\\ [c_{\in R}] &= \{c_{\in R}\} &&\\ [d_{\in G}] &= \{d_{\in G}\} &&\end{aligned}$$

The set of vertices V_H does not contain $\{(5,6)_{\in G}, 6_{\in R}\}$, this is because the vertex $5_{\in L}$ has no image under r, which means that $m(5_{\in L}) = (5,6)_{\in G}$ is deleted.

$$V_H = \{[(1,3)_{\in G}],\ [(2,4)_{\in G}],\ [7_{\in R}]\}$$

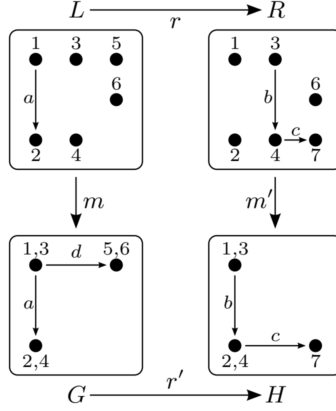


Figure 2.1: Direct transformation by pushout. All edges have the empty label ϵ, the small letters (a-d) and the numbers show how vertices and edges are mapped under the morphisms r, r', m and m'.

We can see that $r'_V((5,6)_{\in G})$ and $m'_V(6_{\in R})$ are both undefined since $[(5,6)_{\in G}] = [6_{\in R}]$ and $[6_{\in R}] \notin H$. All other vertices in G and R have an image under r' or m'.
Now we construct the sets of edges, note that all edges have the empty label ϵ.

$$\begin{aligned} E_{H,add} &= \{([(1,3)_{\in G}], \epsilon, [(5,6)_{\in G}]),\ ([3_{\in R}], \epsilon, [4_{\in R}])\} \\ E_{H,del} &= \{([3_{\in R}], \epsilon, [4_{\in R}])\} \\ E_{H,all} &= \{([3_{\in R}], \epsilon, [4_{\in R}]),\ ([4_{\in R}], \epsilon, [7_{\in R}])\} \\ E_H &= \{([3_{\in R}], \epsilon, [4_{\in R}]),\ ([4_{\in R}], \epsilon, [7_{\in R}])\} \end{aligned}$$

The edge $d_{\in G}$ has no image under r', this edge would be transformed to the edge $([(1,3)_{\in G}], \epsilon, [(5,6)_{\in G}])$, which is in the set $E_{H,add}$, however $[(5,6)_{\in G}] \notin V_H$ (i.e., the edge has no target in V_H), therefore the edge is not an element of $E_{H,all}$ and also not an element of E_H. We also have $r'_E(a_{\in G})$ undefined, because $a_{\in G} = ((1,3)_{\in G}, \epsilon, (2,4)_{\in G})$, and $([(1,3)_{\in G}], \epsilon, [(2,4)_{\in G}]) = ([3_{\in R}], \epsilon, [4_{\in R}]) \in E_{H,del}$, and $d_{\in G} \in m(L)$ therefore, by construction of the edge morphisms $r'_E(a_{\in G})$ is undefined.

2.4 Single-Pushout High-Level Replacement Systems

In this section we present high-level replacement (HLR) systems using Single-Pushout, we consider a category $\boldsymbol{C}$ with a morphism classes $\mathcal{R}$ and $\mathcal{M}$, where all morphisms in $\mathcal{R}$ are allowed as rule morphisms and all morphisms in $\mathcal{M}$ are allowed as matches.

A high-level replacement system consists of a set of rules. These rules allow transforming objects into other objects using rules, matches and pushouts. However, given a rule $p = (L \xrightarrow{r} R) \in P$ and a match $m : L \to G$ for p, we cannot be sure that the pushout over m and r exists. Therefore we give a definition for SPO categories, one of the requirements for these categories is that pushouts exist over $\mathcal{M}$ and $\mathcal{R}$ morphisms.

Our definition for an SPO category differs from the definition given by Ehrig and Löwe [14] in several ways. For example, [14] does not distinguish a class for morphisms which are allowed as rule morphisms, however for the transformation of attributed graphs (see Chapter 5) we do need a separate class of rule morphisms to ensure that pushouts exist. The reason why our definition of an SPO category is different from the definition stated in [14] is because Ehrig and Löwe use SPO categories to prove parallelism properties; we have not include the requirements for those parallelism properties in this definition.

Definition 2.4.1 (SPO Category)**.** Let $\boldsymbol{C}$ be a category with morphism classes $\mathcal{M}$ and $\mathcal{R}$, then we call $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ an *SPO category* if the following properties hold:

1. $\boldsymbol{C}$ has pushouts over any morphism span $B \xleftarrow{f} A \xrightarrow{g} C$, if $f \in \mathcal{M}$ and $g \in \mathcal{R}$ (or vice versa)
2. $\mathcal{M}$ and $\mathcal{R}$ are closed under composition, i.e., $f : A \to B \in \mathcal{M}$ and $g : B \to C \in \mathcal{M}$ implies $g \circ f \in \mathcal{M}$ (and the same for $\mathcal{R}$)
3. $\mathcal{M}$ and $\mathcal{R}$ are closed under decomposition, i.e., $g \circ f \in \mathcal{M}$ and $g \in \mathcal{M}$ implies $f \in \mathcal{M}$

The property 1 is needed to ensure that the pushout over a rule morphism and a match morphism always exists. Properties 2 and 3 are required for various proofs in the next chapters.

Proposition 2.4.2. *Let $\mathcal{M}$ be the class of all total $\boldsymbol{Graph^P}$-morphisms, and let $\mathcal{R}$ be the class of all $\boldsymbol{Graph^P}$-morphisms, then $(\boldsymbol{Graph^P}, \mathcal{M}, \mathcal{R})$ is an SPO category*[4]*.*

Proof. We prove every property of SPO categories separately:

1. $\boldsymbol{Graph^P}$ has pushout over all morphisms[4], see Proposition 2.3.5
2. The composition property follows from Definitions 2.1.2 and 2.1.3.
3. Every $\boldsymbol{Graph^P}$-morphism is an $\mathcal{R}$-morphism, therefore the decomposition property holds for $\mathcal{R}$-morphisms. Let $g \circ f : A \to C \in \mathcal{M}$, assume that $f \notin \mathcal{M}$ let $a \in A$ such that $f(a) = \bot$, then we know that $(g \circ f)(a) = \bot$, a contradiction, therefore we can conclude that $\mathcal{M}$-morphisms are closed under decomposition. □

Now that we know what an SPO category is, we formally define SPO HLR systems.

Definition 2.4.3 (SPO HLR System)**.** An *SPO HLR system* for an SPO category $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ consists of a set of rules P.

- Every rule $p = (L \xrightarrow{r} R) \in P$ consists of an $\mathcal{R}$-morphism r.
- An object G can be directly transformed to an object H by a rule $p = (L \xrightarrow{r} R)$ if there is a $\mathcal{M}$-morphism $m : L \to G$ (called a match for p), and a pushout $(G \xrightarrow{r'} H \xleftarrow{m'} R)$ of (r, m) in $\boldsymbol{C}$. In this case we write $G \stackrel{p,m}{\Longrightarrow} H$.

[4]Unfortunately, this is not true, see Section 8.3 and Appendix A.2.

Example 2.4.4 (Graph Transformation System)**.** We call an HLR system which uses the category $\boldsymbol{Graph^P}$ a *graph transformation system* (GTS). The class $\mathcal{M}$ consists of all total graph morphisms and the class $\mathcal{R}$ consists of all morphisms in $\boldsymbol{Graph^P}$.

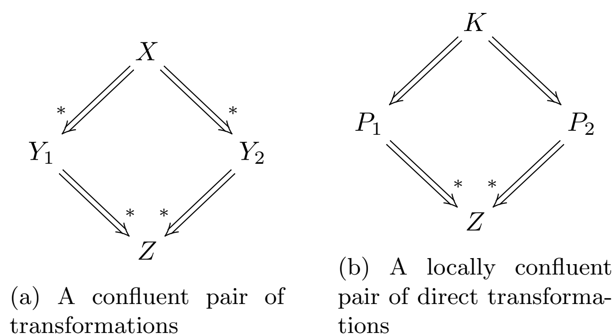
Chapter 3

Confluence, Conflicts, Critical Pairs

In this chapter we will investigate sufficient conditions to decide if an HLR system is locally confluent. Before doing so, we will recall the relevant (general) notions of confluence, local confluence and termination.

We call a pair of transformations $X \stackrel{*}{\Longrightarrow} Y_1$, $X \stackrel{*}{\Longrightarrow} Y_2$ *confluent* when there exist transformations $Y_1 \stackrel{*}{\Longrightarrow} Z$ and $Y_2 \stackrel{*}{\Longrightarrow} Z$. In other words, two diverging transformations are confluent when they can be joined again. An HLR system is called confluent if all derivable transformations are confluent.

A weaker version of confluence is called local confluence, which essentially restricts confluence to direct transformations. A confluent pair of transformations $K \stackrel{*}{\Longrightarrow} P_1$, $K \stackrel{*}{\Longrightarrow} P_2$ is called *locally confluent* when both transformations are direct transformations i.e., $K \Longrightarrow P_1$, $K \Longrightarrow P_2$. The difference between confluence and local confluence is illustrated in Figure 3.1.



(a) A confluent pair of transformations

(b) A locally confluent pair of direct transformations

Figure 3.1: Confluence versus local confluence

We call an HLR system locally confluent when all pairs of direct transformations in the HLR system are locally confluent. Local confluence of an HLR system alone does not imply confluence of the HLR system, the next example demonstrates this.

Example 3.0.5. Consider the following locally confluent HLR system with four

$$\begin{array}{ccccc} G_0 & \Longrightarrow & G_1 & \Longrightarrow & G_2 \\ \downarrow e_0 & & \downarrow e_1 & & \downarrow e_2 \\ X_0 & \Longrightarrow & X_1 & \Longrightarrow & X_2 \end{array}$$

Figure 3.2: An embedding of $\varrho = (G_0 \Longrightarrow G_1 \Longrightarrow G_2)$ into $\delta = (X_0 \Longrightarrow X_1 \Longrightarrow X_2)$

different direct transformations.

$$A \Longleftarrow B \rightleftarrows C \Longrightarrow D$$

There are two pairs of diverging direct transformations $A \Longleftarrow B \Longrightarrow C$ and $B \Longleftarrow C \Longrightarrow D$. We can reason that both pairs are confluent: $A \Longleftarrow B \Longrightarrow C$ is confluent because there exists a transformation $C \overset{*}{\Longrightarrow} A$ and similarly $B \Longleftarrow C \Longrightarrow D$ is confluent because there exists a transformation $B \overset{*}{\Longrightarrow} D$. Since all pairs of direct transformations are confluent, we can conclude that the HLR system is locally confluent. As a whole, however, the HLR system is not confluent: for instance, the diverging pair of transformations $A \overset{*}{\Longleftarrow} C \Longrightarrow D$ is not confluent because both A and D cannot be transformed any more.

We call a transformation $X \overset{*}{\Longrightarrow} Y$ *terminating* if no rules (in the HLR system) can be applied to Y. An HLR system is terminating when there exist no infinite transformation sequences. An HLR system that is both locally confluent and terminating is also confluent [8].

For a locally confluent and terminating HLR system, all pairs of terminating transformation sequences $X \overset{*}{\Longrightarrow} Y_1$ and $X \overset{*}{\Longrightarrow} Y_2$ from the same start object, have targets equal up to isomorphism, i.e., $Y_1 \cong Y_2$.

Confluence and local confluence of an HLR system is undecidable; however, in this chapter we provide sufficient conditions for establishing that an HLR system is locally confluent.

In order to prove our local confluence theorem we make use of so-called *embeddings*. An embedding shows that a transformation ϱ is contained (embedded) in another transformation δ, this means that for every (intermediate) object G_i in ϱ there exists an occurrence ($\mathcal{M}$-morphism) to the (intermediate) object X_i in δ (see Figure 3.2). Embeddings are used to show that the sequence of rules that are applied in the transformation ϱ are applied in the transformation δ, and that the applications have the same effect.

In Section 3.1 we will formally define embeddings, and show when an embedding exists. We will need these embeddings later to prove our sufficient condition for local confluence of an HLR system. Section 3.2 will define what conflicts and critical pairs are, and we will prove that critical pairs are complete (in a sense defined below). In Section 3.3 we will investigate sufficient conditions to decide if an SPO HLR system is locally confluent, based on analysis of the critical pairs.

3.1 Embedding of Transformations

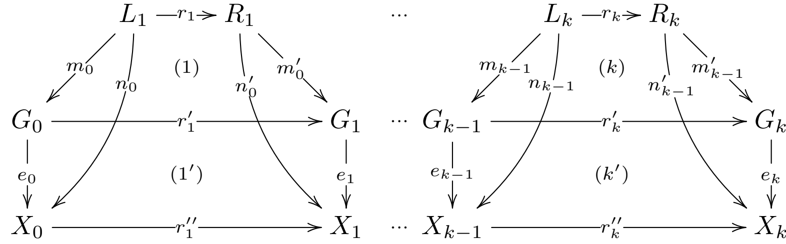
In this section we will formalize under which conditions a transformation in an SPO HLR system using an SPO category $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ can be embedded into a

different (typically larger) context. Given a rule $p = (L \xrightarrow{r} R)$, a transformation $\varrho = (G_0 \overset{p,m}{\Longrightarrow} G_1 \overset{*}{\Longrightarrow} G_n)$ and an $\mathcal{M}$-morphism $e_0 : G_0 \to X_0$, then we know that $e_0 \circ m$ is also an $\mathcal{M}$-morphism, and therefore a match for p. In this section we will investigate when we can conclude that there also exists a transformation $X_0 \overset{p,e_0 \circ m}{\Longrightarrow} X_1 \overset{*}{\Longrightarrow} X_n$ such that there exist $\mathcal{M}$-morphisms $e_i : G_i \to X_i$ for $i = 0 \ldots n$ (see Figure 3.2). We call the family or morphisms e_i an *embedding*. Embeddings allow us to place transformations in a different context, while preserving the structure of the transformed objects in every transformation step.

Embeddings do not always exist: given the transformation $t_1 = (G_0 \overset{p,m}{\Longrightarrow} G_1)$ and $t_2 = (X_0 \overset{p,e_0 \circ m}{\Longrightarrow} X_1)$, we cannot be sure that there also exists an $\mathcal{M}$-morphism $e_1 : G_1 \to X_1$.

In Definition 3.1.1 we formally define embeddings, this definition is more general than the definition for graph embeddings in [13], where an embedding is a family of injective total graph morphisms (in our definition, the morphisms are not necessarily monomorphisms). Later on we will state sufficient conditions for the existence of an embedding.

Definition 3.1.1 (Embedding)**.** Given an SPO category $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ and transformations $\varrho = (G_0 \overset{p_1,m_0}{\Longrightarrow} \ldots \overset{p_k,m_{k-1}}{\Longrightarrow} G_k)$ and $\delta = (X_0 \overset{p_1,n_0}{\Longrightarrow} \ldots \overset{p_k,n_{k-1}}{\Longrightarrow} X_k)$. An *embedding* of ϱ into δ is a family of $\mathcal{M}$-morphisms $e = (G_i \overset{e_i}{\to} X_i)_{i \in \{0,\ldots,k\}}$ such that for all $i \in \{0, \ldots, k-1\}$ we have $e_i \circ m_i = n_i$, see the diagram below. The embedding of ϱ into δ is denoted by $e : \varrho \to \delta$, and the first morphism $e_0 : G_0 \to X_0$ is called the *embedding morphism* of e.



Remark. Since ϱ and δ are transformations, we know for all $i = 1 \ldots k$ that (i) and $(i)+(i')$ are pushouts. Let n'_j be the co-morphism of n_j in the pushout over n_j and r_{j+1}, then we know that $e_{j+1} \circ m'_j = n'_j$ for all $j = 0 \ldots k-1$ because of pushout the pushout composition property and uniqueness of pushouts modulo isomorphism.

Example 3.1.2. In Figure 3.3 we see an embedding of the graph transformation $\varrho = (G_0 \overset{p_1,m_0}{\Longrightarrow} G_1 \overset{p_2,m_1}{\Longrightarrow} G_2)$ into $\delta = (X_0 \overset{p_1,n_0}{\Longrightarrow} X_1 \overset{p_2,n_1}{\Longrightarrow} X_2)$ where $n_i = e_i \circ m_i$ for $i = 1, 2$. We can see that $\langle e_0, e_1, e_2 \rangle : \varrho \to \delta$ is an embedding because $e_0, e_1, e_2 \in \mathcal{M}$ (i.e., e_0, e_1, e_2 are total graph morphisms).

Given a transformation $\varrho = (G \overset{*}{\Longrightarrow} H)$ in an SPO HLR system, and an object X such that there exists an $\mathcal{M}$-morphism $e_0 : G \to X$ (i.e., the structure of G occurs in X), we want to find out when there exists a transformation $\delta = (X \overset{*}{\Longrightarrow} Y)$ with an embedding $e : \varrho \to \delta$. This would mean that the same transformation as ϱ can be applied to X without any additional side-effects.

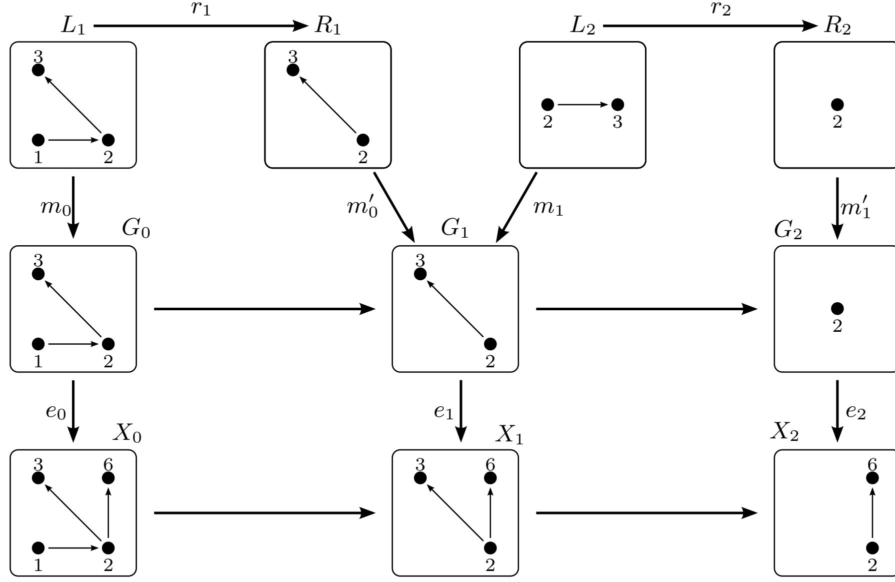


Figure 3.3: An embedding of the transformation $\varrho = (G_0 \overset{*}{\Longrightarrow} G_2)$ into $\delta = (X_0 \overset{*}{\Longrightarrow} X_2)$. The edges in the graphs have the empty label ϵ, the numbers show how vertices are mapped under the morphisms.

In order to reason about what happens in transformation $\varrho = (G \overset{*}{\Longrightarrow} H)$, we need to know what the relation between G and H is, i.e., we need the morphism $\hat{\varrho} : G \to H$. We call this the transformation morphism.

Definition 3.1.3 (Transformation Morphism)**.** Let $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ be an SPO category and let $\varrho = (G_0 \overset{p_1,m_0}{\Longrightarrow} \dots \overset{p_n,m_{n-1}}{\Longrightarrow} G_n)$ be a transformation using the rules $p_i = (L_i \overset{r_i}{\longrightarrow} R_i)$ for $i = 1, \dots, n$, then the *transformation morphism* $\hat{\varrho} : G_0 \to G_n$ is defined as $\hat{\varrho} = r_n' \circ \dots \circ r_1'$ where $r_1', \dots, r_n'$ are the co-rule morphisms in the pushouts $(1), \dots, (n)$, see Figure 3.4.

In case of a zero step transformation $\varrho = (G \overset{*}{\Longrightarrow} G)$, the transformation morphism is defined as $\hat{\varrho} = id_G$.

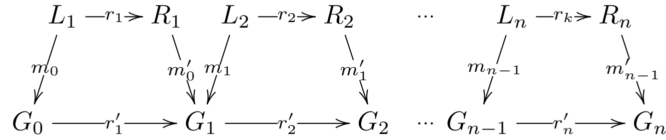


Figure 3.4

Next we define a sufficient condition for when a transformation morphism is an $\mathcal{R}$-morphism.

Definition 3.1.4. Given an SPO category $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$, then we say that pushouts in $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ preserve $\mathcal{R}$-morphisms if for any morphism span $G \overset{m}{\longleftarrow} L \overset{r}{\longrightarrow} R$,

with $m \in \mathcal{M}$ and $r \in \mathcal{R}$, the co-morphism r' of r in the pushout $R \xrightarrow{m'} H \xleftarrow{r'} G$ over m and r is an $\mathcal{R}$-morphism, i.e., $r' \in \mathcal{R}$.

Proposition 3.1.5. *Given an SPO category $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ where pushouts preserve $\mathcal{R}$-morphisms. Let $\varrho = (G \overset{*}{\Longrightarrow} G_n)$, then the transformation morphism $\hat{\varrho}$ is an $\mathcal{R}$-morphism.*

Proof. Consider Figure 3.4, which shows the individual pushouts for the transformations steps of ϱ. By Definition 3.1.4 we know that the morphisms $r'_1 \dots r'_n$ are all $\mathcal{R}$-morphisms. And since (by Definition 2.4.1) the composition of $\mathcal{R}$-morphisms is an $\mathcal{R}$-morphism, we also know that transformation morphisms are $\mathcal{R}$-morphisms. □

Proposition 3.1.6. *Pushouts in the SPO category $(\boldsymbol{Graph^P}, \mathcal{M}, \mathcal{R})$ preserve $\mathcal{R}$-morphisms.*

Proof. Since every $\boldsymbol{Graph^P}$-morphism is an $\mathcal{R}$ morphism, we also know that every morphism in a pushout must be an $\mathcal{R}$-morphism. □

At the end of this section we want to provide a sufficient condition for the existence of an embedding. Given a transformation $\varrho = (G_0 \overset{p_1, m_0}{\Longrightarrow} \dots \overset{p_n, m_{n-1}}{\Longrightarrow} G_n)$ and an $\mathcal{M}$ morphism $e_0 : G \to X_0$, then we could create a (potential) embedding $e = \langle e_0, \dots, e_n \rangle$ and a transformation $\delta = (X_0 \overset{p_1, e_0 \circ m_0}{\Longrightarrow} \dots \overset{p_n, e_n \circ m_{n-1}}{\Longrightarrow} X_n)$ by computing the pushouts $(1) \dots (n)$:

$$\begin{array}{ccccccc}
G_0 & \xrightarrow{r'_1} & G_1 & \cdots & G_{n-1} & \xrightarrow{r'_n} & G_n \\
\downarrow e_0 & (1) & \downarrow e_1 & & \downarrow e_{n-1} & (n) & \downarrow e_n \\
X_0 & \longrightarrow & X_1 & \cdots & X_{n-1} & \longrightarrow & X_n
\end{array}$$

A problem that arises with this approach is that one of the morphisms in e may not be an $\mathcal{M}$-morphism. This would mean that e was not actually an embedding, and this may also mean that not all pushouts $(1) \dots (n)$ exist.

In order to ensure that $e_0 \dots e_n$ are all $\mathcal{M}$-morphisms we will define morphisms which are *strictly $\mathcal{M}$-preserving* with relation to an $\mathcal{M}$-morphism. Before we define what strictly $\mathcal{M}$-preserving means, we will first define the meaning of $\mathcal{M}$-preserving.

In the context of graphs, an $\mathcal{M}$-preserving rule morphism $r : L \to R$ w.r.t. $m : L \to G \;\in \mathcal{M}$ ensures that all elements in $m(L)$ are either meant to be preserved or meant to be deleted, none of these elements will be deleted implicitly.

Definition 3.1.7 ($\mathcal{M}$-Preserving Morphism)**.** Let $\boldsymbol{C}$ be a category with a morphism class $\mathcal{M}$, let $m : A \to B$ be a $\mathcal{M}$-morphism and let $f : A \to C$ be a $\boldsymbol{C}$-morphism with the same source. Then we call the morphism f *$\mathcal{M}$-preserving* w.r.t. m if the pushout $C \xrightarrow{m'} D \xleftarrow{f'} B$ over m and f exists, and the co-morphism m' is an $\mathcal{M}$-morphism.

We show for which morphisms in $\boldsymbol{Graph^P}$ this holds, where $\mathcal{M}$ is the class of total graph morphisms.

Proposition 3.1.8. *Let $r : L \to R$ and $m : L \to G$ be $\boldsymbol{Graph^P}$-morphisms, such that $m \in \mathcal{M}$, where $\mathcal{M}$ is the class of total graph morphisms. Then r is* $\mathcal{M}$-preserving *w.r.t. m if and only $m(x) = m(y)$ implies $x, y \in \text{dom}(r)$ or $x, y \notin \text{dom}(r)$*[1].

Proof. The proof can be found in Appendix A.1 □

Next we define a more strict version of $\mathcal{M}$-preserving morphisms, which states that every prefix (x_1 is a prefix of y of there exists a morphism x_2 such that $x_2 \circ x_1 = y$) of a strictly $\mathcal{M}$-preserving morphism (w.r.t. some morphism x), must be $\mathcal{M}$-preserving (w.r.t. x) as well.

Definition 3.1.9 (Strictly $\mathcal{M}$-Preserving Morphism)**.** Let $\boldsymbol{C}$ be a category with a class $\mathcal{M}$ of morphisms. Let $m : A \to B$ be a $\mathcal{M}$-morphism and let $f : A \to C$ be any morphism. We say that f is *strictly $\mathcal{M}$-preserving* w.r.t. m if for every pair of morphisms f_1, f_2 such that $f_2 \circ f_1 = f$, f_1 is $\mathcal{M}$-preserving w.r.t. m.

From the definition it follows that if f is strictly $\mathcal{M}$-preserving w.r.t. m, then f is $\mathcal{M}$-preserving w.r.t. m (take $f_1 = f$ and $f_2 = id_C$). The next proposition shows a direct consequence of the definition we have just given.

Proposition 3.1.10. *Let $\boldsymbol{C}$ be a category with a class $\mathcal{M}$ of morphisms. Let $m : A \to B$ be a $\mathcal{M}$-morphism and let $f : A \to C$ be any morphism such that f is* strictly $\mathcal{M}$-preserving *w.r.t. m, and let x_1 and x_2 be morphisms such that $x_2 \circ x_1 = f$, then x_1 is* strictly $\mathcal{M}$-preserving *w.r.t. m.*

Proof. We know that x_1 is strictly $\mathcal{M}$-preserving w.r.t. m if for every pair or morphisms y_1, y_2 with $y_2 \circ y_1 = x_1$, y_1 is $\mathcal{M}$-preserving w.r.t. m. We have $x_2 \circ y_2 \circ y_1 = f$. Let $f_2 = x_2 \circ y_2$ and $f_1 = y_1$, then f_1 is $\mathcal{M}$-preserving w.r.t. m because f is strictly $\mathcal{M}$-preserving w.r.t. m. □

We show a sufficient condition for when a $\boldsymbol{Graph^P}$ morphism is strictly $\mathcal{M}$-preserving w.r.t. some morphism r. For $\mathcal{M}$, we take the class of total graph morphisms.

Proposition 3.1.11. *Let $r : L \to R$ and $m : L \to G$ be $\boldsymbol{Graph^P}$-morphisms, such that $m \in \mathcal{M}$, where $\mathcal{M}$ is the class of total graph morphisms. Then r is* strictly $\mathcal{M}$-preserving *w.r.t. m if $m(x) = m(y)$ implies $x = y$ or $x, y \in \text{dom}(r)$*[1].

Proof. Let m be a total morphism and r a morphism such that $m(x) = m(y)$ implies $x = y$ or $x, y \in \text{dom}(r)$. Let r_1 and r_2 be morphisms such that $r_2 \circ r_1 = r$. We must show that r_1 is $\mathcal{M}$-preserving w.r.t. m i.e. (by Proposition 3.1.8) $m(x) = m(y)$ implies $x, y \in \text{dom}(r_1)$ or $x, y \notin \text{dom}(r_1)$. Assume to the contrary that r_1 is not $\mathcal{M}$-preserving w.r.t. m, i.e., there exist $x, y \in L$ such that $m(x) = m(y)$, $x \in \text{dom}(r_1)$ and $y \notin \text{dom}(r_1)$. This means that $x \neq y$ and $y \notin \text{dom}(r)$, which contradicts our conditions on r. □

We can now answer the main question of this section: under which conditions can we embed a transformation into a different context. Ehrig et al. [13] have already given a similar proof (in the context of graphs) for embeddings where every morphism in the embedding must be injective (i.e., a monomorphism),

[1] Unfortunately, we can not be sure that the pushout over r and m exist, therefore we can not be sure if r is (strictly) $\mathcal{M}$-preserving w.r.t. m, see Section 8.3 and Appendix A.2.

the following theorem is more general in the sense that we do not restrict to monomorphisms, but allow our embeddings contain any $\mathcal{M}$-morphism where the transformation morphism is strictly $\mathcal{M}$-preserving w.r.t. the first morphism in the embedding.

Theorem 3.1.12 (Embedding Theorem)**.** *Given an SPO category $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ where pushouts preserve $\mathcal{R}$-morphisms, let $\varrho = (G_0 \overset{p_1,m_0}{\Longrightarrow} \dots \overset{p_k,m_{k-1}}{\Longrightarrow} G_k)$ be a transformation, and $e_0 : G_0 \to X_0$ an $\mathcal{M}$-morphism such that $\hat{\varrho}$ is strictly $\mathcal{M}$-preserving w.r.t. e_0, then there is a transformation $\delta = (X_0 \overset{p_1,n_0}{\Longrightarrow} \dots \overset{p_k,n_{k-1}}{\Longrightarrow} X_k)$ with an embedding $e : \varrho \to \delta$.*

Furthermore, let $e_k : G_k \to X_k$ be the last morphism of the embedding e. Then $X_0 \overset{\hat{\delta}}{\longrightarrow} X_k \overset{e_k}{\longleftarrow} G_k$ is the pushout over $X_0 \overset{e_0}{\longleftarrow} G_0 \overset{\hat{\varrho}}{\longrightarrow} G_k$.

Proof. By induction over the length k of ϱ: Let $k = 0$ then the embedding $e : \varrho \to \delta$ consists of only one morphism e_0. Furthermore we have the transformation morphisms $\hat{\varrho} : G_0 \to G_0 = id_{G_0}$ and $\hat{\delta} : X_0 \to X_0 = id_{X_0}$, and Proposition 2.3.7 implies that $X_0 \overset{id_{X_0}}{\longrightarrow} X_0 \overset{e_0}{\longleftarrow} G_0$ is the pushout over $X_0 \overset{e_0}{\longleftarrow} G_0 \overset{id_{G_0}}{\longrightarrow} G_0$.

Assume as induction hypothesis that this theorem holds for transformations ϱ of length $k = n$. Let ϱ' be a transformation of length $k = n+1$, let ϱ be the prefix of ϱ' consisting of the first n direct transformations, and let t_ϱ be the last transformation of ϱ':

$$\begin{aligned} \varrho' &= (G_0 \overset{p_1,m_0}{\Longrightarrow} \dots \overset{p_n,m_{n-1}}{\Longrightarrow} G_n \overset{p_{n+1},m_n}{\Longrightarrow} G_{n+1}) \\ \varrho &= (G_0 \overset{p_1,m_0}{\Longrightarrow} \dots \overset{p_n,m_{n-1}}{\Longrightarrow} G_n) \\ t_\varrho &= (G_n \overset{p_{n+1},m_n}{\Longrightarrow} G_{n+1}) \end{aligned}$$

$$\begin{array}{ccccccccccc}
L_1 \xrightarrow{r_1} R_1 & & L_2 \xrightarrow{r_2} R_2 & & \cdots & & L_n \xrightarrow{r_k} R_n & & L_{n+1} \xrightarrow{r_{n+1}} R_{n+1} \\
{\scriptstyle m_0} \swarrow \quad \searrow {\scriptstyle m_0'} & & {\scriptstyle m_1} \swarrow \quad \searrow {\scriptstyle m_1'} & & & & {\scriptstyle m_{n-1}} \swarrow \quad \searrow {\scriptstyle m_{n-1}'} & & {\scriptstyle m_n} \swarrow \;(t)\; \searrow {\scriptstyle m_n'} \\
G_0 \xrightarrow{r_1'} G_1 & & \xrightarrow{r_2'} G_2 & & \cdots & & G_{n-1} \xrightarrow{r_n'} G_n & & \xrightarrow{r_{n+1}'} G_{n+1} \\
\downarrow e_0 \quad (1) \quad \downarrow e_1 & & (2) \quad \downarrow e_2 & & & & \downarrow e_{n-1} \quad (n) \quad \downarrow e_n & & (n{+}1) \quad \downarrow e_{n+1} \\
X_0 \xrightarrow{r_1''} X_1 & & \xrightarrow{r_2''} X_2 & & \cdots & & X_{n-1} \xrightarrow{r_n''} X_n & & \xrightarrow{r_{n+1}''} X_{n+1}
\end{array}$$

Furthermore let e_0 be an $\mathcal{M}$-morphism such that $\hat{\varrho}'$ is strictly $\mathcal{M}$-preserving w.r.t. e_0, this implies that $\hat{\varrho}$ is also strictly $\mathcal{M}$-preserving w.r.t. e_0 (by Proposition 3.1.10). Now we apply our induction hypothesis, this gives us a transformation $\delta = (X_0 \overset{p_1,n_0}{\Longrightarrow} \dots \overset{p_n,n_{n-1}}{\Longrightarrow} X_n)$, an embedding $e : \varrho \to \delta$ and we know that the pushout $X_0 \overset{\hat{\delta}}{\longrightarrow} X_n \overset{e_n}{\longleftarrow} G_n$ of $X_0 \overset{e_0}{\longleftarrow} G_0 \overset{\hat{\varrho}}{\longrightarrow} G_n$ exists (depicted as $(1) + (2) + \dots + (n)$), where e_n is the last morphism of e.

There is a transformation $\delta' = (X_0 \overset{p_1,n_0}{\Longrightarrow} \dots \overset{p_n,n_{n-1}}{\Longrightarrow} X_n \overset{p_{n+1},n_n}{\Longrightarrow} X_{n+1})$ and an embedding $e' : \varrho' \to \delta'$ if there are embeddings $e : \varrho \to \delta$ and $\langle e_n, e_{n+1} \rangle : t_\varrho \to t_\delta$ where e_n is also the last morphism of e, δ is the prefix of δ' containing the first

n direct transformations, and t_δ is the last transformation of δ':

$$\delta' = (X_0 \overset{p_1,n_0}{\Longrightarrow} \dots \overset{p_n,n_{n-1}}{\Longrightarrow} X_n \overset{p_{n+1},n_n}{\Longrightarrow} X_{n+1})$$
$$\delta = (X_0 \overset{p_1,n_0}{\Longrightarrow} \dots \overset{p_n,n_{n-1}}{\Longrightarrow} X_n)$$
$$t_\delta = (X_n \overset{p_{n+1},n_n}{\Longrightarrow} X_{n+1})$$

The transformation t_ϱ is formed by the pushout (t), similarly the transformation t_δ is formed by the pushout $(t) + (n+1)$. Using the pushout decomposition property, we know that $(n+1)$ is a pushout. Now we can use the pushout composition property to conclude that $(1) + (2) + \cdots + (n) + (n+1)$, i.e., $X_0 \xrightarrow{\hat{\delta}'} X_{n+1} \xleftarrow{e_{n+1}} G_{n+1}$, is the pushout over $X_0 \xleftarrow{e_0} G_0 \xrightarrow{\hat{\varrho}'} G_{n+1}$. Because ϱ' is strictly $\mathcal{M}$-preserving w.r.t. e_0 we know that e_{n+1} is an $\mathcal{M}$-morphism, e_n is an $\mathcal{M}$-morphism because it is part of the embedding e. Therefore $\langle e_n, e_{n+1}\rangle : t_{n+1} \to t'_{n+1}$ is an embedding. And we have the embedding $e' = \langle e_0, \dots, e_n, e_{n+1}\rangle : \varrho' \to \delta'$. □

3.2 Critical Pairs

In order to reason about local confluence of all pairs of direct transformations, we will distinguish two types of direct transformations: those that are parallel dependent, and those that parallel independent. A pair of transformations (with rules p_1 and p_2) is parallel independent if p_1 can still be applied when p_2 has been applied first (or vice versa). The reason for this distinction is that parallel independent pairs of direct transformations are always locally confluent (Theorem 3.3.3), whereas parallel dependent pairs of direct transformations may not be locally confluent at all. In this section we will formally define parallel independence and we will introduce critical pairs: a subset of all the parallel dependent direct transformations.

A critical pair is a parallel dependent pair of direct transformations where the matches m_1 and m_2 are jointly epimorphic. Critical pairs provide an easy way to reason about conflicts, we will show that (if some conditions hold) there exists a critical pair for every conflict. Furthermore, given an SPO HLR system for $(\boldsymbol{Graph^P}, \mathcal{M}, \mathcal{R})$ where the left-hand sides of all rules consist of a finite graphs, the then the set of critical pairs is finite (the left-hand sides of two rules can be overlapped in only finitely many ways), whereas the set of parallel dependent pairs of direct transformations is not finite: for every parallel dependent pair of direct transformations with matches $m_1 : L_1 \to G$ and $m_2 : L_2 \to G$ there exists a graph H such that $G \subset H$, this means that there also must exist matches $m'_1 : L_1 \to H$ and $m'_2 : L_2 \to H$ which form a different parallel dependent pair of direct transformations.

In Section 3.3, we will use critical pairs for proving local confluence of an HLR system. This section will formally define parallel independence, which will be followed by the formal definition of conflicts and critical pairs. We start with the definition of parallel independence.

Definition 3.2.1 (Parallel Independence)**.** Given an SPO HLR system where $p_1 = (L_1 \xrightarrow{r_1} R_1)$ and $p_2 = (L_2 \xrightarrow{r_2} R_2)$ are two rules, let $t_1 = (G \overset{p_1,m_1}{\Longrightarrow} H_1)$ and $t_2 = (G \overset{p_2,m_2}{\Longrightarrow} H_2)$ be two different direct transformations. Then

t_1 and t_2 are *parallel independent* if $m_2^* = r_1' \circ m_2$ is a match for p_2, and $m_1^* = r_2' \circ m_1$ is a match for p_1, i.e., $m_1^*, m_2^* \in \mathcal{M}$ (see Figure 3.5). We call a pair of transformations parallel dependent, if they are not parallel independent.

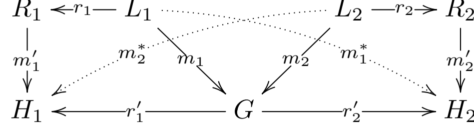


Figure 3.5: The direct transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ are parallel independent when m_1^* is a match for p_1 and m_2^* is a match for p_2

A parallel dependent pair of transformations is also called a *conflict*. Specifically, a pair of transformation that is parallel dependent according to Definition 3.2.1 is called a *delete-use-conflict*. In particular, p_2 deletes something that p_1 uses if $m_1^* = r_2' \circ m_1$ is not total (see Figure 3.5) and/or p_1 deletes something that p_2 uses if $m_2^* = r_1' \circ m_2$ is not total. In Chapter 4, we will see that a different kind of conflict will arise when we redefine parallel independence for rules with negative application conditions.

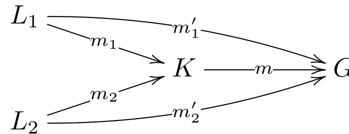
In order to reason about parallel dependent transformations, we define critical pairs. We will show that for SPO HLR systems where $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ is an SPO category, there exists a critical pair for every pair of parallel dependent derivations.

Definition 3.2.2 (Critical Pair)**.** Given an SPO HLR System, a critical pair is a pair of parallel dependent direct transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ such that m_1 and m_2 are jointly epimorphic.

A critical pair is a minimal conflict situation, in the sense that the object G is as small as possible (G contains no elements that are not required for the matches m_1 and m_2). We mentioned before that every conflict can be represented by a critical pair, in the sense that every pair of direct transformations in conflict is an embedding of a critical pair in a different context. This is not true for all SPO categories, therefore we will define some extra assumptions on the SPO categories before we can prove that critical pairs are complete.

Definition 3.2.3 (Strict SPO Category)**.** An SPO category $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ is a *strict SPO category* if the following properties hold:

1. Pushouts in $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ preserve $\mathcal{R}$ morphisms.
2. All $\mathcal{R}$-morphisms are strictly $\mathcal{M}$-preserving w.r.t. any monomorphism in $\mathcal{M}$
3. For any pair of $\mathcal{M}$-morphisms $m_1' : L_1 \to G$ and $m_2' : L_2 \to G$, there exists an object K and $\mathcal{M}$-morphisms $m_1 : L_1 \to K, m_2 : L_2 \to K, m : K \to G$, such that m_1 and m_2 are jointly epimorphic, m is a monomorphism, $m \circ m_1 = m_1'$ and $m \circ m_2 = m_2'$:



Proposition 3.2.4. *The SPO category* $(\textbf{\textit{Graph}}^{\textbf{\textit{P}}}, \mathcal{M}, \mathcal{R})$ *is a strict SPO category*[2].

Proof. We prove the three requirements separately:

1. By Proposition 3.1.10, we know that pushouts preserve $\mathcal{R}$-morphisms.

2. Since a monomorphism is an injective total graph morphism (Proposition 2.2.13), this follows from Proposition 3.1.11.

3. We can construct $K \subseteq G$ containing only those edges and vertices of G that are in the image of m'_1 or m'_2. Let m be the inclusion of K in G (m is total and injective and therefore a monomorphism). For $i = 1, 2$, $m_i : L_i \to K$ is the restriction of m'_i to the codomain K. By construction we know that m_1 and m_2 are total i.e., $m_1, m_2 \in \mathcal{M}$. By construction of m, m_1 and m_2 we know $m \circ m_1 = m'_1$, $m \circ m_2 = m'_2$ and the pair (m_1, m_2) is jointly epimorphic. □

Given an SPO HLR system with a strict SPO category, we can prove that there exists a critical pair for every conflict. To our knowledge, completeness of critical pairs has not been proven in any existing work on critical pairs in SPO high-level replacement (or graph transformation).

Theorem 3.2.5 (Completeness of Critical Pairs)**.** *Given an SPO HLR system where* $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ *is a strict SPO category, for each pair of parallel dependent direct transformations* $t'_1 = (G \overset{p_1, m'_1}{\Longrightarrow} H'_1)$ *and* $t'_2 = (G \overset{p_2, m'_2}{\Longrightarrow} H'_2)$*, there exists a critical pair* $t_1 = (K \overset{p_1, m_1}{\Longrightarrow} H_1)$ *and* $t_2 = (K \overset{p_2, m_2}{\Longrightarrow} H_2)$ *and a monomorphism* $m : K \to G$ *in* $\mathcal{M}$ *such that embeddings* $\langle m, m^*_i \rangle : t_i \to t'_i$ *for* $i = 1, 2$ *exist.*

Proof. By Definition 3.2.3, there exists an object K, with $\mathcal{M}$-morphisms $m_1 : L_1 \to K, m_2 : L_2 \to K, m : K \to G$, such that m_1 and m_2 are jointly epimorphic, m is a monomorphism, $m \circ m_1 = m'_1$ and $m \circ m_2 = m'_2$. We form the critical pair of direct transformations $t_1 = (K \overset{p_1, m_1}{\Longrightarrow} H_1)$ and $t_2 = (K \overset{p_2, m_2}{\Longrightarrow} H_2)$. Let $\hat{t_1} : K \xrightarrow{r'_1} H_1$ and $\hat{t_2} : K \xrightarrow{r'_2} H_2$ be the transformation morphisms for t_1 and t_2. Since m is a monomorphism in $\mathcal{M}$, Definition 3.2.3 implies that both $\hat{t_1}$ and $\hat{t_2}$ are $\mathcal{M}$-preserving w.r.t. m. Therefore Theorem 3.1.12 implies the existence of the embeddings $\langle m, m^*_i \rangle : t_i \to t'_i$ for $i = 1, 2$. □

3.3 Sufficient Condition for Local Confluence

Now that we have defined critical pairs, we can use these to reason about all parallel dependent transformations. In this section we will show a sufficient condition for when an HLR system is locally confluent. We will show by an example (based on [27]) that the local confluence of all critical pairs is not sufficient for an HLR system to be locally confluent. Therefore, we define a stronger notion of local confluence with does suffice.

In Section 3.1 we have shown that transformations can be embedded in a different context. Now intuitively, it would seem that given a locally confluent

[2]This is actually not true because $(\textbf{\textit{Graph}}^{\textbf{\textit{P}}}, \mathcal{M}, \mathcal{R})$ is not an SPO category, see Section 8.3 and Appendix A.2.

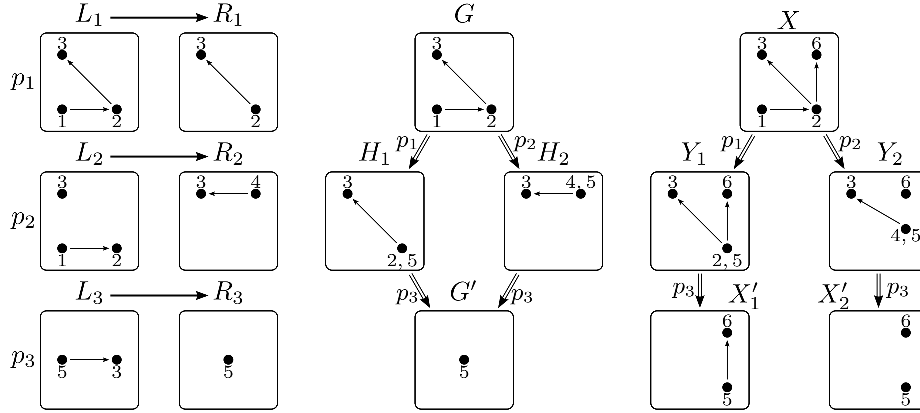


Figure 3.6: A situation where a critical pair is locally confluent, but an embedding of this critical pair in a larger context is not. The rules are shown on the left, the critical pair in the middle, and the larger context on the right. We have numbered the vertices to show how they are mapped under their matches, the morphisms for the edges are defined by their source and target, and edges have the empty label ϵ.

critical pair, all embeddings of this critical pair are locally confluent too. This is not the case, as was already noted by Löwe [27]; the next example demonstrates this.

Example 3.3.1. In Figure 3.6, we show three rules on the left: $p_1 = (L_1 \xrightarrow{r_1} R_1)$, $p_2 = (L_2 \xrightarrow{r_2} R_2)$ and $p_3 = (L_3 \xrightarrow{r_3} R_3)$. In the middle we show a critical pair of p_1 and p_2 that is locally confluent (up to isomorphism). The right part of Figure 3.6 shows an embedding of the critical pair in a larger context. We can see that the embedding is not confluent, even though we apply the same rules with the same matches. We can conclude that not all embeddings of a locally confluent critical pair are locally confluent, in other words, local confluence is not preserved by the embedding.

Clearly the edge $(5, \epsilon, 6) \in X'_1$ (and the absence of a similar edge in X'_2) is the reason why X'_1 and X'_2 are not isomorphic. Let $\hat{\delta_1}$ and $\hat{\delta_2}$ be the transformation morphisms of $\delta_1 = (X \Longrightarrow Y_1 \Longrightarrow X'_1$ and $\delta_2 = (X \Longrightarrow Y_2 \Longrightarrow X'_2$, respectively. We can see that $\hat{\delta_1}$ is defined the vertex $2 \in X$ (i.e., the vertex is not deleted in the transformation δ_1), but $2 \in X$ is not defined under $\hat{\delta_2}$ (i.e., the vertex is deleted in the transformation δ_2). The fact that the transformation morphisms are not equivalent seems to be the reason that the embedding of the critical pair is not locally confluent.

In this example we can see that the rule morphisms of the transformations do not commute, since one transformation deletes the vertex $2 \in X$, while the other transformation preserves this vertex. We will formulate a stronger notion of local confluence, which requires that the transformation morphisms must commute.

Definition 3.3.2 (Strict Local Confluence)**.** Let $p_1 = (L_1 \xrightarrow{r_1} R_1)$ and $p_2 = (L_2 \xrightarrow{r_2} R_2)$ be two rules. A pair of direct transformations $P_1 \overset{p_1,m_1}{\Longleftarrow} K \overset{p_2,m_2}{\Longrightarrow} P_2$ is called *strictly locally confluent* (modulo isomorphism) if we have the following:

1. Confluence: the pair of direct transformations is locally confluent, i.e., there are transformations $P_1 \stackrel{*}{\Longrightarrow} K'$, $P_2 \stackrel{*}{\Longrightarrow} K'$.
2. Strictness: The transformations $\varrho_1 = (K \stackrel{p_1,m_1}{\Longrightarrow} P_1 \stackrel{*}{\Longrightarrow} K')$ and $\varrho_2 = (K \stackrel{p_2,m_2}{\Longrightarrow} P_2 \stackrel{*}{\Longrightarrow} K')$ commute, i.e., given the transformation morphisms $\hat{\varrho}_1$ and $\hat{\varrho}_2$, we have $\hat{\varrho}_1 = \hat{\varrho}_2$.

In order to show that all pairs of direct transformations in an HLR system are (strictly) locally confluent, we need to show that all parallel independent transformations and all parallel dependent transformations are (strictly) locally confluent. We will first show that all parallel independent direct transformations are strictly locally confluent.

Theorem 3.3.3 (Local Confluence of Parallel Independent Direct Transformations)**.** *Given an SPO HLR system where* $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ *is an SPO category, let* $t_1 = (G \stackrel{p_1,m_1}{\Longrightarrow} H_1)$ *and* $t_2 = (G \stackrel{p_2,m_2}{\Longrightarrow} H_2)$ *be two parallel independent direct transformations. Then* t_1 *and* t_2 *are strictly locally confluent.*

Our proof is based on the proof given by Ehrig et al. [13].

Proof. Consider the diagram below. Subdiagrams (1) and (2) depict the transformations t_1 and t_2, respectively.

$$\begin{array}{ccccc}
 & & L_1 & \xrightarrow{r_1} & R_1 \\
 & & \downarrow{\scriptstyle m_1} & (1) & \downarrow{\scriptstyle m_1'} \\
L_2 & \xrightarrow{m_2} & G & \xrightarrow{r_1'} & H_1 \\
\downarrow{\scriptstyle r_2} & (2) & \downarrow{\scriptstyle r_2'} & (3) & \downarrow{\scriptstyle r_2''} \\
R_2 & \xrightarrow{m_2'} & H_2 & \xrightarrow{r_1''} & X
\end{array}$$

The pushout diagram $(2) + (3)$ is the pushout over r_2 and $r_1' \circ m_2$, which forms the transformation $H_1 \stackrel{p_2, r_1' \circ m_2}{\Longrightarrow} X$ provided that $r_1' \circ m_2$ is a match, i.e., $r_1' \circ m_2 \in \mathcal{M}$. But this is ensured since t_1 and t_2 have been required to be parallel independent. Analogously we obtain a transformation $H_2 \stackrel{p_1, r_2' \circ m_1}{\Longrightarrow} X$ using the pushout $(1) + (3)$. Using the pushout decomposition property we know that (3) is a pushout (over r_1' and r_2'), the uniqueness of pushouts ensures that X is unique up to isomorphism.

It is clear that the transformations $\varrho_1 = (G \stackrel{p_1,m_1}{\Longrightarrow} H_1 \stackrel{p_2, r_1' \circ m_2}{\Longrightarrow} X)$ and $\varrho_2 = (G \stackrel{p_2,m_2}{\Longrightarrow} H_2 \stackrel{p_1, r_2' \circ m_1}{\Longrightarrow} X)$ commute because $\hat{\varrho}_1 = r_2'' \circ r_1' = r_1'' \circ r_2' = \hat{\varrho}_2$. □

Now that we have seen that all parallel independent transformations are locally confluent, we will now formulate a requirement for when all parallel dependent transformations are locally confluent. The next theorem we provide a sufficient condition for when an SPO HLR system is (strictly) locally confluent. For graph transformation systems, some of this theory has already been worked out by Löwe [27], however the proof for the local confluence theorem in that paper is very concise.

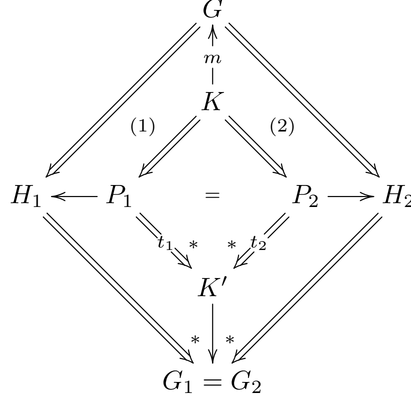


Figure 3.7: A pair of direct transformations which is strictly locally confluent because the critical pair is strictly locally confluent

Theorem 3.3.4 (Local Confluence Theorem for SPO HLR systems)**.** *An SPO HLR system for a strict SPO category* $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ *is strictly locally confluent if and only if all its critical pairs are strictly locally confluent.*

Proof. ($\Rightarrow$) Assume that the SPO HLR system is strictly locally confluent i.e., every pair of direct transformations $H_1 \overset{p_1,m_1}{\Longleftarrow} G \overset{p_2,m_2}{\Longrightarrow} H_2$ with the same source is strictly locally confluent. Since the set of critical pairs of the SPO HLR system is a subset of all the pairs of direct transformations with the same source, we can conclude that all critical pairs are strictly locally confluent.

($\Leftarrow$) Given a pair of direct transformations $H_1 \overset{p_1,m_1}{\Longleftarrow} G \overset{p_2,m_2}{\Longrightarrow} H_2$ we have to show the existence of transformations $\delta_1 = (G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{*}{\Longrightarrow} G_1)$ and $\delta_2 = (G \overset{p_2,m_2}{\Longrightarrow} H_2 \overset{*}{\Longrightarrow} G_2)$ such that $G_1 = G_2$ and $\hat{\delta_1} = \hat{\delta_2}$. If the given pair is parallel independent, then this follows from Theorem 3.3.3. If the given pair is parallel dependent, Theorem 3.2.5 implies the existence of a critical pair $P_1 \overset{p_1,m_1'}{\Longleftarrow} K \overset{p_2,m_2'}{\Longrightarrow} P_2$ and an $\mathcal{M}$-monomorphism $m : K \to G$ (see Figure 3.7).

By assumption the critical pair is strictly locally confluent, leading to transformations $t_1 = (P_1 \overset{*}{\Longrightarrow} K')$ and $t_2 = (P_2 \overset{*}{\Longrightarrow} K')$ such that the transformations $\varrho_1 = (K \overset{p_1,m_1'}{\Longrightarrow} P_1 \overset{*}{\Longrightarrow} K')$ and $\varrho_2 = (K \overset{p_2,m_2'}{\Longrightarrow} P_2 \overset{*}{\Longrightarrow} K')$ commute, i.e., $\hat{\varrho_1} = \hat{\varrho_2}$.

Since m is a monomorphism in $\mathcal{M}$ we know (by Definition 3.2.3) that $\hat{\varrho_1}$ and $\hat{\varrho_2}$ are strictly $\mathcal{M}$-preserving w.r.t. m. Theorem 3.1.12 implies the existence of transformations $\delta_1 = (G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{*}{\Longrightarrow} G_1)$ and $\delta_2 = (G \overset{p_2,m_2}{\Longrightarrow} H_2 \overset{*}{\Longrightarrow} G_2)$, with embeddings $e_1 : \varrho_1 \to \delta_1$ and $e_2 : \varrho_2 \to \delta_2$. Let x_1 be the last morphism of e_1 and let x_2 be the last morphism of e_2 then (by Theorem 3.1.12) $G \xrightarrow{\hat{\delta_1}} G_1 \xleftarrow{x_1} K'$ is the pushout over $G \xleftarrow{m} K \xrightarrow{\hat{\varrho_1}} K'$, and $G \xrightarrow{\hat{\delta_2}} G_2 \xleftarrow{x_2} K'$ is the pushout over $G \xleftarrow{m} K \xrightarrow{\hat{\varrho_2}} K'$. Because $\hat{\varrho_1} = \hat{\varrho_2}$, G_1 and G_2 are pushout objects over the same span, therefore we can assume w.l.o.g. that $G_1 = G_2$ and $\hat{\delta_1} = \hat{\delta_2}$. We can conclude that the pair of direct transformations $H_1 \overset{p_1,m_1}{\Longleftarrow} G \overset{p_2,m_2}{\Longrightarrow} H_2$ is strictly locally confluent. □

Chapter 4

Negative Application Conditions

Up to this point we have considered HLR systems where the rules only consisted of a rule morphism. In this chapter we introduce rules with *negative application conditions* (NACs). With NACs it is possible to specify when a rule may not be applied. In Figure 4.1 we see an graph L and a NAC $l : L \to \hat{L}$. Assuming that L is the left-hand side of some rule p then a match $m : L \to G$ for p is *applicable* only if there does not exist a morphism $n : \hat{L} \to G$ such that $n \circ l = m$; informally, the structure of $\hat{L}$ must not exist in G.

We call an SPO HLR system where rules are allowed to have NACs an *SPO HLR system with NACs*.

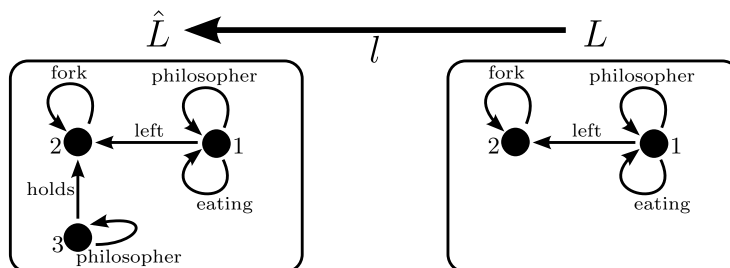


Figure 4.1: $\hat{L}$ is a NAC over L

First we will formally define what rules with NACs are. We will see that there are two possible types of NACs, namely left NACs and right NACs. In Section 4.2 discuss the conversion from right NACs to equivalent left NACs: we conjecture that this is possible in $\boldsymbol{Graph^P}$. Assuming that right NACs can be converted to left NACs, we will be able to formulate a new version of the embedding theorem for SPO HLR systems with NACs in Section 4.3. Finally in Section 4.4 we will define parallel independence and we will show some properties which must hold for critical pairs for SPO HLR systems with NACs. We will define critical pairs, and show that an SPO HLR system is (strictly) locally confluent if there are no critical pairs. Investigation of a necessary and sufficient condition for (strict) local confluence is future work.

4.1 Rules with NACs

In this section we will introduce (rules with) NACs, and we will prove some basic properties of NACs. We will start with a formal definition of NACs and rules with NACs.

Definition 4.1.1 (Negative Application Conditions)**.** Given an SPO category $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ and a rule p with rule morphism $r : L \to R$ we define the following:

1. A *negative application condition* (NAC) over L (resp. R) is an $\mathcal{R}$-morphism $l : L \to \hat{L}$ (resp. $l' : R \to \hat{R}$). We call l (resp. l') a *left* (resp. *right*) NAC on p.

2. An $\mathcal{M}$-morphism $m : L \to X$ *satisfies* a left NAC $l : L \to \hat{L}$, written $m \models l$, if there is no $\mathcal{M}$-morphism $n : \hat{L} \to X$ such that $n \circ l = m$.

3. An $\mathcal{M}$-morphism $m : L \to X$ *satisfies* a right NAC $l' : R \to \hat{R}$, written $m \models l'$, if, given the pushout $X \xrightarrow{r'} Y \xleftarrow{m'} R$ over r and m, if there is no $\mathcal{M}$-morphism $n : \hat{R} \to Y$ such that $n \circ l' = m'$.

4. A (left or right) NAC $l : L \to \hat{L}$ is *falsifiable* if there exists an $\mathcal{M}$-morphism $m : \hat{L} \to X$ such that $m \not\models l$.

5. A (left or right) NAC $l : L \to \hat{L}$ is *consistent* (or satisfiable) if there exists an $\mathcal{M}$-morphism $m : L \to X$ such that $m \models l$.

6. A *rule with negative application conditions* $p = (L \xrightarrow{r} R, A, B)$, or *rule* for short, consists of an $\mathcal{R}$-morphism r, a set A of NACs over L (left NACs) and a set B of NACS over R (right NACs).

7. Given a rule with NACs $p = (L \xrightarrow{r} R, A, B)$, an $\mathcal{M}$-morphism $m : L \to X$ satisfies a set A (resp. B) of NACs, written $m \models A$ (resp. $m \models B$) , if m satisfies all NACs in A (resp. B) .

8. Given a rule with NACs $p = (L \xrightarrow{r} R, A, B)$, we say that p is *applicable* to an object X at $m : L \to X$ (written $m \models p$) if $m \in \mathcal{M}$, $m \models A$ and $m \models B$.

Let $p = (L \xrightarrow{r} R, A, B)$ be a rule and let $l \in A$ be a NAC for p. If l is not falsifiable (i.e., every match m for p satisfies the NAC l), then this means that the rule $p' = (L \xrightarrow{r} R, A \setminus \{l\}, B)$ is applicable in the exact same situations as p. In the next proposition we will show when a left NAC is falsifiable.

Proposition 4.1.2 (Falsifiability of Left NACs)**.** *Let $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ be an SPO category; given a rule $p = (L \xrightarrow{r} R, A, B)$, a left NAC $l : L \to \hat{L} \in A$ is falsifiable if and only if $l \in \mathcal{M}$.*

Proof. ($\Rightarrow$) Let l be falsifiable, then there exist $\mathcal{M}$-morphisms $m : \hat{L} \to X$ and $n : \hat{L} \to X$ such that $n \circ l = m$. Since $\mathcal{M}$-morphisms are closed under decomposition we can conclude that $l \in \mathcal{M}$.

($\Leftarrow$) Let $l \in \mathcal{M}$. If we take $m = l$ and $n = id_{\hat{L}}$, then we have $n \circ l = m$, i.e. $m \not\models l$. We can conclude that l is falsifiable. $\square$

Remark. This does not hold for right NACs: given a rule $p = (L \xrightarrow{r} R, A, B)$ and an $\mathcal{M}$-morphism $m : L \to G$ then given the pushout $X \xrightarrow{r'} Y \xleftarrow{m'} R$ over r and m, the morphism m' may not be an $\mathcal{M}$-morphism.

If a NAC l is not falsifiable, then it does not add anything to the rule, since the same rule without the NAC l is applicable in exactly the same situations. From this point we will assume that every left NAC is falsifiable, which means that every left NAC l is both an $\mathcal{R}$- and an $\mathcal{M}$-morphism.

The next proposition states exactly when a NAC is consistent.

Proposition 4.1.3 (Consistency of NACs). *A (left or right) NAC $l : L \to \hat{L}$ is consistent if and only if there does not exist an $\mathcal{M}$-morphism $n : \hat{L} \to L$ such that $n \circ l = id_L$.*

Remark. Given morphisms $l : L \to \hat{L}$ and $n : \hat{L} \to L$ such that $n \circ l = id_L$, then we say that n is a *left inverse* (or retraction) of l.

Our proof is based on a similar proof in [13].

Proof. ($\Leftarrow$) If there does not exist an $\mathcal{M}$-morphism $n : \hat{L} \to L$ such that $n \circ l = id_L$, then $id_L \models l$, i.e., l is consistent by Definition 4.1.1.

($\Rightarrow$) Let $n : \hat{L} \to L$ be an $\mathcal{M}$-morphism such that $n \circ l = id_L$. Since for any given morphism $m : L \to X$ we have $m = m \circ id_L$, this implies that $m \circ n \circ l = m$, i.e., m does not satisfy the NAC l, therefore l is not consistent. ☐

4.2 Converting Right NACs to Left NACs

Given a match m for a rule $p = (L \xrightarrow{r} R, A, B)$, then deciding if the match satisfies all left NACs is easier than deciding if the match satisfies all right NACs. This is because for right NACs we need to compute the transformation (i.e., the pushout over m and r) before we can decide if the match satisfies the right NACs.

If there would exist a rule $p' = (L \xrightarrow{r} R, A', \varnothing)$ with the same rule morphism r, which has only left NACs, such that for any $\mathcal{M}$-morphism $m : L \to X$ we have $m \models p$ iff $m \models p'$ then it would be much easier to decide whether a rule is applicable. In this section we will investigate if there exists an equivalent rule p' (without right NACs) for any rule p such that p and p' are applicable in exactly the same situations.

One way to find such an equivalent rule, is to convert every right NAC into a number of left NACs. Consider a rule $p = (L \xrightarrow{r} R, A, B)$, where B is non-empty. Then we want to find a set of left NACs $A' = \{l_1 : L \to \hat{L_1}, \ldots, l_n \to \hat{L_n}\}$ such that for every $\mathcal{M}$-morphism m we have $m \models A'$ if and only if $m \models p$. To construct the set A' we will use *pushout complements*.

Definition 4.2.1 (Pushout Complement). Given morphisms $f : A \to B$ and $h : B \to D$, a *pushout complement* of (f, h) is an object C and morphisms $g : A \to C$ and $i : C \to D$ such that the cospan $C \xrightarrow{i} D \xleftarrow{h} B$ is the pushout over f and g. Two pushout complements C_j with morphisms $g_j : A \to C_j$ and $i_j : C_j \to D$ for $j = 1, 2$ are isomorphic when there exists an isomorphism $x : C_1 \to C_2$ such that $x \circ g_1 = g_2$ and $i_2 \circ x = i_1$.

Pushout complements may not always exist, even in categories where pushouts over all morphisms exist. In Figure 4.2 we show three situations where pushout complements do not exist in $\boldsymbol{Graph^P}$.

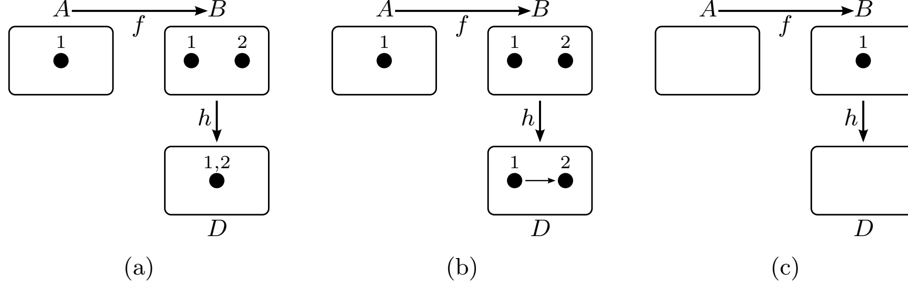


Figure 4.2: Examples of situations where no pushout complement exists

We conjecture that the non-existence of pushout complements does not cause any problems, because if there would not exists a pushout complement for a right NAC $l' : R \to \hat{R}$ and a rule morphism $r : L \to R$, then we conjecture that l' is not falsifiable. Consider any of the examples in Figure 4.2. If the morphism f would be a rule morphism, and h would be a right NAC, then we see that there is no way to falsify h, because (in all three examples) h models an additional side effect to one of the vertices which was added by the rule. We can conclude that h is not falsifiable.

Unfortunately this reasoning does not hold in general for all categories, the following example will illustrate this.

Example 4.2.2. Consider a category $\boldsymbol{C}$ with set of objects $\{L, R, \hat{R}, X, Y\}$ the morphisms are the identity morphisms for every object, the morphisms in the figure below, and their compositions.

$$
\begin{array}{ccc}
L & \xrightarrow{r} & R \\
 & & \downarrow{\scriptstyle l'} \\
\downarrow{\scriptstyle m} & & \hat{R} \\
 & & \downarrow{\scriptstyle n'} \\
X & \xrightarrow{r''} & Y
\end{array}
$$

Assume that every morphism is both an $\mathcal{M}$- and an $\mathcal{R}$-morphism. We claim (without proof) that $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ is an SPO category.

We can see that there exists no pushout complement for r, l' however there does exists an $\mathcal{M}$-morphism m, such that $m \models l'$, i.e., l' is falsifiable.

Next we will propose a construction which transforms a rule with right NACs to a rule with no right NACs. The purpose of this construction is that the rules are equivalent, meaning that they are applicable in the same situations and that they transform the same source graphs in the same target graphs.

Construction 4.2.3. Given a rule $p = (L \xrightarrow{r} R, A, B)$, we convert p to a rule $p' = (L \xrightarrow{r} R, A, \varnothing)$ in the following way:

1. Construct the set $A'_r = \{l : L \to \hat{L} \ \in \mathcal{M} \mid$ there exists a pushout $R \xrightarrow{l'} \hat{R} \xleftarrow{r'_i} \hat{L}$ over l and r such that $l' \in B\}$
2. $A' = A'_r \cup A$

The first step for showing that this fits the purpose, is proving that $m \models p$ implies $m \models p'$.

Proposition 4.2.4. *Given a rule $p = (L \xrightarrow{r} R, A, B)$ and a match $m : L \to X$. Let $p' = (L \xrightarrow{r} R, A', \varnothing)$ be the rule where all right NACs have been converted to left NACs as in Construction 4.2.3, then $m \models p$ implies $m \models p'$.*

Proof. Consider Figure 4.3, let $m \not\models p'$, then there exists a left NAC $l : L \to \hat{L} \ \in A'$ such that $m \not\models l$. If $l \in A$ then $m \not\models A$, otherwise l was part of a pushout complement of a right NAC in B, let $R \xrightarrow{l'} \hat{R} \xleftarrow{r^*} \hat{L}$ be the pushout (1) over l and r. We know that $l' \in B$ because l is part of a pushout complement of r and l'. Because $m \not\models l$, there exists an $\mathcal{M}$-morphism $n : \hat{L} \to X$ such that $n \circ l = m$. Let (2) be the pushout over r^* and n, then (1) + (2) is the pushout over r and $n \circ l = m$. We can conclude that $l' \circ n' \not\models l'$ therefore $m \not\models p$. □

Unfortunately the converse is not true for all categories, as the following example illustrates.

Example 4.2.5. Consider a category $\boldsymbol{C}$ with set of objects $\{L, R, \hat{L}, \hat{R}, X, Y\}$ the morphisms are the identity morphisms for every object, the morphisms in the figure below, and their compositions.

$$\begin{array}{ccc} L & \xrightarrow{r} & R \\ \downarrow l & (1) & \downarrow l' \\ \hat{L} & \xrightarrow{r'} & \hat{R} \\ & & \downarrow n' \\ X & \xrightarrow{r''} & Y \end{array} \qquad m : L \to X$$

Assume that every morphism is both an $\mathcal{M}$- and an $\mathcal{R}$-morphism. We claim (without proof) that $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ is an SPO category.

Let $p = (L \xrightarrow{r} R, \varnothing, B)$ be a rule where $B = \{l'\}$ (i.e., p has a right NAC). Using Construction 4.2.3 we can derive the rule $p' = (L \xrightarrow{r} R, A', \varnothing)$ where $A' = \{l\}$ and p' has no right NACs. We can see that $m \not\models p$ since the pushout over m and r is $X \xrightarrow{r''} Y \xleftarrow{n' \circ l'} R$. However there does not exist an $\mathcal{M}$-morphism $n : \hat{L} \to X$ such that $n \circ l = m$, and therefore $m \models l$ and $m \models p'$. We can conclude that $m \models p'$ does not imply $m \models p$.

Definition 4.2.6 (NAC equivalence property for SPO categories)**.** Given an SPO category $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$, we say that $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ satisfies the *NAC equivalence property* if for any rule $p = (L \xrightarrow{r} R, A, B)$, there exists a rule $p' = (L \xrightarrow{r} R, A', \varnothing)$ such that for any match $m : L \to X$ we have $m \models p'$ iff $m \models p$.

We conjecture that the NAC equivalence property holds for the SPO category $(\boldsymbol{Graph^P}, \mathcal{M}, \mathcal{R})$. In fact, we came up with a stronger property called

$$\begin{array}{ccc} L & \xrightarrow{r} & R \\ \downarrow{\scriptstyle l} & (1) & \downarrow{\scriptstyle l'} \\ \hat{L} & \xrightarrow{r^*} & \hat{R} \\ \downarrow{\scriptstyle n} & (2) & \downarrow{\scriptstyle n'} \\ X & \xrightarrow{r'} & Y \end{array} \qquad m : L \to X$$

Figure 4.3

the *pushout/pullback property* which we will define next. We will also show that the pushout/pullback property implies the NAC equivalence property. This property uses pullbacks, the dual of a pushout. More details on definitions and properties of pullbacks can be found in [10]. It is future work to find out if pullbacks exist for $\boldsymbol{Graph^P}$, and if the pushout/pullback property holds for $(\boldsymbol{Graph^P}, \mathcal{M}, \mathcal{R})$.

Definition 4.2.7 (Pushout/Pullback property)**.** Let $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ be an SPO category, then $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ satisfies the pushout/pullback property (PO/PB property) if for any pushout (1) + (2) over $m : L \to X$ and $r : L \to R$ where $m, n' \in \mathcal{M}$ and $r \in \mathcal{R}$, there exists a pullback (2) over r' and n' such that $n \in \mathcal{M}$, $n \circ l = m$, and (1) is a pushout (where $l : L \to \hat{L}$ is the mediating morphism: since $n' \circ l' \circ r = r' \circ m$ there must exist a unique morphism $l : L \to \hat{L}$ by the universal pullback property) (see Figure 4.3).

Now we will show that the PO/PB property indeed implies the NAC equivalence property.

Proposition 4.2.8. *Let $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ be an SPO category where the PO/PB property holds. Given a rule $p = (L \xrightarrow{r} R, A, B)$ and a match $m : L \to X$. Let $p' = (L \xrightarrow{r} R, A', \varnothing)$ be the rule where all right NACs have been converted to left NACs as in Construction 4.2.3, then $m \models p$ if and only if $m \models p'$ (i.e. the NAC equivalence property holds for $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$).*

Proof. ($\Rightarrow$) This follows from Proposition 4.2.4.

($\Leftarrow$) Consider Figure 4.3. Assume that $m \not\models p$. If there is a left NAC $l^* \in A$ such that $m \not\models l$ then $m \not\models p'$ (because $l^* \in A'$). Otherwise there must be a right nac $l' : R \to \hat{R} \in B$ and an $\mathcal{M}$-morphism $n' : \hat{R} \to Y$ such that $n' \circ l'$ is the co-morphism of m in the pushout $X \xrightarrow{r'} Y \xleftarrow{n' \circ l'} R$ over r and m. By the PO/PB property, there exist a morphism $l : L \to \hat{L}$ and a pullback $X \xleftarrow{n} \hat{L} \xrightarrow{r^*} \hat{R}$ over r' and n' such that $n \in \mathcal{M}$ and $\hat{L} \xrightarrow{r^*} \hat{R} \xleftarrow{l'} R$ is the pushout over l and r. We have $n \circ l = m \in \mathcal{M}$ and $n \in \mathcal{M}$ therefore $l \in \mathcal{M}$. Because (1) is a pushout we have found a pushout complement of r and l', therefore (by Construction 4.2.3) $l \in A'$ and $m \not\models l$ we have $m \not\models p'$. $\square$

From this point onwards we assume that every rule has only left NACs. Instead of writing $p = (L \xrightarrow{r} R, A, \varnothing)$ we leave out the empty set of right NACs and denote the rule as $p = (L \xrightarrow{r} R, A)$.

In the next section, most of our proofs will require that the NAC equivalence property holds. Proving that this property holds for $(\boldsymbol{Graph^P}, \mathcal{M}, \mathcal{R})$ is future work.

4.3 Derived Rules and Embeddings

In this section we will formulate a new embedding theorem which allows rules with NACs. In order to be able to do so we will define derived rules. Similarly to transformation morphisms (as defined in Definition 3.1.3), a derived rule is derived from a transformation. In fact, the rule morphism of a derived rule is the transformation morphism of the transformation. A derived rule also has a set of left NACs, which we will explain in more detail at a later point.

First we will define another assumption that we must make, namely that the SPO category that we use is closed under pushouts over $\mathcal{M}$-morphisms. The meaning of this is defined below.

Definition 4.3.1. Given an SPO category $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$, we say that $\mathcal{M}$-morphisms are closed under pushouts if for any $f : A \to B \in \mathcal{M}$, $g : A \to C \in \mathcal{M}$ and $g \in \mathcal{R}$ (g is both an $\mathcal{M}$- and an $\mathcal{R}$-morphism) then the pushout over (f, g) is also a pushout in the category with the same class of objects as $\boldsymbol{C}$ and the class of $\mathcal{M}$-morphisms[1].

Proposition 4.3.2. *In the SPO category* $(\boldsymbol{Graph^P}, \mathcal{M}, \mathcal{R})$*,* $\mathcal{M}$*-morphisms are closed under pushouts.*

Proof. $\mathcal{M}$-morphisms are total graph morphisms. We can use Construction 2.3.4 [1] to construct the pushout of two total graph morphisms, we can repeat the proof of Proposition 2.3.5 to show that pushouts exist in the category with the object class of all graphs and morphisms in the class $\mathcal{M}$. □

Next we will define derived rules. First we will define a so-called single-step derived rule. This is a rule (with a set of NACs), which is derived from a single-step transformation.

Definition 4.3.3 (Single-Step Derived Rule with NACs)**.** Given an SPO category $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ where $\mathcal{M}$ is closed under pushouts. Let $p = (L \xrightarrow{r} R, A)$ be a rule where $A = \{l_1, \ldots, l_n\}$ and let $\varrho = (X \overset{m,p}{\Longrightarrow} Y)$ be a transformation. The single-step derived rule $\bar{\varrho} = (X \xrightarrow{\hat{\varrho}} Y, A')$ of ϱ is defined as follows:

- $\hat{\varrho}$ is the transformation morphism of ϱ;
- $A' = \{l'_1, \ldots, l'_n\}$ such that, for $i = 1 \ldots n$, l'_i is the co-morphism of $l_i : L \to \hat{L}_i$ in the pushout $\hat{L}_i \xrightarrow{\hat{m}'_i} \hat{X}_i \xleftarrow{l'_i} X$ over m and l_i.

Remark. We know that every $l'_i \in A$ is an $\mathcal{M}$-morphism because $\mathcal{M}$-morphisms are closed under pushouts.

Next we will show that the definition we have just given is correct, in the sense that the NACs for the derived rule are satisfied under precisely the right circumstances.

[1]Unfortunately, our pushout construction is incorrect, therefore we our not sure if this proof is still valid, see Section 8.3 and Appendix A.2.

Proposition 4.3.4. *Let $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ be an SPO category where $\mathcal{M}$ is closed under pushouts. Given a rule $p = (L \xrightarrow{r} R, A)$, a transformation $\varrho = (G \overset{m,p}{\Longrightarrow} H)$ with derived rule $\bar{\varrho} = (G \xrightarrow{\hat{\varrho}} H, A')$, and an $\mathcal{M}$-morphism $t : G \to X$ then we have $t \models A'$ if and only if $m \circ t \models A$.*

Proof. ($\Rightarrow$) Suppose $m \circ t \not\models A$ then there exists a NAC $l : L \to \hat{L} \in A$ and an $\mathcal{M}$-morphism $n : \hat{L} \to X$ such that $n \circ l = t \circ m$. Let $\hat{L} \xrightarrow{m'} \hat{G} \xleftarrow{l'} G$ be the pushout (1) over l and m. By definition we know that $l' \in A'$. Because (1) is a pushout we know that there exists a morphism $n' : G \to X$ such that $t = n' \circ l'$, we know that n' is an $\mathcal{M}$-morphism because by Definition 4.3.1 the pushout over m and l is a pushout in the category with all $\boldsymbol{C}$ objects and $\mathcal{M}$-morphisms. We can conclude that $t \not\models l'$ which implies $t \not\models A'$.

$$\begin{array}{ccccc} \hat{L} & \xleftarrow{\;l\;} & L & \xrightarrow{\;r\;} & R \\ \downarrow{\scriptstyle m'} & (1) & \downarrow{\scriptstyle m} & & \downarrow{\scriptstyle m''} \\ \hat{G} & \xleftarrow{\;l'\;} & G & \xrightarrow{\;\hat{\varrho}\;} & H \\ \downarrow{\scriptstyle n'} & \swarrow{\scriptstyle t} & & & \\ X & & & & \end{array} \qquad (n : \hat{L} \to X)$$

($\Leftarrow$) Suppose $t \not\models A'$ then there exists a NAC $l' : G \to \hat{G} \in A'$ and an $\mathcal{M}$-morphism $n' : \hat{G} \to X$ such that $t = n' \circ l'$. By definition there exists a NAC $l \in A$ such that l' is the co-morphism in the pushout (1) over m and l. The morphism $n = n' \circ m'$ is an $\mathcal{M}$-morphism because $\mathcal{M}$-morphisms are closed under composition and pushouts. We have $n \circ l = t \circ m$ (because (1) is a pushout) and therefore $t \circ m \not\models l$ which implies $t \circ m \not\models A$. □

In the next construction we will show how we can construct a derived rule from any finite transformation sequence. This process requires converting all intermediate NACs in the transformation sequence to left NACs for the derived rule.

Construction 4.3.5 (Derived Rule with NACs)**.** Let $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ be an SPO category where $\mathcal{M}$ is closed under pushouts and the NAC equivalence property holds, let $\varrho = (G_0 \overset{p_1,m_0}{\Longrightarrow} \dots \overset{p_k,m_{k-1}}{\Longrightarrow} G_k)$ be a finite transformation sequence consisting of direct transformations $t_i = (G_{i-1} \overset{p_i,m_{i-1}}{\Longrightarrow} G_i)$ where $\bar{t_i} = (G_{i-1} \xrightarrow{\hat{t_i}} G_i, A_i)$ are the single-step derived rules (for $i = 1, \dots, n$). Then the *derived rule* $\bar{\varrho} = (G_0 \xrightarrow{\hat{\varrho}} G_k, A')$ constructed as follows:

1. $\hat{\varrho}$ is the transformation morphism of ϱ.
2. Let $\bar{t'_n} = \bar{t_n}$, and repeat the following for all $i = 1, \dots n-1$ in descending order:
   (a) Let A'_{i+1} be the set of left NACs of the rule $\bar{t'_{i+1}}$, and let A_i be the set of left NACs for $\bar{t_i}$.
   (b) Create the rule $\bar{t^*_i} = (G_{i-1} \xrightarrow{\hat{t_i}} G_i, A_i, A'_{i+1})$.

(c) Since the NAC equivalence property holds, we can construct the rule $\bar{t}'_i = (G_{i-1} \xrightarrow{\hat{t}_i} G_i, A'_i)$ which is equivalent to $\bar{t}^*_i$.

3. $A' = A'_1$ is the set of left NACs for $\bar{\varrho}$.

Remark. For a zero-step transformation $\delta = (X \overset{*}{\Longrightarrow} X)$ the derived rule has no NACs, i.e., $\bar{\delta} = (X \xrightarrow{id_X} X, \varnothing)$.

Before we formulate and prove our new embedding theorem, we will first prove a lemma that will aid our induction proof for the embedding theorem.

We will show that given two transformation sequences ϱ and ϱ', such that ϱ is a prefix of ϱ' (i.e., if ϱ has n transformation steps, then the first n transformation steps of ϱ' are the same as ϱ), then the derived rule of ϱ' has all the NACs of the derived rule of ϱ (and possibly more).

Lemma 4.3.6. *Let $\varrho' = (G_0 \overset{p_1,m_0}{\Longrightarrow} \dots \overset{p_n,m_{n-1}}{\Longrightarrow} G_n \overset{p_{n+1},m_n}{\Longrightarrow} G_{n+1})$ be a transformation of length $n+1$ and let $\varrho = (G_0 \overset{p_1,m_0}{\Longrightarrow} \dots \overset{p_n,m_{n-1}}{\Longrightarrow} G_n)$ be a a prefix of ϱ'. Then given the derived rules $\bar{\varrho}' = (G_0 \xrightarrow{\hat{\varrho}'} G_{n+1}, A')$ and $\bar{\varrho} = (G_0 \xrightarrow{\hat{\varrho}} G_n, A)$, we have $A \subseteq A'$.*

Proof. In Construction 4.3.5, we see that all intermediate NACs in a sequence of transformations are moved to the left (step by step). We also see that Construction 4.2.3 adds left NACs (and preserves the existing left NACs) i.e., given equivalent rule $\bar{t}'_i = (G_{i-1} \xrightarrow{\hat{t}_i} G_i, A'_i)$ of $\bar{t}^*_i = (G_{i-1} \xrightarrow{\hat{t}_i} G_i, A_i, A'_{i+1})$ we see that $A_i \subseteq A'_i$.

Now since the transformation ϱ is a prefix of ϱ', we see that Construction 4.3.5 has to iterate over all single-step derived rules of ϱ, even when computing the derived rule for ϱ'. Therefore $A \subseteq A'$ follows from Construction 4.3.5. $\square$

Next we prove the new embedding theorem which allows rules with NACs. The theorem itself is very similar to Theorem 3.1.12 (the embedding theorem without NACs), the difference is, that the morphism e_0 must now also satisfy all the NACs of the derived rule $\bar{\varrho}$.

Theorem 4.3.7 (Embedding Theorem for Rules with NACs)**.** *Let $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ be an SPO category where $\mathcal{M}$ is closed under pushouts and the NAC equivalence property holds, let $\varrho = (G_0 \overset{p_1,m_0}{\Longrightarrow} \dots \overset{p_k,m_{k-1}}{\Longrightarrow} G_k)$ be a transformation, let $\bar{\varrho} = (G_0 \xrightarrow{\hat{\varrho}} G_k, A')$ be the derived rule for ϱ, and let $e_0 : G_0 \to X_0$ be an $\mathcal{M}$-morphism such that $\hat{\varrho}$ is strictly $\mathcal{M}$-preserving w.r.t. e_0 and $e_0 \models \bar{\varrho}$, then there is a transformation $\delta = (X_0 \overset{p_1,n_0}{\Longrightarrow} \dots \overset{p_k,n_{k-1}}{\Longrightarrow} X_k)$ with an embedding $e : \varrho \to \delta$.*

Furthermore, let $e_k : G_k \to X_k$ be the last morphism of the embedding e. Then $X_0 \xrightarrow{\hat{\delta}} X_k \xleftarrow{e_k} G_k$ is the pushout over $X_0 \xleftarrow{e_0} G_0 \xrightarrow{\hat{\varrho}} G_k$.

Proof. Analogous the proof of Theorem 3.1.12. The only difference is the fact that we have NACs here. We will show that the induction step can still be applied. Given transformations $\varrho' = (G_0 \overset{p_1,m_0}{\Longrightarrow} \dots \overset{p_n,m_{n-1}}{\Longrightarrow} G_n \overset{p_{n+1},m_n}{\Longrightarrow} G_{n+1})$ and $\varrho = (G_0 \overset{p_1,m_0}{\Longrightarrow} \dots \overset{p_n,m_{n-1}}{\Longrightarrow} G_n)$ and an $\mathcal{M}$ morphism e_0 such that $\hat{\varrho}'$ is strictly $\mathcal{M}$-preserving w.r.t. e_0 and $e_0 \models \bar{\varrho}'$, this implies that $\hat{\varrho}$ is also strictly

$\mathcal{M}$-preserving w.r.t. e_0 (by Proposition 3.1.10) and we also know that $e_0 \models \bar{\varrho}$ because of Lemma 4.3.6.

Because we have $e_0 \models \bar{\varrho}$ we can apply the induction step. The rest of the proof (including the base case for the induction, where we do not have any NACs at all) is analogous to Theorem 3.1.12. □

4.4 Critical Pairs and Confluence

In Section 3.3 we have seen that the embedding theorem is important for proving that an HLR system (without NACs) is strictly locally confluent if all critical pairs are strictly locally confluent.

Given a transformation $\varrho = (G_0 \overset{*}{\Longrightarrow} G_n)$ and an $\mathcal{M}$-monomorphism $e_0 : G_0 \to X_0$, all conditions of the original embedding theorem (Theorem 3.1.12) are satisfied (because all $\mathcal{R}$-morphisms are strictly $\mathcal{M}$-preserving w.r.t. any monomorphism in $\mathcal{M}$). We cannot say the same for our new embedding theorem, since the fact that e_0 is a monomorphism does not necessarily mean that e_0 satisfies all NACs of the derived rule $\bar{\varrho}$. Because of this, finding a sufficient condition for strict local confluence for HLR systems with NACs is not easy.

In this section we will define parallel independence for transformations with NACs. We will define critical pairs with NACs. We will also show that it is not easy to prove strict local confluence of an HLR system based on analysis of the critical pairs, since critical pairs do not only depend on a pair of transformations, but they should take other rules in the HLR system into account.

Definition 4.4.1 (Parallel Independence)**.** Let $t_1 = (X \overset{p_1,m_1}{\Longrightarrow} Y_1)$ and $t_2 = (X \overset{p_2,m_2}{\Longrightarrow} Y_2)$ be two direct transformations using $p_1 = (L_1 \overset{r_1}{\longrightarrow} R_1, A_1)$ and $p_2 = (L_2 \overset{r_2}{\longrightarrow} R_2, A_2)$. Then t_1 and t_2 are *parallel independent* if $m_1^* = r_2' \circ m_1$ is a match for p_1, and $m_2^* = r_1' \circ m_2$ is a match for p_2 with $m_1^* \models p_1$ and $m_2^* \models p_2$. We call a pair of transformations *parallel dependent*, if they are not parallel independent.

From this definition we can see that there can be different reasons why a pair of transformations is parallel dependent. First of all, it can be the case that $m_1^* = r_2' \circ m_1$ is not a match for p_1, or $m_2^* = r_1' \circ m_2$ is not a match for p_2 (i.e., $m_1^* \notin \mathcal{M}$ or $m_2^* \notin \mathcal{M}$). This is a *delete-use conflict*, which we have already seen in Section 3.2 on page 31. The other reason why a pair of transformations can be parallel dependent is the situation where $m_1^* \not\models p_1$ or $m_2^* \not\models p_2$. Such a situation is called a *produce-forbid conflict*, since one rule produces something which is forbidden by one of the NACs of the other rule.

It is possible that a parallel dependent pair of transformations has both a delete-use conflict and a produce-forbid conflict, for instance if $m_1^* \notin \mathcal{M}$ and $m_2^* \in \mathcal{M}$ but $m_2^* \not\models p_2$.

Given the new definition for parallel independence, we can prove that all parallel independent pairs of transformations in a HLR system with NACs are strictly locally confluent.

Theorem 4.4.2 (Strict Local Confluence of Parallel Independent Direct Transformations)**.** *Given an SPO HLR system with NACs, let* $t_1 = (G \overset{p_1,m_1}{\Longrightarrow} H_1)$ *and*

$t_2 = (G \overset{p_2,m_2}{\Longrightarrow} H_2)$ *be two parallel independent direct transformations. Then* t_1 *and* t_2 *are strictly locally confluent.*

Proof. This follows from Definition 4.4.1 and Theorem 3.3.3. □

Analogously to what we have done in Section 3.2, we will give a definition for critical pairs with NACs, we call these *conditional critical pairs*. This definition is based on an existing definition for critical pairs with NACs for DPO graph transformation systems [24].

Definition 4.4.3 (Conditional Critical Pair)**.** Given an SPO HLR System with NACs, a *conditional critical pair* is a pair of direct transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ where $p_1 = (L_1 \xrightarrow{r_1} R_1, A_1)$ and $p_2 = (L_2 \xrightarrow{r_2} R_2, A_2)$, such that

1. m_1 and m_2 are jointly epimorphic and

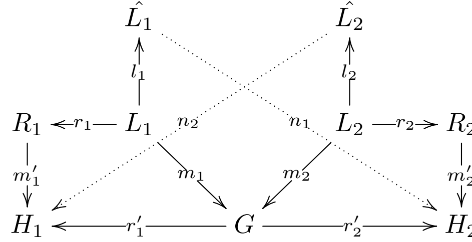
   (a) $m_1^* = r_2' \circ m_1$ is not an $\mathcal{M}$-morphism
   or

   (b) $m_2^* = r_1' \circ m_2$ is not an $\mathcal{M}$-morphism

   or

2. (a) $m_1^* = r_2' \circ m_1$ is an $\mathcal{M}$-morphism, but for one of the NACs $l_1 \in A_1$ there exists an $\mathcal{M}$-morphism $n_1 : \hat{L_1} \to H_2$ such that $n_1 \circ l_1 = m_1^*$ (i.e., $m_1^* \not\models l_1$) and the pair (n_1, m_2') is jointly epimorphic
   or

   (b) $m_2^* = r_1' \circ m_2$ is an $\mathcal{M}$-morphism, but for one of the NACs $l_2 \in A_2$ there exists an $\mathcal{M}$-morphism $n_2 : \hat{L_2} \to H_1$ such that $n_2 \circ l_2 = m_2^*$ (i.e., $m_2^* \not\models l_2$) and the pair (n_2, m_1') is jointly epimorphic



Remark. In cases 2(a) and 2(b) m_1 and m_2 may fail to be jointly epimorphic.

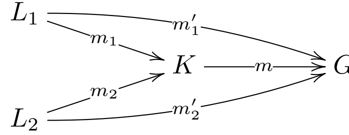
In order to be able to prove the completeness theorem for conditional critical pairs we have to make some assumptions. The next definition is an extension to strict SPO categories (Definition 3.2.3). The third part of this definition is fairly similar to what we have already defined in Definition 3.2.3 only the implications at the end ($m_1 \in \mathcal{M}$ and $m_2' \in \mathcal{M}$) are new.

Definition 4.4.4. A strict SPO category $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ is a *NAC SPO category* if the following properties hold:

1. The PO/PB property (see Definition 4.2.7) holds for $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$;

2. Pullbacks preserve monomorphisms, i.e., for morphisms $f : B \to D$ and $g : C \to D$ where f is a monomorphism and the pullback $B \xleftarrow{g'} A \xrightarrow{f'} C$ over f and g exists, it must follow that f' is a monomorphism.

3. For any pair of $m'_1 : L_1 \to G$ and $m'_2 : L_2 \to G$, there exists an object K and morphisms $m_1 : L_1 \to K, m_2 : L_2 \to K, m : K \to G$, such that m_1 and m_2 are jointly epimorphic, m is a monomorphism in $\mathcal{M}$, $m \circ m_1 = m'_1$ and $m \circ m_2 = m'_2$, furthermore $m'_1 \in \mathcal{M}$ implies $m_1 \in \mathcal{M}$ and $m'_2 \in \mathcal{M}$ implies $m_2 \in \mathcal{M}$.

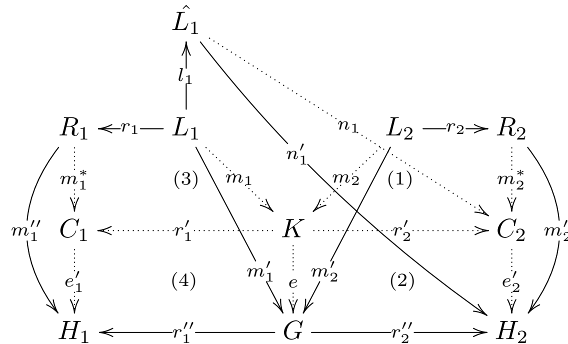


The following shows that every conflict (i.e., every pair of parallel dependent transformations) can be "explained" by a conditional critical pair. This is called completeness; it is the extension of the analogous result without NACs, see Theorem 3.2.5.

Theorem 4.4.5 (Completeness of Conditional Critical Pairs)**.** *Given an SPO HLR system with NACs where $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ is a NAC SPO category, for each pair of parallel dependent direct transformations $t'_1 = (G \overset{p_1,m'_1}{\Longrightarrow} H_1)$ and $t'_2 = (G \overset{p_2,m'_2}{\Longrightarrow} H_2)$, there exists a conditional critical pair $t_1 = (K \overset{p_1,m_1}{\Longrightarrow} C_1)$ and $t_2 = (K \overset{p_2,m_2}{\Longrightarrow} C_2)$ and a monomorphism $e : K \to G$ in $\mathcal{M}$ such that embeddings $\langle e, e'_i \rangle : t_i \to t'_i$ for $i = 1, 2$ exist.*

Proof. Let $r''_1 = \hat{t'_1}$ and $r''_2 = \hat{t'_2}$, in case $r''_2 \circ m_1 \notin \mathcal{M}$ or $r''_1 \circ m_2 \notin \mathcal{M}$ the situation is a delete-use conflict and our proof follows from Theorem 3.2.5.

It is also possible that (a) $r''_2 \circ m_1 \in \mathcal{M}$ and $r''_2 \circ m_1 \not\models p_1$, or (b) $r''_1 \circ m_2 \in \mathcal{M}$ and $r''_1 \circ m_2 \not\models p_1$. In this case we have a produce-forbid conflict. We assume the case (a) holds, the other case is analogous.



We have $r''_2 \circ m'_1$ is an $\mathcal{M}$-morphism, but for one of the NACs $l_1 \in A_1$ there exists an $\mathcal{M}$-morphism $n'_1 : \hat{L_1} \to H_2$ such that $n'_1 \circ l_1 = r''_2 \circ m'_1$ (i.e., $r''_2 \circ m'_1 \not\models l_1$).

We know by Definition 4.4.4 that there exists an object C_2 with morphisms $e'_2 : C_2 \to H_2$, $m^*_2 : R_2 \to C_2$ and $n_1 : \hat{L_1} \to C_2$ such that $n_1 \in \mathcal{M}$ and

$e'_2 \in \mathcal{M}$ which is, moreover, mono. By the PO/PB property there exists a pullback $G \overset{e}{\longleftarrow} K \overset{r'_2}{\longrightarrow} C_2$ over e'_2 and r''_2 with a morphism $m_2 : L_2 \to K$ such that $e \circ m_2 = m'_2$ and $K \overset{r'_2}{\longrightarrow} C_2 \overset{m*_2}{\longleftarrow} R_2$ is a pushout (1) over m_2 and r_2. We know that e is a monomorphism because e'_2 is mono and pullbacks preserve monomorphisms (Definition 4.4.4).

We must still show that there exists an $\mathcal{M}$-morphism $m_1 : L_1 \to K$ such that $e \circ m_1 = m'_1$. We have the morphisms $m'_1 : L_1 \to G$ and $n_1 \circ l_1 : L_1 \to C_2$ such that $r''_2 \circ m'_1 = e'_2 \circ n_1 \circ l_1$. Because (2) is a pullback this means that there exists a unique morphism $m_1 : L_1 \to K$ such that $r'_2 \circ m_1 = n_1 \circ l_1$ and $e \circ m_1 = m'_1$. Because e and m'_1 are $\mathcal{M}$-morphisms, we know that m_1 is also an $\mathcal{M}$-morphism (by the decomposition property of $\mathcal{M}$-morphisms), similarly, because e and $m'_2 = e \circ m_2$ are $\mathcal{M}$-morphisms, we know that m_2 is also an $\mathcal{M}$-morphism.

Before we can show that an embedding indeed exists, we will first show that m_1 and m_2 satisfy the NACs of p_1 and p_2. We show this by contradiction, assume that $m_1 \not\models p_1$ (the proof for $m_2 \not\models p_2$ is analogous) then there exists a NAC $l^*_1 : L_1 \to \hat{L}^*_1$ and an $\mathcal{M}$ morphism $n^* : \hat{L}^*_1 \to K$ such that $n^* \circ l^*_1 = m_1$. This means that $n^* \circ l^*_1 \circ e = m_1 \circ e = m'_1$, but then $m'_1 \not\models l^*_1$ which would mean that the transformation t_1 could not exist, a contradiction.

We know that the pullback (2) is also a pushout because (1) and (1) + (2) are pushouts. We can construct the pushouts (3) and (4). Because e is a monomorphism, we know (by Definition 3.2.3) that e'_1 is an $\mathcal{M}$-morphism (we have already shown that e'_2 is an $\mathcal{M}$-morphism).

Now we have conditional critical pair of transformations $t_i = (K \overset{p_i, m_i}{\Longrightarrow} C_i)$ (for $i = 1, 2$) and embeddings $\langle e, e'_i \rangle : t_i \to t'_i$. □

Now that we have shown that conditional critical pairs are complete, we can provide a sufficient condition for strict local confluence of an HLR system with NACs.

Theorem 4.4.6. *An SPO HLR system with NACs and a NAC SPO category* $(\boldsymbol{C}, \mathcal{M}, \mathcal{R})$ *is strictly locally confluent if there are no conditional critical pairs.*

Proof. Since there exists a conditional critical pair for every pair of parallel dependent pair of transformations, we know that every pair of transformations in the HLR system must be parallel independent. Therefore by Theorem 4.4.2 we know that all transformations are strictly locally confluent. □

One thing that we still need to find out is when an SPO HLR system with NACs is locally confluent if there are conditional critical pairs. Unfortunately it turns out that the strict local confluence of all conditional critical pairs does not imply strict local confluence of an SPO HLR system with NACs. We will illustrate this in the next example.

Example 4.4.7. Consider a graph transformation system with NACs, which has the rules $p_1 = (L_1 \overset{r_1}{\longrightarrow} R_1, \varnothing)$, $p_2 = (L_2 \overset{r_2}{\longrightarrow} R_2, \varnothing)$ and $p_3 = (L_3 \overset{r_3}{\longrightarrow} R_3, \{l\})$ as depicted in Figure 4.4 on the left side; and consider the pair of transformations $a = (Y_1 \overset{p_1}{\Longleftarrow} X \overset{p_2}{\Longrightarrow} Y_2)$ depicted in Figure 4.4 on the right which is in delete-use conflict. The conditional critical pair for this pair of transformations is $c = (H_1 \overset{p_1}{\Longleftarrow} G \overset{p_2}{\Longrightarrow} H_2)$ as depicted in Figure 4.4. We also see that there exists a monomorphism $e_0 : G \to X$.

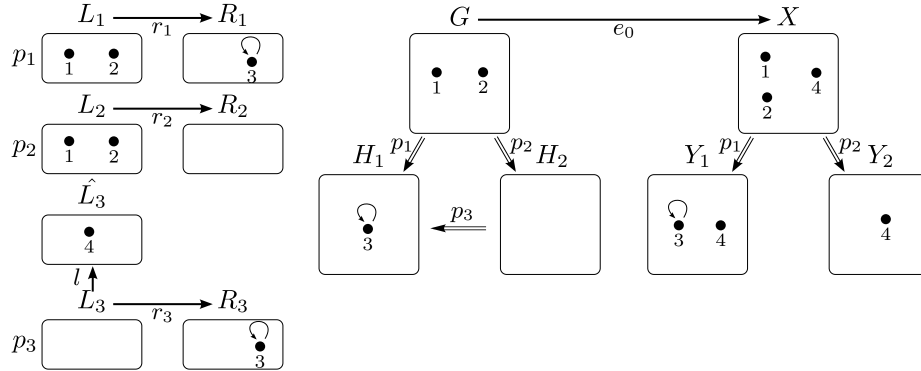


Figure 4.4: A situation where a critical pair is strictly locally confluent, but the same sequence of rules is not applicable to an embedding in a larger context. The rules are shown on the left, the critical pair in the middle, and the larger context on the right. We have numbered the vertices to show how they are mapped under their matches, the morphisms for the edges are defined by their source and target, and edges have the empty label ϵ.

We can see that the conditional critical pair c is strictly locally confluent, since there exists a transformation $H_2 \overset{p_3}{\Longrightarrow} H_1$ such that the transformation morphisms for $G \overset{p_1}{\Longrightarrow} H_1$ and $G \overset{p2}{\Longrightarrow} H_2 \overset{p_3}{\Longrightarrow} H_1$ commute (both transformation morphisms are empty morphisms).

One way to show that the pair of transformations a is strictly locally confluent, is by showing that there exists an embedding for the locally confluent transformations of the critical pair c into the transformations of a, however this is not the case, because the derived rule $\bar{\varrho}$ for $\varrho = (G \overset{p2}{\Longrightarrow} H_2 \overset{p_3}{\Longrightarrow} H_1)$ has a NAC which is not satisfied by $e_0 : G \to X$. We can conclude that the embedding theorem cannot be satisfied because $e_0 \not\models \bar{\varrho}$.

Informally, we can see that it is not possible to apply the transformations which made the critical pair c confluent to the parallel dependent pair a. This is because we cannot apply the rule p_3 to Y_2, since the NAC for p_3 is not satisfied for the match $m : L_3 \to Y_2$.

From this example we can conclude that strict local confluence of all critical pairs does not imply local confluence of all parallel dependent transformations in an SPO HLR system. Next, we will informally discuss some potential solutions to this problem based on related work on high-level replacement using double-pushout. Investigating if these solutions are also applicable to SPO HLR systems is future work.

Related Work

Lambers [23] has presented a method to analyse local confluence for adhesive HLR systems (which are HLR systems for DPO) with NACs. Similarly to our work, Lambers has defined critical pairs with NACs and an embedding theorem.

Like us, Lambers observed that strict local confluence of all (conditional) critical pairs was not sufficient for local confluence of the HLR system, therefore Lambers has defined *strict NAC-confluence*.

Definition 4.4.8 (Strict NAC-confluence)**.** A conditional critical pair of transformations $t_1 = (K \overset{p_1,m_1}{\Longrightarrow} C_1)$ and $t_2 = (K \overset{p_2,m_2}{\Longrightarrow} C_2)$ is *strictly NAC-confluent* if

1. it is strictly confluent via some transformations $\varrho_1 = (K \overset{p_1,m_1}{\Longrightarrow} C_1 \overset{*}{\Longrightarrow} X)$ and $\varrho_2 = (K \overset{p_2,m_2}{\Longrightarrow} C_2 \overset{*}{\Longrightarrow} X)$
2. and for every morphism $e_0 : K \to G \ \in \mathcal{M}$ where $e_0 \models \bar{t_1}$ and $e_0 \models \bar{t_2}$ (i.e., e_0 satisfies the derived rules of t_1 and t_2), it follows that $e_0 \models \bar{\varrho}_1$ and $e_0 \models \bar{\varrho}_2$ (i.e., e_0 satisfies the derived rules of ϱ_1 and ϱ_2).

Using this definition, we would be able to prove that an SPO HLR system is strictly locally confluent if all conditional critical pairs are strictly NAC-confluent (this proof would be analogous to Theorem 3.3.4, the strict NAC-confluence ensures that the embedding theorem with NACs is applicable).

What we do not yet know is when a conditional critical pair is strictly NAC-confluent. Lambers [23] provides a sufficient condition for strict NAC-confluence in the DPO setting, it should be investigated if these conditions also work for in the SPO setting.

We conclude this chapter with a note on future work. Strict NAC-confluence is not the only thing that is yet to be investigated. In order to apply these results to $(\boldsymbol{Graph^P}, \mathcal{M}, \mathcal{R})$, we must investigate whether the PO/PB property holds for this SPO category. Furthermore, in order to compute critical pairs and derived rules, we require a construction for pushout complements, we have not yet found such a construction for $\boldsymbol{Graph^P}$. This leaves a lot of future work.

Chapter 5

Attributed Graphs

In this chapter we will introduce attributed graphs and show that attributed graphs can be used for SPO high-level replacement. Attributed graphs are graphs where some vertices represent values, such as integers, strings, booleans or characters. These value vertices may only be the target of edges.

In Figure 5.1 we see an example of a rule with uses attributed graphs. In the rule the vertex labelled 1 in the attributed graph L represents a hungry philosopher. The edge 'forks' points to a vertex with the integer value 2, which means that the philosopher has two forks. This rule removes the edge with the label 'hungry', and adds a self edge with the label 'eating'.

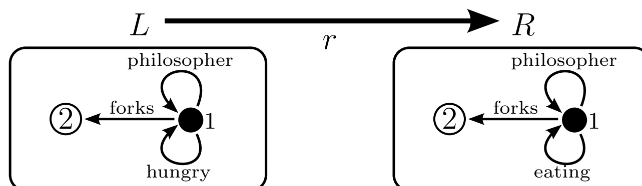


Figure 5.1: A rule which transforms a hungry philosopher with two forks to an eating philosopher with two forks.

In the graphs L and R we have shown only one value vertex. However, for every integer there exists a unique value vertex in both graphs L and R; we only display those value vertices which have incident edges.

In Section 5.1 we will introduce algebraic signatures and algebras, which are necessary to define attributed graphs. In Section 5.2 we will define attributed graphs and show that there exists an SPO category with attributed graphs as objects.

5.1 Algebraic Signatures and Algebras

In this section we introduce some necessary concepts which are needed to define attributed graphs. The following definitions are taken from [10].

Definition 5.1.1 (Algebraic Signature)**.** An *algebraic signature* $\Sigma = (S, OP)$, or signature for short, consists of a set S of sorts and a family of operation symbols $OP = (OP_{w,s})_{(w,s)\in S^*\times S}$.

For an operation $op \in OP_{w,s}$, we write $op : w \to s$ or $op : s_1, \ldots, s_n \to s$, where $w = s_1, \ldots, s_n$. If $w = \lambda$ then $op :\to s$ is called a constant symbol.

Example 5.1.2. The signature *SIG* below describes integers. *SIG* has a constant integer symbol *zero*, it has the operations *succ*, *pred*, *add*, and *mult* (successor, predecessor, addition and multiplication operations).

$$\begin{array}{ll} \mathit{SIG} = & \\ \mathit{sorts}: & \mathit{int} \\ \mathit{ops}: & \mathit{zero} :\to \mathit{int} \\ & \mathit{succ} : \mathit{int} \to \mathit{int} \\ & \mathit{pred} : \mathit{int} \to \mathit{int} \\ & \mathit{add} : \mathit{int}, \mathit{int} \to \mathit{int} \\ & \mathit{mult} : \mathit{int}, \mathit{int} \to \mathit{int} \end{array}$$

Definition 5.1.3 (Σ-algebra)**.** For a given signature $\Sigma = (S, OP)$, a Σ-algebra $A = ((D^A_s)_{s\in S}, (f^A_{op})_{op\in OP})$ is defined by

- for each sort $s \in S$ a set D^A_s, called the carrier set;
- for each symbol $op : s_1, \ldots, s_n \to s \in OP$, a function $f^A_{op} : A_{s_1} \times \cdots \times A_{s_n} \to A_s$.

We write $D^A = \bigcup_{s\in S} D^A_s$ for the union of all carrier sets.

Remark. The definition implies that for a constant symbol $c :\to s \ \in OP$, we have a value $f^A_c \in A_s$.

Definition 5.1.4 (Homomorphism)**.** Given a signature $\Sigma = (S, OP)$ and Σ-algebras A and B, a *homomorphism* $h : A \to B$ is a family $h = (h_s)_{s\in S}$ of total mappings $h_s : A_s \to B_s$, such that for each operation symbol $op : s_1, \ldots, s_n \to s \in OP$, we have $h_s(f^A_{op}(x_1, \ldots, x_n)) = f^B_{op}(h_{s_1}(x_1), \ldots, h_{s_n}(x_n))$ for all $x_i \in A_{s_i}$ (for $i = 0 \ldots n$). Homomorphisms can be composed: given $h : A \to B$ and $i : B \to C$, then $j = i \circ h : A \to C$ is a family of mappings $(j_s)_{s\in S}$ such that $\forall s \in S : j_s = i_s \circ h_s$

Remark. By this definition, we have $h_s(f^A_c) = f^B_c$ for each constant symbol $c :\to s \ \in OP$.

Example 5.1.5. We present an *SIG*-algebra A for the signature *SIG* defined in Example 5.1.2.

$$\begin{array}{lll} D^A_{\mathit{int}} & = \mathbb{Z} & \\ f^A_{\mathit{zero}} & = 0 \in D^A_{\mathit{nat}} & \\ f^A_{\mathit{succ}} & = x \mapsto x + 1 & : D^A_{\mathit{int}} \to D^A_{\mathit{int}} \\ f^A_{\mathit{pred}} & = x \mapsto x - 1 & : D^A_{\mathit{int}} \to D^A_{\mathit{int}} \\ f^A_{\mathit{add}} & = (x, y) \mapsto x + y & : D^A_{\mathit{int}} \times D^A_{\mathit{int}} \to D^A_{\mathit{int}} \\ f^A_{\mathit{mult}} & = (x, y) \mapsto x \cdot y & : D^A_{\mathit{int}} \times D^A_{\mathit{int}} \to D^A_{\mathit{int}} \end{array}$$

5.2 Attributed Graphs as an SPO Category

Now that we have presented the preliminaries, we can define attributed graphs. An attributed graph has a graph and an algebra, such that every carrier set of the algebra is a subset of the vertices of the graph. These value vertices may only be the target of edges.

Definition 5.2.1 (Attributed Graph, Attributed Graph Morphism)**.** Given a universal label alphabet L and a signature $\Sigma = (S, OP)$, a ***Σ****-attributed graph* $AG = (G_{AG}, A_{AG})$ is defined by

- A_{AG}, a Σ-algebra.
- $G_{AG} = (V_{AG}, E_{AG})$ is a graph (see Definition 2.1.1) with
  - V_{AG}, the set of vertices, such that $D^{AG} \subseteq V_{AG}$,
  - $E_{AG} \subseteq \hat{V}_{AG} \times L \times V_{AG}$, where $\hat{V}_{AG} = V_{AG} \setminus D^{AG}$

A (Σ-attributed graph) morphism between two Σ-attributed graphs $AG_i = (G_i, A_i)$ for $i = 1, 2$ is a tuple $h = (h_G : G_1 \to G_2,\ h_A : A_1 \to A_2)$ where $h_G = (h_V, h_E)$ is a partial graph morphism, and h_A is a total algebra homomorphism such that for all $s \in S$ we have $h_{A,s} = h_V|_{D_s^{A_1}}$. We say that h is total (resp. injective, surjective) if h_G and h_A are total (resp. injective, surjective).

Remark. We will often speak of attributed graphs (and not be explicit about the signature) instead of Σ-attributed graphs.

Since both algebra morphisms and graph morphisms can be composed, we can form the category $\boldsymbol{AGraph^P}$ having attributed graphs as objects and all attributed graph morphisms.

Since we want to show that there exists an SPO category using $\boldsymbol{AGraph^P}$, we need to define the morphism classes $\mathcal{M}$ and $\mathcal{R}$. $\mathcal{M}$ is the class of morphisms we can use as matches, therefore we require the graph component of an $\mathcal{M}$-morphism to be total, there is no (additional) restriction on the algebra component of the morphism (algebra homomorphisms are always total).

Definition 5.2.2 (class $\mathcal{M}$)**.** A Σ-attributed graph morphism $f : AG_1 \to AG_2$ with $f = (f_G, f_A)$ belongs to the class $\mathcal{M}$ if f_G is a total morphism.

The class $\mathcal{R}$ of rule morphisms requires that the algebra component of the $\mathcal{R}$-morphism is an isomorphism On the one hand this is natural (it would be strange if the data domain changes in the course of a transformation); at the same time this ensures the existence of pushouts over $\mathcal{R}$- and $\mathcal{M}$-morphisms.

Definition 5.2.3 (class $\mathcal{R}$)**.** A Σ-attributed graph morphism $f : AG_1 \to AG_2$ with $f = (f_G, f_A)$ belongs to the class $\mathcal{R}$ if f_A is an isomorphism of Σ-algebras.

Given a rule $p = (L \xrightarrow{r} R)$ where $L = (G_L, A_L)$ then it can be the case that for some Σ-algebra A there exists no algebra homomorphism $A_L \to A$. This means that for any Σ-attributed graph of the form $X = (X_G, A)$, there does not exist an attributed graph morphism $L \to X$. In order to ensure that the algebra does not restrict applicability of the rule, we can use the term algebra in the rule graphs. Given a signature Σ the term algebra $T_\Sigma(X)$ has homomorphisms to all other σ algebras. For more details see [10].

In order to show that $(\boldsymbol{AGraph^P}, \mathcal{M}, \mathcal{R})$ is an SPO category, we will first show pushouts exist in $\boldsymbol{AGraph^P}$ over $\mathcal{M}$- and $\mathcal{R}$ morphisms

Proposition 5.2.4 (Pushouts in $\boldsymbol{AGraph^P}$ over $\mathcal{M}$- and $\mathcal{R}$-morphisms)**.** *Given attributed graph morphisms* $f : AG_1 \to AG_2 \in \mathcal{R}$ *and* $g : AG_1 \to AG_3 \in \mathcal{M}$*, the pushout* $AG_2 \xrightarrow{f'} AG_4 \xleftarrow{g'} AG_3$ *over* $AG_2 \xleftarrow{f} AG_1 \xrightarrow{g} AG_3$ *is constructed as follows, where* $AG_i = (G_i, A_i)$ *for* $i = 1, \ldots, 4$.

$$\begin{array}{ccc} AG_1 & \xrightarrow{f=(f_G,f_A)} & AG_2 \\ \downarrow{\scriptstyle g=(g_G,g_A)} & (1) & \downarrow{\scriptstyle g'=(g'_G,g'_A)} \\ AG_3 & \xrightarrow{f'=(f'_G,f'_A)} & AG_4 \end{array}$$

- $G_2 \xrightarrow{f'_G} G_4 \xleftarrow{g'_G} G_3$ *is the pushout in* $\boldsymbol{Graph^P}$ *of* $G_2 \xleftarrow{f_G} G_1 \xrightarrow{g_G} G_3$. [1]
- *Let* $f'_G = (f'_V, f'_E)$, $A_4 = ((D_s^{A_4})_{s\in S}, (f_{op}^{A_4})_{op\in OP})$ *where*
  - $\forall s \in S : D_s^{A_4} = f'_V(D_s^{A_3})$
  - $\forall op \in OP : f_{op}^{A_4} = f'_V \circ f_{op}^{A_3}$
- $f'_A = (f'_{A,s})_{s\in S}$ *such that* $\forall s \in S : f'_{A,s} = f'_V|_{D_s^{A_3}}$
- $g'_A = (g'_{A,s})_{s\in S}$ *such that* $\forall s \in S : g'_{A,s} = (f'_V \circ g_{A,s} \circ f_{A,s}^{-1})$

Proof. We have $f'_G \circ g_G = g'_G \circ f_G$ and we know that $f'_A \circ g_A = g'_A \circ f_A$ because the algebra homomorphisms are defined by the graph morphisms. The pushout properties for the $\boldsymbol{Graph^P}$ component follows from the pushout construction in $\boldsymbol{Graph^P}$ [1], since the algebra homomorphisms are defined by the graph morphisms we know that the pushout properties also hold for the algebra component. We can conclude that (1) is a pushout.[1] □

In the example we have shown in Figure 5.1, the number of forks needed to be equal to two before the philosopher could start eating. In the next example we will show that it is also possible to change an attribute value. This is done by removing an edge to a value vertex and adding an edge to a different value vertex.

It is even possible (using some assumptions) to apply operations of a Σ-algebra A. Let A be the algebra defined in Example 5.1.5. Consider the operation f_{add}^A, then for every pair of integers we assume that there exists a product vertex, which has outgoing edges π_0 and π_1 (which represent the arguments) and *add* (which represents the result of the operation), such that $f_{add}^A(t(\pi_0), t(\pi_1)) = t(add)$. More details on this method can be found in [22].

Figure 5.2 shows an example of a rule where an operator is applied. The empty circles (with the number 3 or 4 next to them) are value vertices, their value is not specified (they are variables in the term algebra). The diamond shaped vertex is a product vertex which models the *add* operation.

The rule models a hungry philosopher which takes a free fork which on his left. When the rule is applied, the fork is no longer free (the philosopher holds it), and the edge 'forks' is replaced: the operation *add* is applied to the value vertex with the label 3 and the value vertex which represents the number 1, the result is the value vertex labelled 4. A new edge $(1, \text{forks}, 4)$ is added.

[1] Because our pushout construction for $\boldsymbol{Graph^P}$ is incorrect, it follows that the pushout construction for attributed graphs is also incorrect, see Section 8.3 and Appendix A.2.

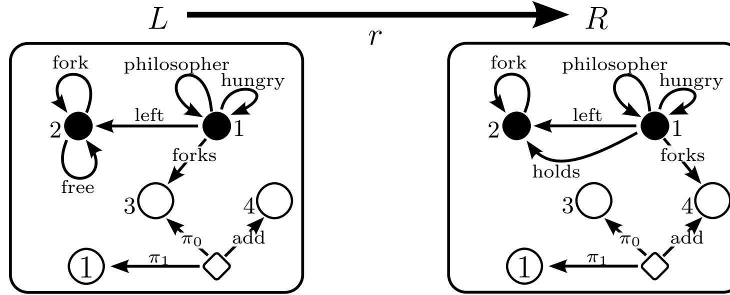


Figure 5.2: A rule which models a hungry philosopher which takes a free left fork. The number of forks is incremented by one.

We want to show that $(\boldsymbol{AGraph^P}, \mathcal{M}, \mathcal{R})$ is a strict SPO category, before we can show this we will first show that the critical pair property (which ensures the existence and completeness of critical pairs, see Theorem 3.2.5) holds.

Lemma 5.2.5 (Critical Pair Property for $\boldsymbol{AGraph^P}$)**.** *For any pair of $\mathcal{M}$-morphisms $m'_1 : X \to G$ and $m'_2 : Y \to G$, there exists an attributed graph K and $\mathcal{M}$-morphisms $m_1 : X \to K, m_2 : Y \to K, m : K \to G$, such that m_1 and m_2 are jointly epimorphic, m is a monomorphism, $m \circ m_1 = m'_1$ and $m \circ m_2 = m'_2$:*

$$\begin{array}{ccccc} X & \xrightarrow{m_1} & K & \xrightarrow{m} & G \\ Y & \xrightarrow{m_2} & K & & \end{array} \qquad X \xrightarrow{m'_1} G, \quad Y \xrightarrow{m'_2} G$$

Proof. We have already proven this for graphs in Proposition 2.4.2, therefore we can construct $G_K \subseteq G_G$ and the morphisms $m_{1,G}, m_{2,G}$ and m_G analogously to Proposition 2.4.2. Since $G_K \subseteq G_G$ we know that $D^{A_G} \subseteq V_K$, therefore we can take $A_K = A_G$, $m_A = id_{G_A}$, $m_{1,A} = m'_{1,A}$ and $m_{2,A} = m'_{2,A}$. Clearly m is a monomorphism because the components m_G and m_A are monomorphisms. We must show that $m'_{1,A}$ and $m'_{2,A}$ are jointly epimorphic, this follows from the fact that $m'_{1,G}$ and $m'_{2,G}$ are jointly epimorphic and Definition 5.2.1. We know m_1, m_2 and m are $\mathcal{M}$-morphisms because their graph components are total graph morphisms (see Proposition 2.4.2). □

Using these results we can show that $(\boldsymbol{AGraph^P}, \mathcal{M}, \mathcal{R})$ is a strict SPO category.

Proposition 5.2.6. *$(\boldsymbol{AGraph^P}, \mathcal{M}, \mathcal{R})$ is a strict SPO category.*[2]

Proof. First we show that $(\boldsymbol{AGraph^P}, \mathcal{M}, \mathcal{R})$ is an SPO category

1. $\boldsymbol{C}$ has pushouts over any morphism span $B \xleftarrow{f} A \xrightarrow{g} C$, if $f \in \mathcal{M}$ and $g \in \mathcal{R}$ (or vice versa), this follows from Proposition 5.2.4.[2]

[2]Because our pushout construction for $\boldsymbol{Graph^P}$ is incorrect, it follows that the pushout construction for attributed graphs is also incorrect, see Section 8.3 and Appendix A.2.

2. $\mathcal{M}$ is closed under composition because total graph morphisms are closed under composition. $\mathcal{R}$ is closed under composition because (algebra) isomorphisms are closed under composition.

3. $\mathcal{M}$ is closed under decomposition, because total graph morphisms are closed under decomposition. $\mathcal{R}$ is closed under decomposition, because (algebra) isomorphisms are closed under decomposition.

Next we show that $(\boldsymbol{AGraph^P}, \mathcal{M}, \mathcal{R})$ is also a strict SPO category

1. For any morphism span $G \stackrel{m}{\longleftarrow} L \stackrel{r}{\longrightarrow} R$, with $m \in \mathcal{M}$ and $r \in \mathcal{R}$, the co-morphism r' of r in the pushout $R \stackrel{m'}{\longrightarrow} H \stackrel{r'}{\longleftarrow} G$ over m and r is an $\mathcal{R}$-morphism, i.e., $r' \in \mathcal{R}$. To show that $r' \in \mathcal{R}$ we must show that r'_A is an isomorphism; this is the case because the algebra components of the morphisms m, r, m' and r' form a pushout in the category of algebras. By Proposition 2.3.7 we know that r'_A must be an isomorphism.

2. All $\mathcal{R}$-morphisms are strictly $\mathcal{M}$-preserving w.r.t. any monomorphism in $\mathcal{M}$; this follows from the that this property holds for $(\boldsymbol{Graph^P}, \mathcal{M}, \mathcal{R})$ (Proposition 2.4.2)

3. For any pair of $\mathcal{M}$-morphisms $m'_1 : L_1 \to G$ and $m'_2 : L_2 \to G$, there exists an object K and $\mathcal{M}$-morphisms $m_1 : L_1 \to K, m_2 : L_2 \to K, m : K \to G$, such that m_1 and m_2 are jointly epimorphic, m is a monomorphism, $m \circ m_1 = m'_1$ and $m \circ m_2 = m'_2$. This follows from Lemma 5.2.5. $\square$

Chapter 6

Efficient Confluence Detection

When computing all critical pairs for a pair of rules, it is possible that there is a very large number of them (we will elaborate on this in Section 7.2). In order to analyse if a HLR system is (strictly) locally confluent, we need to know whether all critical pairs are strictly locally confluent. In this chapter we will investigate if it is possible to avoid the need to analyse all critical pairs for this purpose.

In Section 6.1 we will discuss *essential critical pairs* (based on [26]), which are a subset of the set of critical pairs, such that all critical pairs are strictly locally confluent if and only if all essential critical pairs are strictly locally confluent. It turns out that the requirements on these essential critical pairs are too strict to give a significant improvement: the set of essential critical pairs is not significantly smaller than the set of critical pairs.

Section 6.2 will discuss another method to determine that a critical pair is strictly locally confluent. Given a strictly locally confluent critical pair a, then it is possible that the confluent transformations of a can be embedded in another critical pair b. This means that b is also strictly locally confluent. We will state a sufficient condition for when this is the case.

6.1 Essential Critical Pairs

Our original hypothesis what that it was possible to determine a subset of the set of all critical pairs called *essential critical pairs*, which represents all critical pairs, in the sense that all critical pairs are strictly locally confluent if and only if all essential critical pairs are strictly locally confluent. When analysing an HLR system for (strict) local confluence, only the set of essential critical pairs would need to be computed and checked for strict local confluence.

We need to find out when a critical pair is essential. Consider Figure 6.1 which shows the critical pairs $a = (X_1 \overset{p_1,m_1}{\Longleftarrow} A \overset{p_2,m_2}{\Longrightarrow} X_2)$ and $b = (Y_1 \overset{p_1,m \circ m_1}{\Longleftarrow} B \overset{p_2,m \circ m_2}{\Longrightarrow} Y_2)$. We know that the critical pair b is not essential if the strict local confluence of a implies the strict local confluence of b.

We should therefore determine when the strict local confluence of a implies

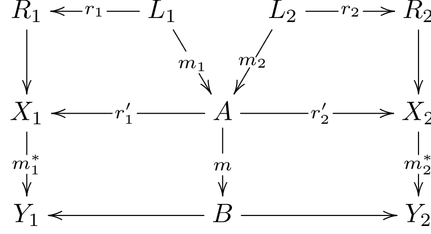


Figure 6.1: The critical pair $X_1 \overset{p_1,m_1}{\Longleftarrow} A \overset{p_2,m_2}{\Longrightarrow} X_2$ can be embedded in the critical pair $Y_1 \overset{p_1,m\circ m_1}{\Longleftarrow} B \overset{p_2,m\circ m_2}{\Longrightarrow} Y_2$ if m, m_1^* and m_2^* are $\mathcal{M}$-morphisms

the strict local confluence of b. Let $a_i = (A \overset{p_i,m_i}{\Longrightarrow} X_i)$ and $b_i = (B \overset{p_i,m\circ m_i}{\Longrightarrow} Y_i)$ for $i = 1,2$ be the transformations induced by the critical pairs. The first requirement is that the transformations a_i can be embedded in the transformations b_i (for $i = 1,2$), i.e., the embeddings $e_i = \langle m, m_i^* \rangle : a_i \to b_i$ exist (for $i = 1,2$). However, this is not yet sufficient to prove that b is strict locally confluent if and only if a is strict locally confluent. If a is strictly locally confluent, then there exist transformations $t_i = (A \overset{p_i,m_i}{\Longrightarrow} X_i \overset{*}{\Longrightarrow} Y)$ for $i = 1,2$ such that $\hat{t_1} = \hat{t_2}$. We can follow a reasoning similar to the proof of Theorem 3.3.4 if we know that $\hat{t_1}$ (and therefore also $\hat{t_2}$) is strictly $\mathcal{M}$-preserving w.r.t. m. However we do not know enough about $\hat{t_1}$ and m to be able to decide this.

In the next definition we state when a strictly locally confluent critical pair represents another critical pair. It formalizes when the strict local confluence of one critical pair implies the strict local confluence of another critical pair.

Definition 6.1.1 (Representation)**.** Let $a = (X_1 \overset{p_1,m_1}{\Longleftarrow} A \overset{p_2,m_2}{\Longrightarrow} X_2)$ and $b = (Y_1 \overset{p_1,n_1}{\Longleftarrow} B \overset{p_2,n_2}{\Longrightarrow} Y_2)$ be critical pairs such that a is strictly locally confluent, i.e., there exist commuting transformations $\varrho_i = (A \overset{p_i,m_i}{\Longrightarrow} X_i \overset{*}{\Longrightarrow} K)$ for $i = 1,2$. Then we say that a *represents* b if there exists a $\mathcal{M}$ morphism $m : A \to B$ and transformations $\delta_i = (B \overset{p_i,n_i}{\Longrightarrow} Y_i \overset{*}{\Longrightarrow} N)$ with embeddings $e_i : \varrho_i \to \delta_i$ for $i = 1,2$ such that m is the embedding morphism for both embeddings and $\hat{\delta_1} = \hat{\delta_2}$.

Remark. By definition of embeddings we have $n_i = m \circ m_i$ for $i = 1,2$. We know that m is an epimorphism since $m \circ m_1$ and $m \circ m_2$ are jointly epimorphic (else b would not be a critical pair). Furthermore by this definition we know that if a represents b then both a and b are strictly locally confluent.

Next we will formalize when a critical pair is an essential critical pair. Given a critical pair, we can not use the definition we have just given because we do not yet know if this critical pair is strictly locally confluent, and (if the pair is strictly locally) what the transformations are that make the pair strictly locally confluent. However using theorem Theorem 3.1.12 and the fact that (by Definition 3.2.3) every monomorphism is strictly $\mathcal{M}$-preserving, we can conclude that a represents b if m is a monomorphism. However, since m is also an epimorphism, m will be an isomorphism in set- and graph-based categories (Proposition 2.2.13), which means that a and b are the same critical pair.

An alternative is to require that m_1^* and m_2^* are monomorphisms. Assuming a is strictly locally confluent i.e., there exist transformations $\varphi_i = (X_i \overset{*}{\Longrightarrow} K)$

for $i = 1, 2$. Since any morphism is strictly $\mathcal{M}$-preserving w.r.t. any monomorphism, Theorem 3.1.12 implies the existence of transformations $\sigma_i = (Y_i \overset{*}{\Longrightarrow} N)$ and embeddings $e_i^* : \varphi_i \to \sigma_i$ for $i = 1, 2$. This way we can show that b is strictly locally confluent if a is strictly locally confluent.

Definition 6.1.2 (Essential Critical Pair)**.** Let b be a critical pair with transformations $b_i = (B \overset{p_i, n_i}{\Longrightarrow} Y_i)$ (for $i = 1, 2$). We say that b is an *essential critical pair* if there exists no critical pair a with transformations $a_i = (A \overset{p_i, m_i}{\Longrightarrow} X_i)$ and embeddings $e_i = \langle m : A \to B,\ m_i^* : X_i \to Y_i \rangle : a_i \to b_i$ (for $i = 1, 2$) such that m_1^* and m_2^* are monomorphisms (see Figure 6.1).

The requirement that both m_1^* and m_2^* are monomorphisms is very strict. We are looking for critical pairs with transformations $a_i = (A \overset{p_i, m_i}{\Longrightarrow} X_i)$ and $b_i = (B \overset{p_i, m \circ m_i}{\Longrightarrow} Y_i)$ for $i = 1, 2$ (see Figure 6.1), such that m is not a monomorphism (else m would be an isomorphism), and both m_1^* and m_2^* are monomorphisms. In the category $\boldsymbol{Graph^P}$, this is the case when there are at least two vertices in A which are deleted by both rules, furthermore, m must be non-injective on those two vertices and m must be injective on all elements of A which are not being deleted (by either rule).

An example of such a case is shown in Figure 6.2. Here we see the critical pair $a = (X_1 \overset{p_1, m_1}{\Longleftarrow} A \overset{p_2, m_2}{\Longrightarrow} X_2)$ which is embedded into the critical pair $b = (Y_1 \overset{p_1, m \circ m_1}{\Longleftarrow} B \overset{p_2, m \circ m_2}{\Longrightarrow} Y_2)$, such that m_1^* and m_2^* are monomorphisms.

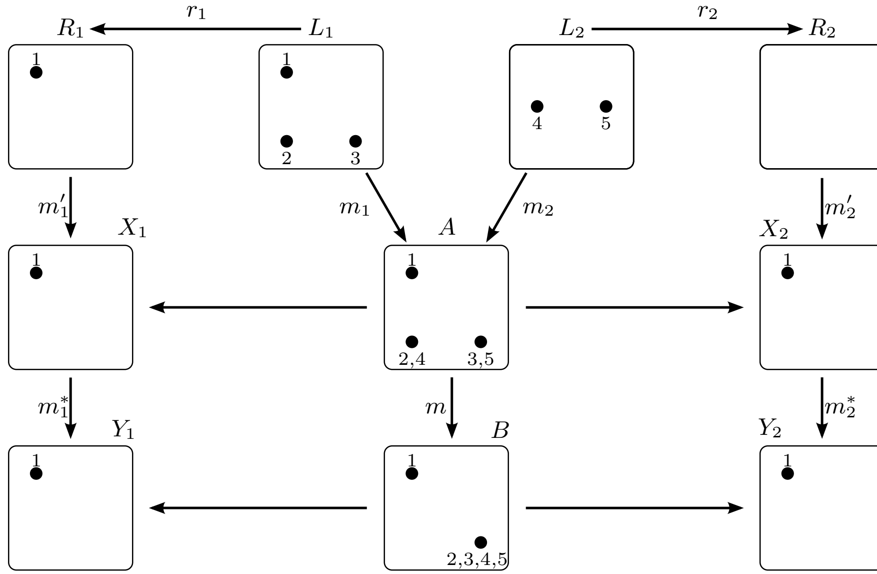


Figure 6.2: The critical pair $Y_1 \overset{p_1, m \circ m_1}{\Longleftarrow} B \overset{p_2, m \circ m_2}{\Longrightarrow} Y_2$ is not an essential critical pair.

Because of the strict requirement on m, the size of the set of essential critical pairs will not differ very much from the set of essential critical pairs (which we will also see in our case study, see Section 7.5). Hence the notion of essential critical pairs is probably not very useful. Therefore in the next section, we will

work out a theory which allows us to decide if a critical pair is strictly locally confluent, based on the fact that another pair is strictly locally confluent.

6.2 Subsumption of Critical Pairs

Because in general the set of essential critical pairs is not significantly smaller than the set of critical pairs, we propose another method to analyse whether a critical pair is strictly locally confluent, based on the fact that some other critical pair is locally confluent.

Given a strictly locally confluent critical pair a, we can check if a can represent (see Definition 6.1.1) some other critical pair b using the sufficient condition we provide in the next proposition.

Proposition 6.2.1. *Let $a = (X_1 \overset{p_1,m_1}{\Longleftarrow} A \overset{p_2,m_2}{\Longrightarrow} X_2)$ and $b = (Y_1 \overset{p_1,n_1}{\Longleftarrow} B \overset{p_2,n_2}{\Longrightarrow} Y_2)$ be critical pairs such that a is strictly locally confluent, i.e., there exist transformations $\varrho_i = (A \overset{p_i,m_i}{\Longrightarrow} X_i \overset{*}{\Longrightarrow} K)$ for $i = 1,2$ such that $\hat{\varrho_1} = \hat{\varrho_2}$. Then a represents b if there exists a $\mathcal{M}$-morphism m such that $\hat{\varrho_1}$ is strictly $\mathcal{M}$-preserving w.r.t. m.*

Proof. We know that there exist transformations $\delta_i = (B \overset{p_i,n_i}{\Longrightarrow} Y_i \overset{*}{\Longrightarrow} N)$ with embeddings $e_i : \varrho_i \to \delta_i$ by Theorem 3.1.12. We know that the object N is the same (modulo isomorphism) for both transformations and $\hat{\delta_i} = \hat{\delta_2}$ because (by Theorem 3.1.12) N is the pushout object of the same span of morphisms $B \overset{m}{\longleftarrow} A \overset{\hat{\varrho_1}=\hat{\varrho_2}}{\longrightarrow} K$. □

When analysing (strict) local confluence of an HLR system, the order in which critical pairs are checked for (strict) local confluence is important if we want to avoid unnecessary local confluence checks. If at some point during our (strict) local confluence analysis we conclude that the critical pair a represents b, then it would be a shame if we had already computed an example to show that b was strictly confluent. Therefore we should analyse the critical pair a before b if a weakly represents b, in the following sense:

Definition 6.2.2 (Weak Representation)**.** Let $a = (X_1 \overset{p_1,m_1}{\Longleftarrow} A \overset{p_2,m_2}{\Longrightarrow} X_2)$ and $b = (Y_1 \overset{p_1,n_1}{\Longleftarrow} B \overset{p_2,n_2}{\Longrightarrow} Y_2)$ be critical pairs. We say that a *weakly represents* b if there exists an $\mathcal{M}$-morphism $m : A \to V$ such that $m \circ m_1 = n_1$ and $m \circ m_2 = n_2$.

For graphs as defined in Definition 2.1.1 it is sufficient to order by number of vertices. We will show this in the next proposition.

Proposition 6.2.3. *Given two critical pairs $a = (X_1 \overset{p_1,m_1}{\Longleftarrow} A \overset{p_2,m_2}{\Longrightarrow} X_2)$ and $b = (Y_1 \overset{p_1,n_1}{\Longleftarrow} B \overset{p_2,n_2}{\Longrightarrow} Y_2)$ in the category $\boldsymbol{Graph^P}$, then $|V_A| < |V_B|$ implies a does not weakly represent b.*

Proof. Assume that a weakly represents b, then we have a total graph morphism $m : A \to B$ such that $m \circ m_1 = n_1$ and $m \circ m_2 = n_2$. Since the pairs (m_1, m_2) and (n_1, n_2) are jointly surjective, this means that m must be an epimorphism. However, since $|V_A| < |V_B|$ the mappings for the vertices cannot be surjective. Therefore a total and surjective morphism $m : A \to B$ cannot exist, which means a does not represent b. □

Next we will show that given two critical pairs $a = (X_1 \overset{p_1,m_1}{\Longleftarrow} A \overset{p_2,m_2}{\Longrightarrow} X_2)$ and $b = (Y_1 \overset{p_1,n_1}{\Longleftarrow} B \overset{p_2,n_2}{\Longrightarrow} Y_2)$ where $|V_A| = |V_B|$, we do not need to check whether a weakly represents b, because if a represents b, then the critical pairs are isomorphic.

Before we can show this, we will first shown that a total graph morphism is uniquely defined by the vertex component of the morphism.

Proposition 6.2.4. *Given graphs G and H, and a total set morphism $f_V : V_G \to V_H$ the edge component of the total graph morphism $f = (f_V, f_E) : G \to H$ is uniquely defined by f_V.*

Proof. Because f is total, f_E must also be total. For every edge $e = (s, l, t) \in E_G$ we have $f_E(e) = (f_V(s), l, f_V(t))$ because edges must preserve sources, targets and labels (see Definition 2.1.3). □

Proposition 6.2.5. *Given a two critical pairs $a = (X_1 \overset{p_1,m_1}{\Longleftarrow} A \overset{p_2,m_2}{\Longrightarrow} X_2)$ and $b = (Y_1 \overset{p_1,n_1}{\Longleftarrow} B \overset{p_2,n_2}{\Longrightarrow} Y_2)$ in the category $\boldsymbol{Graph^P}$, such that $|V_A| = |V_B|$ and a represents b, then $A \cong B$.*

Proof. Because a weakly represents b, we have a total graph morphism $m : A \to B$ such that $m \circ m_1 = n_1$ and $m \circ m_2 = n_2$. We know that m is an epimorphism, because the pairs (m_1, m_2) and (n_1, n_2) are jointly surjective. Since $|V_A| = |V_B|$ we know that the vertex mapping m_V must be one-to-one, this means m_V is injective. Since m_V was also surjective, we know m_V is an isomorphism.

We know that (Proposition 6.2.4) m_E is uniquely defined by m_V. The edge mapping m_E must be injective because the mapping of the source and target vertices of every edge are injective. We can conclude that m_E and m_V are both surjective and injective, which means m is an isomorphism, and $A \cong B$. □

In order to apply our new method for critical pair detection, we need to order the critical pairs by decreasing size of the host graph. This ensures that, when analysing a critical pair a, all critical pairs with a larger host graph have already been analysed. If there exist a critical pair b which is strictly locally confluent and uses the same rules as a, then we can analyse (using Proposition 6.2.1) if a is also strictly locally confluent. If Proposition 6.2.1 is not applicable, then there may exist other critical pairs which can represent a, or the regular method for confluence analysis can be applied to determine whether a is strictly locally confluent.

Since our method can only decide if a critical pair is strictly locally confluent, but not the converse, this method will not be faster when analysing critical pairs which are not strictly locally confluent. In Section 7.5 we will see how this method performs in practice.

Chapter 7

Implementation and Experiments

In this chapter we will analyse how well critical pair detection and local confluence analysis work in practice. First we will discuss the algorithm that we have implemented to find all critical pairs for two rules, this will be followed by a discussion on the complexity of this algorithm in Section 7.2. Section 7.3 will discuss how we can analyse whether a critical pair is locally confluent. The rest of the chapter will be about the critical pair and confluence analysis in practice. We will show some graph transformation systems and analyse if these are strictly locally confluent.

7.1 Critical Pair Detection Algorithm

The algorithm for computing the set of all critical pairs for a pair of rules can be derived from the definition. First we need to compute all jointly surjective matches for the two rules. Then we need to analyse if the transformations given by the rules and matches are indeed parallel dependent. If this is the case then we have found a critical pair.

Algorithm 1 Simple algorithm for computing all critical pairs

Require: Two rules $p_1 = (L_1 \xrightarrow{r_1} R_1)$ and $p_2 = (L_2 \xrightarrow{r_2} R_2)$

1: $CP \leftarrow \varnothing$
2: Compute all jointly surjective matches $(m_1 : L_1 \to K,\ m_2 : L_2 \to K)$
3: for p_1 and p_2
4: **for all** (m_1, m_2) **do**
5:     Compute $t_1 = (K \overset{p_1,m_1}{\Longrightarrow} P_1)$ and $t_2 = (K \overset{p_2,m_2}{\Longrightarrow} P_2)$
6:     **if** t_1 and t_2 are parallel dependent **then**
7:         $CP \leftarrow CP \cup \{(t_1, t_2)\}$
8:     **end if**
9: **end for**

This algorithm has been implemented in the graph transformation tool

GROOVE[1] [35]. The implementation was written in Java, and reuses many of the existing GROOVE classes. The implementation for generating all critical pairs adds two classes to the existing GROOVE code. These two classes together consist of roughly 1000 lines of code.

7.2 Complexity of the Algorithm

The most complex part of the algorithm is the computation of the jointly surjective matches, since the number of these matches will grow very large if the left-hand sides for the two rule are large. In Proposition 7.2.1 we will show that it is sufficient to compute only the jointly surjective matches for the vertices of a pair of rules, since edges are defined uniquely by their source and target vertex. From this we can conclude that the number of jointly surjective matches is defined by the number of vertices in the two left-hand sides of the rules.

Proposition 7.2.1. *Given a pair of rules $p_1 = (L_1 \xrightarrow{r_1} R_1)$ and $p_2 = (L_2 \xrightarrow{r_2} R_2)$, a pair of jointly surjective and total set morphisms ($m_V : V_{L_1} \to V_G$ and $n_V : V_{L_2} \to V_G$), then for any jointly surjective and total $m : L_1 \to G$ and $n : L_2 \to G$, where $m = (m_V, m_E)$ and $n = (n_V, n_E)$ the graph G is unique up to isomorphism.*

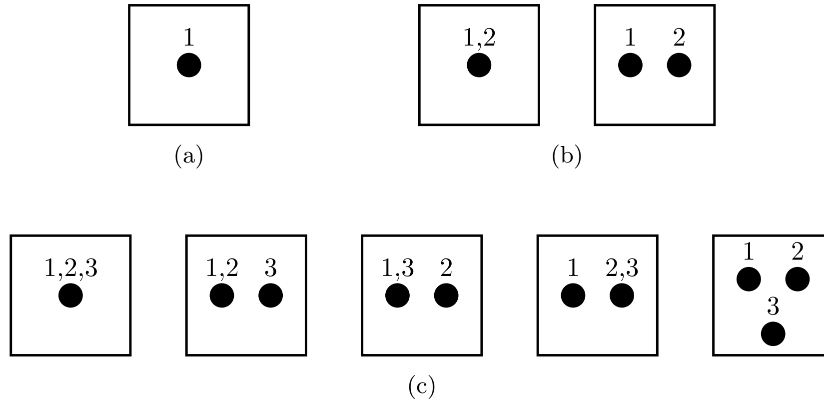
Proof. Because m and n are total, we know (by Proposition 6.2.4) that the edge components of m and n are uniquely defined by their vertex components. The morphisms m and n are jointly surjective therefore $G = (V_G, E_G)$ where $V_G = m_V(V_{L_1}) \cup n_V(V_{L_2})$ and $E_G = m_E(E_{L_1} \cup n_E(E_{L_2})$. Assume there exists total and jointly surjective morphisms $m' : L_1 \to G'$ and $n' : L_1 \to G'$ such that $m'_V = m_V$ and $n'_V = n_V$. Then we have $G' = (V'_G, E'_G)$ where $V'_G = m_V(V_{L_1}) \cup n_V(V_{L_2}) = V_G$ and $E'_G = m'_E(E_{L_1}) \cup n'_E(E_{L_2})$. Because the edge components m' and n' are uniquely defined by their vertex components, we know that $E'_G \cong E_G$ and therefore $G \cong G'$. □

Since the number of vertices in a (non-attributed) graph is finite, the number of jointly surjective matches is also finite. However, since we allow non-injective matches, the number of jointly surjective matches explodes quickly. In the next example, we will show how we can calculate the total number of (distinct) jointly surjective matches for two different rules with (added together) have n vertices.

Example 7.2.2. Consider a pair of rules with left-hand-sides $L_1 = (V_1, E_1)$ and $L_2 = (V_2, E_2)$, let V be the disjoint union of the sets of vertices V_1 and V_2. We will now show how to compute all surjective total morphisms $x : V \to X$. We call the set X with the surjective and total morphism x an overlapping of the vertices in V.

In case $|V| = 1$ then there is exactly one overlapping. For $V = \{1\}$ this overlapping is depicted in Figure 7.1a. In Figure 7.1b, we show overlappings for the set $V = \{1, 2\}$. Here we see that there are two possibilities: the two vertices can be two separate vertices, or they can be joined together.

[1]We have discovered that the pushout construction we have given is not correct, because pushouts do not always exist. GROOVE was built on the assumption that pushouts always exist, therefor the correctness of our implementation is no longer guaranteed (see Section 8.3 and Appendix A.2).



(a) (b) (c)

Figure 7.1

More generally we can compute the overlappings for any set V in the following way: let $v \in V$ and let $V' = V \setminus \{v\}$ (we have $|V'| = |V| - 1$). Compute all overlappings for V'. For every overlapping $(X', x' : V' \to X')$ we can create an overlapping $(X, x : V \to X)$ of the vertices of V in the following ways:

1. Take $X = X'$ and define x such that $x(v) \in X'$ and for all $v' \in V'$ we have $x(v') = x'(v')$ (there are $|X|$ ways to do this).
2. Take $X = X' \cup \{v\}$ and define x such that $x(v) = v$ and for all $v' \in V'$ we have $x(v') = x'(v')$.

In Figure 7.1c we see an example of all the overlappings of $V = \{1, 2, 3\}$. Let $v = 3$, then $V' = V \setminus \{3\} = \{1, 2\}$. We have already computed the two overlappings of V' in Figure 7.1b. The first overlapping in Figure 7.1b has only 1 vertex. We can add the vertex 3 in two ways: either we 'merge' it with the existing vertex (this gives us 1 vertex as result) or we add the vertex 3 separately. The results are the two leftmost overlappings in Figure 7.1c. The overlapping of V' has two vertices (this is the right overlapping in Figure 7.1b). From this overlapping we can create three new overlappings: we can 'merge' the vertex 3 with either one of the two existing vertices, or we can add the vertex 3 separately; this results in the three rightmost overlappings in Figure 7.1c.

Below we see how we can compute $v(n)$, which is the total number of possible overlappings for n vertices. To compute this number we take the sum for $i = 1 \ldots n$ of $v(n, i)$. With $v(n, i)$ we compute the number of ways to overlap n vertices, such that the overlapping has i vertices.

If $n < i$ then there are no overlappings (e.g., there is no way to overlap 4 vertices such that the overlapping has 5 vertices). If $n \geq i$ and $i = 1$ then there is exactly one overlapping (there is exactly one way to overlap $n > 0$ vertices, such that the overlapping has 1 vertex).

In any other case we can calculate the number of overlappings as follows. First we calculate the number of ways to overlap $n - 1$ vertices such that the overlapping has i vertices ($v(n - 1, i)$) for each of these overlappings there are i ways to create a new overlap (see 1. in Example 7.2.2). For every overlapping of $n - 1$ vertices with size $i = 1$ ($v(n - 1, i - 1)$); there is one way to create a

new overlapping of n vertices with size i (see 2. in Example 7.2.2).

$$v(n) = \sum_{n}^{i=1} v(n,i)$$

$$v(n,i) = \begin{cases} 0 & \text{if } n < i \text{ or} \\ 1 & \text{if } i = 1 \text{ and } i \leq n \\ v(n-1,i)i + v(n-1,i-1) & \text{otherwise} \end{cases}$$

If $n \geq 1$ then $v(n)$ is equal to the nth Bell number [5]. For these Bell numbers, the following upper bound is known [6]:

$$v(n) = B_n < \left(\frac{0.792n}{\ln(n+1)}\right)^n, \quad n \in \mathbb{N}_{>0}$$

It is clear that the total number of jointly surjective matches can be extremely large if the two left-hand-sides for which we are finding the matches are large. For future work we could write a more efficient algorithm which only computes those jointly surjective matches which actually forms a critical pair (i.e., there must be an overlap of a vertex or edge that in being deleted by one of the rules). However the worst-case complexity of such an algorithm would still be the same as this algorithm.

7.3 Confluence Analysis

Using our implementation of the critical pair detection algorithm, we can find the set of all critical pairs for each graph transformation system. Then we can use the tool GROOVE to analyse if these critical pairs are strictly locally confluent. This confluence analysis uses many existing features in GROOVE:

- We analyse which rules are applicable to a certain state (a state is a graph which is the target of the last transformation, together with the transformation morphism of the transformation).
- For every state we can apply all possible rules via all possible matches to find the next state. In this process it is important that we also keep track of what the first transformation for a state was (i.e., from which of the two transformations of this critical pair the state originates).
- If we have found two isomorphic states with different origins, then we check if the transformation morphisms commute, if this is the case then we have found proof the critical pair was strictly locally confluent.

It is also possible that after finding all possible states, no proof for strict local confluence was found. This means that the critical pair is not strictly locally confluent.

As we have mentioned before, local confluence is undecidable. This is because the state space in which we need to search can be infinitely large. To solve this, we have limited our search depth to 100 states. If the search reaches this depth, then the result will be undecided.

7.4 Graph Transformation Systems in Practice

We will give several examples of graph transformation systems which have been created in GROOVE. From some of these we know that they are not confluent, for others it is expected that they are (locally) confluent.

For every graph transformation system we will discuss, we will analyse if there exist critical pairs which are not strictly locally confluent, or critical pairs for which we cannot decide strict local confluence (since we limit our search depth).

Attribution, Algebras and Operations For non-attributed graphs, we know that the set of vertices is always finite (in practice), therefore the set of jointly surjective matches will also be finite. For attributed graphs the set of vertices may not be finite, however we compute critical pairs with the same algebra as the rule. This means the algebra morphism is a monomorphism. Because of this, two constants must always overlap if they are equal. When computing the jointly surjective matches, the match for the constants is already defined by the algebra isomorphism, the vertices for which we still need to compute the match (i.e., the vertices which are not values) is finite.

In GROOVE, rules use the term algebra, which is an algebra where values can be variables, in this case, it is also possible that variables overlap with other variables or constants (however two different constants can never overlap). In our critical pair algorithm we compute all possible overlappings, this means that if the GTS is strictly locally confluent for the term algebra, then it will also be strictly locally confluent if any other algebra is used (the converse does not hold).

Some of the rules that we will discuss use operations, since the result of the operation is defined by the term algebra, we cannot overlap this result with any other values. Because the rules which we discuss have no edges to these values in the LHS, and algebra elements cannot be deleted, these vertices cannot be the cause of parallel independence, therefore it is justified that we do not overlap the targets of operations which other values.

Throughout this section we will show some figures which depict rules in the graph transformation systems. For these rules we use the following notational conventions:

- Vertices or edges which are deleted (elements of the LHS, which have no image in the RHS) are depicted as thin, dashed (blue) vertices and edges.
- Vertices or edges which are added by the rule (elements of the RHS, which have no preimage in the LHS) are depicted as wide (green) vertices or edges.
- All other vertices and edges are both in the LHS and RHS.
- Every line of text on a vertex is a self edge (of which the line is the label). If such a self edge is deleted by the rule is prefixed with a '-', if it is being added by the rule then it is prefixed with a '+'.

- If there are multiple edges with the same source, target and effect (e.g., all edges are deleted), but different labels, then we will depict only one edge and show the edge labels on different lines.

7.4.1 Dining Philosophers

Figure 7.2 shows five rules for a graph transformation system that models the Dining Philosophers problem. In this GTS a philosopher can become hungry, at this point he must take the forks on his left and right side (one by one), before he can start eating, afterwards he will put the fork down again so they are available for other philosophers. It is possible that every philosopher takes the left fork, such that the right fork is already taken for ever philosopher, in this state, no rules can be applied any more.

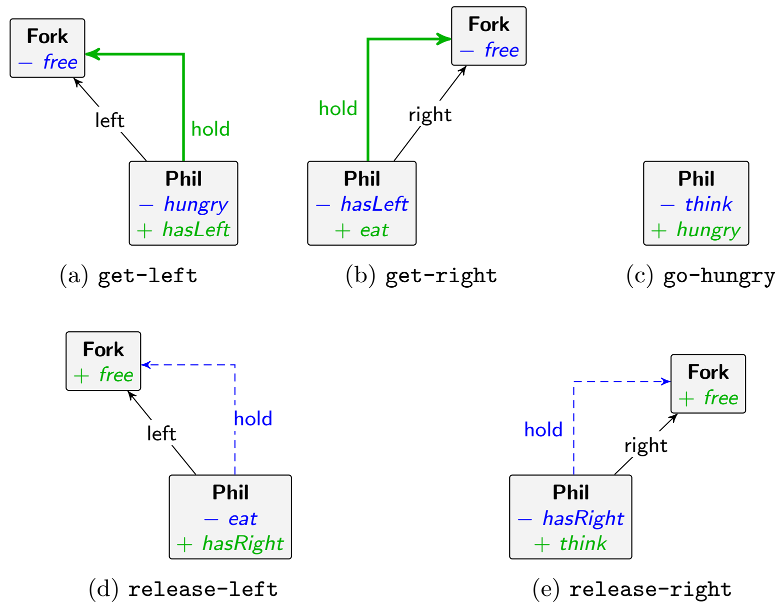


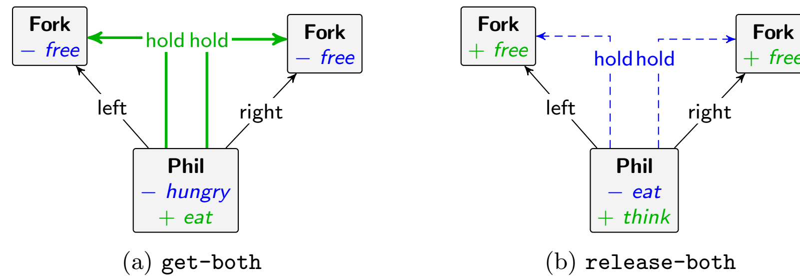
Figure 7.2: Rules for the Dining Philosophers Graph Transformation System

Expected outcome We expect that this graph transformation is locally confluent. As far as we are aware, the only state in which no rules can be applied any more is the state in which every philosopher hold exactly one fork (we call this the deadlock state). From any other state, it should be possible to reach the deadlock state, since every philosopher which has two forks can release these forks and take the left one. Philosophers which were waiting for the left fork to become available (because it was not free) should be able to pick up their left fork at this point. Once all philosophers have picked up their left fork, they will arrive at the deadlock state.

In conclusion, since there is exactly one final state and all other states should be able to reach this state, the graph transformation system is expected to be confluent.

7.4.2 Dining Philosophers (2)

Since the Dining Philosophers GTS we have just specified has a deadlock state, we have created a different graph transformation system in which the rules `get-left` and `get-right` have been replaced by `get-both`. Similarly, the rules `release-left` and `release-right` have been replaced by `release-both`. We have depicted these new rules in Figure 7.3. This GTS also has the rule `go-hungry` which we have depicted in Figure 7.2c.



(a) `get-both` (b) `release-both`

Figure 7.3: Different rules for the second Dining Philosophers Graph Transformation System

Expected outcome Similarly to the previous GTS that models the Dining Philosophers problem, we expect that this GTS is locally confluent. The main difference with the previous GTS is that it is no longer possible that all philosophers are waiting to get more forks, since they always pick up both forks at once.

7.4.3 Counting Dining Philosophers

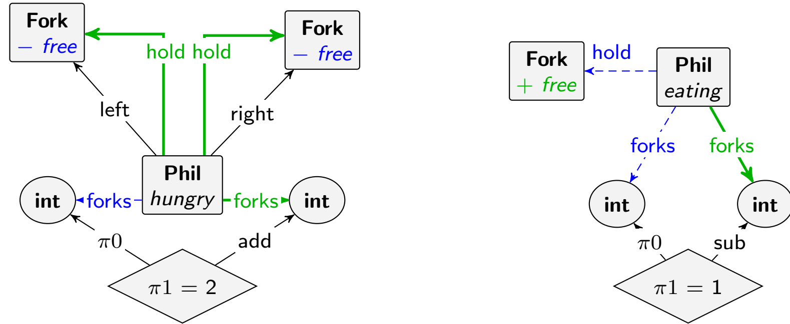
The counting dining philosophers GTS is also based on the dining philosophers problem, the main difference with the previous two graph transformation systems is that this GTS has attributed graphs. The philosophers count how many forks they have, if they have two forks, then they can start eating. If they have zero forks then they can go and think.

Expected outcome This GTS has a rule `get-both` which allows the philosopher to pick up two free forks. Because if this, the deadlock state which we have discussed in the first dining philosophers GTS cannot happen here. Therefore we expect that this graph transformation system is confluent.

7.4.4 Circular Buffer

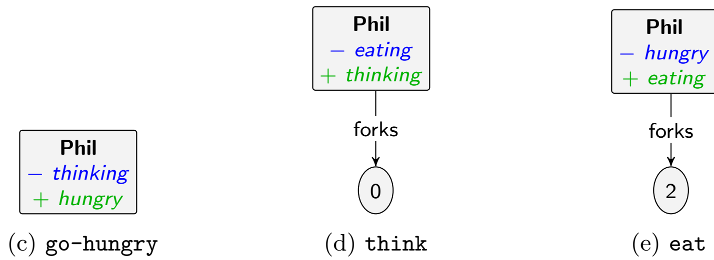
In Figure 7.5 we see a GTS which models a circular buffer. The rules `get` and `put` allow objects to be removed and added to the buffer, respectively. The rule `extend` increases the size of the buffer.

Expected outcome From every state of the circular buffer, it should be possible to apply the rule `put` a number of times to make sure that the buffer is completely full. The rule `extend` should be applicable when the buffer is full,



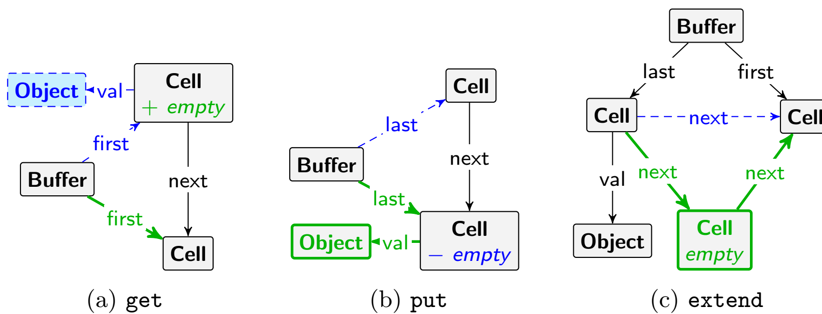
(a) **get-both**, the diamond vertex represents the integer addition operation, where π_0 and π_1 are the arguments.

(b) **release**, the diamond vertex represents the integer subtraction operation, where π_0 and π_1 are the arguments.



(c) go-hungry (d) **think** (e) **eat**

Figure 7.4: Rules for the Counting Dining Philosophers Graph Transformation System



(a) get (b) put (c) **extend**

Figure 7.5: The rules for the Circular Buffer GTS

repeated application of `put` and `extend` allows the buffer to grow to any size. We expect that this GTS is confluent. However, since it may be possible that the rules `put` and `extend` can be applied infinitely often (alternating). Therefore it may be possible that we are not able to decide local confluence for some critical pairs.

7.4.5 Counting Circular Buffer

The counting circular buffer GTS is similar to the circular buffer GTS, however this circular buffer does not use `empty` self edges. Instead this buffer counts how many cells are free. This GTS uses attributed graphs.

The rules are shown in Figure 7.12. We should note that the rule `put` contained a check if the buffer was not empty (using the 'greater than' operation). However since the result of operations in the term algebra are not constants (such as `true` or `false`), but variables, the rule is never applicable in the term algebra. Therefore we have removed this check from the rule.

(a) `get`

(b) `put`

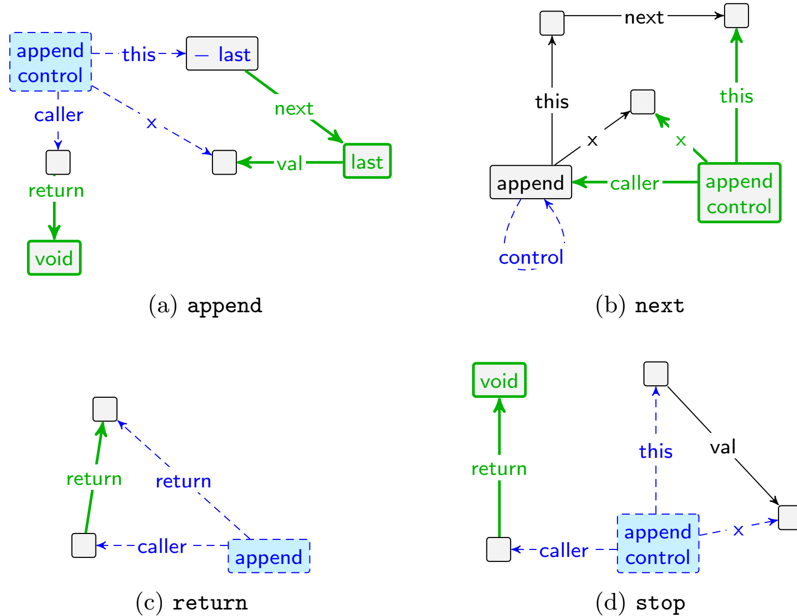
(c) `extend`

Figure 7.6: The rules for the Counting Circular Buffer GTS

Expected outcome Similarly to the regular circular buffer it is possible that the rule sequence `put` and `extend` is applied infinitely often. Therefore it may be possible that confluence cannot always be decided. We expect that this GTS is locally confluent.

7.4.6 Append

The Append GTS (Figure 7.7), models appending an element at the end of a linked list. In order to do so, the last element of the list must be found by iterating over the list.



(a) append (b) next

(c) return (d) stop

Figure 7.7: Rules for the Append GTS

Expected outcome In case two elements a and b are being appended at the same time, then two outcomes (final states) are possible: one where a is the last element of the linked list, and one where b is the last element of the linked list. Therefore this GTS is not confluent.

7.5 Results

In this section we present the results for the critical pair and confluence analysis. The table below shows the number of critical pairs we have found for each GTS, it also shows how many of these pairs are not locally confluent, and for how many of these pairs we could not decide local confluence.

GTS	Number of critical pairs			Time (s)	Strictly confluent
	Total	Non conf.	Undecided		
Dining Phil	25	25	0	0.1	No
Dining Phil (2)	191	0	0	0.2	Yes
Counting DPhil	318	276	20	0.4	No
Circular Buffer	6689	3683	2634	20.5	No
Counting CBuffer	14788	7257	6099	128.1	No
Append	17787	16023	1388	33.9	No

Essential Critical Pairs In Section 6.1 we concluded that the conditions on essential critical pairs were very strict: we conjectured that the set of essential critical pairs would not be significantly smaller than the set of all critical pairs. We have performed analysis to find out which critical pairs were essential and which were not. It turns out that in all six graph transformation systems every critical pair is an essential critical pair. This is the case because none of the rules (in any of the graph transformation systems) deletes more than one vertex.

Subsumption of Critical Pairs In Section 6.2 we have described when a strictly locally confluent critical pair a can represent another critical pair b, which means that the strict local confluence of b is implied by the strict local confluence of a. In the next table compare the time it takes to analyse strict local confluence using our original method (as explained in Section 7.3), and the new method, which first analyses if the strict local confluence of a critical pair can be implied by another critical pair.

The table also shows the total number of critical pairs, the number of strictly locally confluent pairs and the number of subsumed critical pairs (i.e., the number of critical pairs for which the strict local confluence was implied by another critical pair).

	Time (s)		Critical Pairs		
GTS	Original	Subsumption	Total	Confluent	Subsumed
Dining Phil	0.1	0.1	25	0	0
Dining Phil (2)	0.2	0.2	191	191	154
Counting DPhil	0.4	0.4	318	22	14
Circular Buffer	20.5	20.4	6689	372	313
Counting CBuffer	128.1	121.9	14788	1432	1355
Append	33.9	34.4	17787	376	277

We can see that the time it takes for both methods to analyse all critical pairs does not differ very much. For the first three graph transformation systems both methods analyse the GTS very quickly, the speed difference was not measurable. We do see that our subsumption approach is more efficient for the Counting Circular Buffer GTS, however the difference is small (roughly 5%).

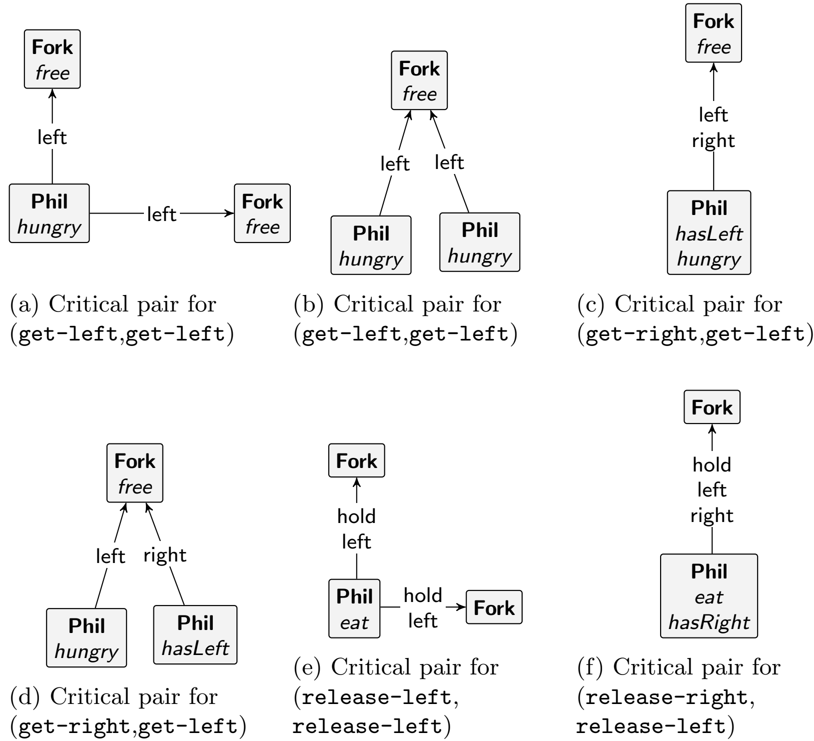
We can see that the last three graph transformation systems have many critical pairs, however only a few of these are strictly locally confluent. Our subsumption method falls back to the original way for confluence analysis if strict local confluence can not be determined using subsumption. Therefore, for most of these critical pairs, the old method was used to determine whether they were strictly locally confluent.

In the next sections, will show some of the critical pairs for every GTS (which are not strictly confluent, or for which we could not decide strict local confluence). In order to depict these pairs, we only show the names of the rules and target graph of the two jointly surjective matches.

7.5.1 Dining Philosophers

Analysis of the Dining Philosophers graph transformation system shows us that there are 25 critical pairs. For all of these our implementation was able to decide

that they were not strictly locally confluent. In some of these critical pairs, (at least) one of the vertices represents both a philosopher and a fork. If we exclude these critical pairs, there are nine critical pairs remaining. We have shown six of these pairs in Figure 7.8; for the critical pairs Figures 7.8a, 7.8b and 7.8e the pair for getting/releasing the right forks (instead of left) is analogous.



(a) Critical pair for (`get-left`,`get-left`)

(b) Critical pair for (`get-left`,`get-left`)

(c) Critical pair for (`get-right`,`get-left`)

(d) Critical pair for (`get-right`,`get-left`)

(e) Critical pair for (`release-left`, `release-left`)

(f) Critical pair for (`release-right`, `release-left`)

Figure 7.8: Critical pairs for the Dining Philosophers Graph Transformation System, none of these are strictly locally confluent.

We see that all of these pairs are situations which we do not want in reality. Ideally we would require that ever philosopher has at most one left fork, and at most one right fork. We also should not allow the left fork to be equal to the right fork. We see that only the critical pair in Figure 7.8d satisfies these constraints.

We could also require that every philosopher has at least one left and right fork. This can be done by modelling this constraint in every rule. The rules `get-left`, `release-left`, `get-right` and `release-right` should be modified such every philosopher has a fork on his left, and a fork on his right.

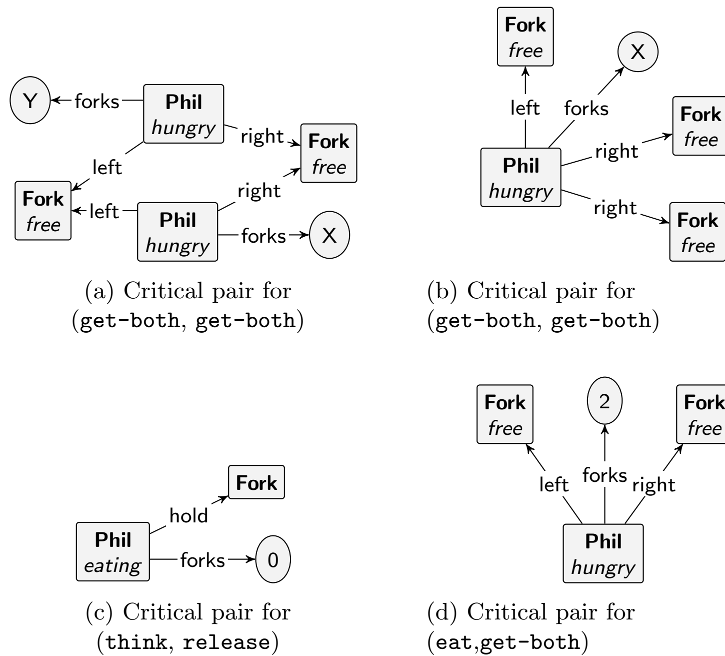
7.5.2 Dining Philosophers (2)

Our results were as expected, there are more critical pairs because the rules `get-both` and `release-both` are bigger than the rules which they have replaced. It turns out however, that every critical pair is strictly locally confluent. Therefore we can conclude that this GTS is strictly locally confluent.

7.5.3 Counting Dining Philosophers

Analysis has shown that the counting dining philosophers GTS is not strictly locally confluent. There were many critical pairs which were not confluent and also some pairs for which confluence could not be decided. For these 'undecided' pairs, it was the case that an infinite sequence of `get-both`, `release`, `release` could be applied. Because we compare the algebra attributes in the term algebra the attributes do not become equal after applying operations. For example the expression $2 - 1 - 1$ is not equal to 0 in the term algebra.

Many of these critical pairs are situations which we do not want in practice. These critical pairs show philosophers with multiple `forks` edges, a philosopher where the left and right fork are the same, or a philosopher which is both *eating* and *thinking* (or *hungry*). If we exclude such situations we are left with 15 critical pairs which are not confluent, and zero critical pairs for which confluence cannot be decided. Some of these pairs are shown in Figure 7.9.



(a) Critical pair for (`get-both`, `get-both`)

(b) Critical pair for (`get-both`, `get-both`)

(c) Critical pair for (`think`, `release`)

(d) Critical pair for (`eat`,`get-both`)

Figure 7.9: Critical pairs for the Counting Dining Philosophers Graph Transformation System, none of these are strictly locally confluent. The circular nodes with X or Y represent variables.

The situation in Figures 7.9a and 7.9b is not strictly locally confluent because after applying either of the possible applications for the rule `get-both`, it is not possible to apply any rules any more, because a philosopher needs to have two forks before the rule `eat` can be applied. In the term algebra, an operation can never result in a constant. Note that these critical pairs are locally confluent (but not strictly), because both possible applications for `get-both` give an isomorphic result.

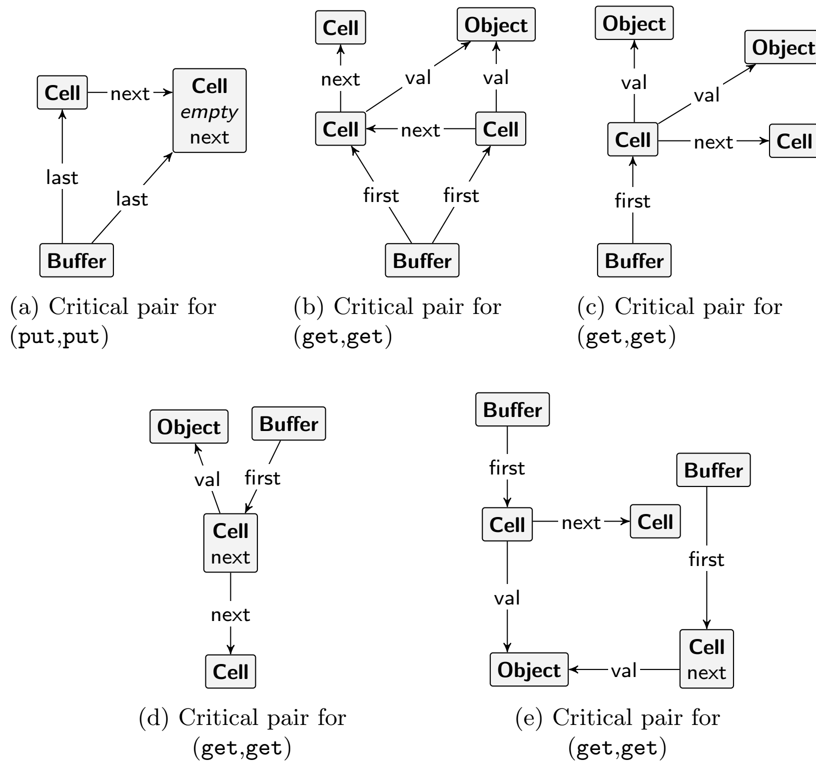
In Figure 7.9c, we see that the philosopher does not have a left and right fork specified. However the philosopher does hold a fork, and the number of

forks is zero. It is a situation which should not happen in practice.

In Figure 7.9d, we see a philosopher where the number of forks is two, but the philosopher does not hold any forks. This is also not a situation which we want in practice.

7.5.4 Circular Buffer

We expected the circular buffer to be confluent, however there are many critical pairs which are not confluent, some of these are shown in Figure 7.10. Figure 7.11 shows two of the pairs for which confluence could not be decided.



(a) Critical pair for (put,put)

(b) Critical pair for (get,get)

(c) Critical pair for (get,get)

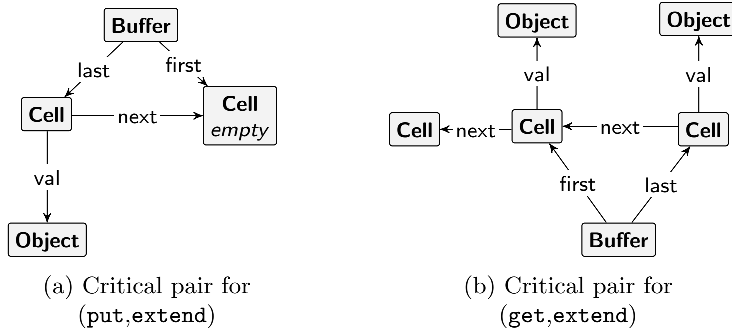
(d) Critical pair for (get,get)

(e) Critical pair for (get,get)

Figure 7.10: Critical pairs for the Circular Buffer GTS which are not strictly locally confluent.

Many of these critical pairs represent situations which are not desired in practice. In some of these critical pairs we see that there is more than one `Buffer`. Other pairs have `Cells` with more than one outgoing `next` or `val` edge. There are critical pairs in which the `Buffer` has more than one `first` (or `last`) edge. We also have some critical pairs in which there exists an `Object` with more than one incoming `val` edge.

If we exclude all critical pairs for which the above constraints do not hold, and for which the type constraints hold (i.e., every vertex is either a `Buffer`, a `Cell` or an `Object` but not multiple of these at the same time), then there are no pairs remaining which are not confluent, but there are two remaining pairs for which confluence could not be decided. These are depicted in Figure 7.11.



(a) Critical pair for (`put`,`extend`)

(b) Critical pair for (`get`,`extend`)

Figure 7.11: Critical pairs for the Circular Buffer GTS for which strict local confluence could not be decided.

Consider the critical pair in Figure 7.11a. The rules `extend` and `put` can be applied infinitely often (alternating). This pair is also a situation which is not desired in practice, since the buffer in this critical pair is not circular. For similar reasons, we cannot decide strict local confluence for the critical pair in Figure 7.11b. An interesting observation here is that this pair is actually locally confluent (but not strictly locally confluent), we can check this by applying the rule sequence `get`, `put`, `extend` to the graph in Figure 7.11b, if we also apply the sequence `extend`, `get`, `put`, then the results are isomorphic (but the transformation morphisms do not commute).

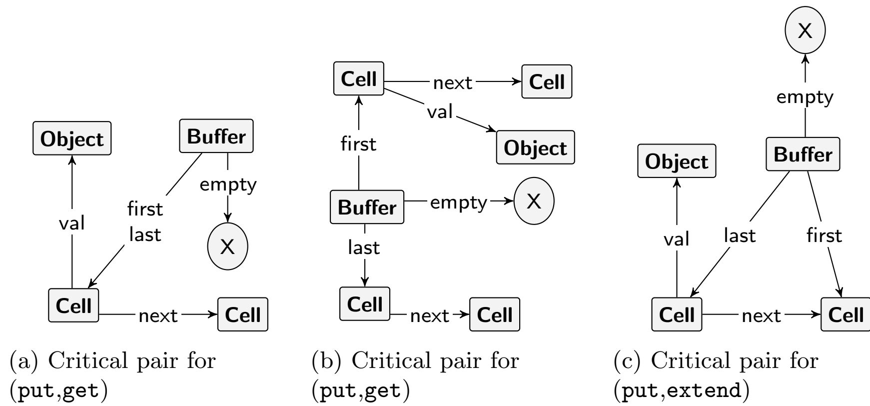
7.5.5 Counting Circular Buffer

Analysis has shown that this GTS was not confluent. Similarly to the previous GTS, many of these situations are situations which we do not desire in practice. We have excluded all cases where there are multiple `Buffer`s, where the `Buffer` has multiple `first`, `last` or `empty` edges, situations where cells have more than one incoming or outgoing `next` or `val` edge, and situations where an `Object` has more than one incoming `val` edge.

After this exclusion we are left with two pairs which are not confluent (shown in Figures 7.12a and 7.12b), and one critical pair for which confluence could not be decided (shown in Figure 7.12c).

For the critical pairs shown in Figures 7.12a and 7.12b, our analysis shows that the critical pair is not locally confluent. This is because of the term algebra: if we would apply the sequence of rules `get`, `put`. Then our the `empty` edge would point to $X + 1 - 1$, however if we apply the sequence of rules `put`, `get`, then the `empty` edge would point to $X - 1 + 1$, in the term algebra these expressions are not equal. However in other algebras these expressions can be equal to each other, if this would be the case then these pairs would be strictly locally confluent.

Confluence for the critical pair in Figure 7.12c could not be decided, since the sequence of transformations `extend`, `put` can be applied infinitely often. If we apply the rule `put` to the graph in Figure 7.12c, then no rule applications are possible any more. We see that the buffer in this situation is not actually circular, if this would have been the case, then the pair could still be (strictly) locally confluent.



(a) Critical pair for (`put`,`get`)

(b) Critical pair for (`put`,`get`)

(c) Critical pair for (`put`,`extend`)

Figure 7.12: Critical pairs for the Counting Circular Buffer GTS, the circular vertices with an X are variables.

7.5.6 Append

Our expectation was that this GTS was not locally confluent, our analysis confirms that we were correct. In Figure 7.13 we see one of the critical pairs. In contrast to the graph transformation systems we have seen before, this critical pair shows a situation which can happen in practice.

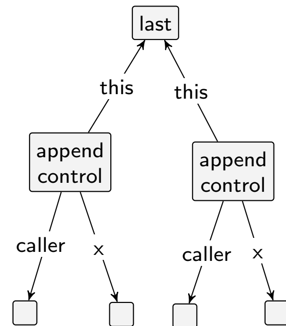


Figure 7.13: Critical pair for (`append`,`append`)

There are exactly two different ways to apply the rule `append` to this graph. Both applications give an isomorphic result, therefore the critical pair is locally confluent. However the transformation morphisms for these two rule applications do not commute, therefore critical pair is not strictly locally confluent.

This is a typical example of why strict local confluence is required instead of local confluence, if the two vertices which are targets of the `x` edge could be distinguished (for instance if they would have a self-edge), then the transformations would no longer be confluent. This case is similar to the example we have given in Figure 3.6 on page 33.

7.6 General Conclusion

For most of these graph transformation systems we see that the critical pairs which are not strictly confluent are situations which we do not desire in practice.

In order to solve this problem we could define constraints for each graph transformation system which must always hold. An example of such a constraint would be: every philosopher has exactly one left fork and exactly one right fork, the left and right forks may not be the same fork. We can only model a constraint like this if we can use negative application conditions; for example, using negative application conditions we can ensure that the left and right fork are not the same. Unfortunately we can not yet detect and analyse critical pairs with NACs at this point.

In the circular buffer, we have seen that the critical pairs show buffers which are not circular. In GROOVE it is possible to model an application condition like this using a regular expression; however, we are not able to analyse such a rule at this point.

Using rules with constraints, critical pair analysis should no longer give counterexamples which cannot happen in practice (provided that sufficient constraints have been modelled into the rules).

In the circular buffer GTS, we have found that the alternation of the rules `extend` and `put` was applicable infinitely often in some cases, this is because there exists a match from the LHS of this rule to its own RHS. Because of this, we reached our maximum search depth in some cases, and the confluence search was terminated. For the pairs where this was the case, our algorithm could not decide if the pairs were confluent.

For one of the systems which we have tested (Append), we have expected that the GTS was not locally confluent because we could think of a conflicting situation. In our analysis if this system, we have managed to find a critical pair which represents this conflict. We know that, for every GTS which is not locally confluent, there must exist a parallel dependent pair of transformations. We have also proven that for such a parallel dependent pair of transformations, there exists a critical pair. Since we find and analyse all critical pairs, this means that our analysis will never give false positives, i.e., if the GTS is not locally confluent, then the result of our analysis will never be that the GTS is locally confluent.

The converse of this may not be true, it can be the case that a locally confluent GTS is not *strictly* locally confluent. Furthermore, it can be possible that we cannot find evidence that a critical pair is strictly locally confluent (we only search a limited depth of states to prevent searching infinitely), in this case local confluence cannot be decided.

Two of the transformation systems we have analysed made use of our theory on attributed graphs. For these GTSs, we have only analysed critical pairs in the term algebra. For some of these critical pairs we have seen that they were not confluent, because some expressions were not equal in the term algebra. It would be nice if we could also analyse these expressions using different algebras. This would allow us to show local confluence for other algebras as well.

Chapter 8

Conclusion

In the following, we first summarize the achievements of this thesis; we then return to the research questions formulated in Section 1.6 to see how far we got in answering these. Finally we discuss possible directions for future research.

8.1 Achievements

We have presented theory on confluence analysis for SPO high-level replacement. We have generalized some existing theorems (embedding theorem and local confluence theorem) on SPO graph transformation to high-level replacement. In order to do this we have defined (strict) SPO categories, which specify requirements which must hold for the categories for which our proofs are valid. We have defined critical pairs and shown that there exists a critical pair for every conflict (i.e., every pair of parallel independent transformations).

Our theory is not only applicable to graphs with partial graph morphisms: we have also shown that our work is also applicable to attributed graphs. Both of these are supported by the graph transformation tool GROOVE. These are not the only categories for which our theory is applicable: we conjecture that typed (attributed) graphs and hypergraphs are also strict SPO categories.

We have also presented some of the foundations for confluence analysis for HLR systems with NACs. By defining NAC SPO categories, we have defined for which categories our theorems are valid. Using these NAC SPO categories, we have shown an embedding theorem which allows NACs, we have defined conditional critical pairs (critical pairs with NACs), and we have shown that there exists a conditional critical pair for every conflict. Unfortunately we found that the strict local confluence of all conditional critical pairs does not imply strict local confluence of an HLR system, and we have also not yet shown that graphs with partial graph morphisms form a NAC SPO category. This is future work.

In search for a more efficient method for critical pair detection, we researched whether there exist essential critical pairs, which are a subset of critical pairs such that all critical pairs are strictly locally confluent if and only if all essential critical pairs are strictly locally confluent. It turns out that the restrictions on essential critical pairs are very strict, which has as a consequence that the set of

essential critical pairs is not significantly smaller than the set of critical pairs.

We have solved this issue by presenting an alternative method for confluence analysis: we have defined when a strictly locally confluent critical pair can represent another critical pair, which means that the other critical pair is also strictly locally confluent. Because we know that the first pair is strictly locally confluent, the restrictions on this sufficient condition are less strict than the restrictions on essential critical pairs.

8.2 Evaluation

In our introduction (Section 1.6) we have mentioned three main steps: first, we wanted to research whether the existing theory on DPO and SPO graph transformation could be modified in order to be applicable in our situation; secondly, we wanted to implement an algorithm for critical pair detection in GROOVE; and lastly, we wanted to use GROOVE to analyse whether some graph transformation systems were (strictly) locally confluent, to see how well we could determine local confluence in practice. We will evaluate these three steps separately.

8.2.1 Modifying existing theory

In order to show local confluence of a HLR system, we needed to prove several main theorems. The embedding theorem (Theorem 3.1.12), the completeness theorem for critical pairs (Theorem 3.2.5), and the local confluence theorem (Theorem 3.3.4). Similar theorems have also been proven to show local confluence of DPO HLR systems [10].

A basic version of the embedding theorem was already shown by Ehrig et al. [13]. This embedding theorem by Ehrig is only applicable to graphs, however the definition of graphs by Ehrig is slightly different from ours (edges are unlabelled and pairs of vertices could be connected by multiple edges). Furthermore the embedding theorem by Ehrig was restricted to injective embeddings (i.e., these embeddings are families of monomorphisms).

We have generalized this embedding theorem so that embeddings are not necessarily monomorphisms, and our embedding theorem is not only applicable to the category of graphs with partial morphisms, but to any strict SPO category. We have shown that graphs and attributed graphs form strict SPO categories[1].

Our definition for critical pairs is based on (and very similar to) the existing definition for DPO HLR systems in [10]. We were able to show that there exist a critical pair for every conflict which has not been previously shown in work on SPO graph transformation or high-level replacement.

In order to provide a sufficient condition for local confluence of an HLR system, we needed to require strict local confluence, which was also the case in work on both SPO graph transformation [27] and DPO high-level replacement [10]. Using the notion of strict local confluence, we were able to prove a sufficient condition for local confluence of an SPO HLR system.

[1] Unfortunately, we have discovered that this is not true, see Section 8.3 and Appendix A.2.

We continued our work in order to find a sufficient condition for local confluence of an HLR system with NACs. In order to do so, we had to require that right NACs could be converted to (equivalent) left NACs. We came up with the Pushout/Pullback property which ensures that right NACs can indeed be converted to left NACs. However, we have not yet proven that this property holds for $\boldsymbol{Graph^P}$ (this is future work). Using this property, we have defined and proven an embedding theorem which allows NACs.

The notions of parallel independence and critical pairs (now called conditional critical pairs) needed to be redefined to allow NACs, the definitions we have given were again similar to the definitions for DPO [24]. Using the Pushout/Pullback property we were able to show that conditional critical pairs were complete. This allowed us to conclude that a SPO HLR system with NACs is locally confluent if there are no critical pairs. Unfortunately we found that strict local confluence of all conditional critical pairs does not imply the (strict) local confluence of the HLR system. It is future work to find a sufficient condition for local confluence of an HLR which does have conditional critical pairs.

8.2.2 Algorithm for Critical Pair Detection

Using the theory we have developed, we have implemented an algorithm to find all critical pairs for a graph transformation system in GROOVE. For every pair of rules, the algorithm has to compute all possible ways to combine the vertices in the left-hand sides. Every combination forms a pair of jointly surjective matches. Unfortunately, the total number of these jointly surjective matches can grow very quickly. Therefore in the future, the algorithm could be improved so that it will only compute those jointly surjective matches which will form a critical pair (i.e., the parallel independent pairs of transformations are not computed). This improvement will not reduce the worst case complexity.

We have tested our algorithm by recreating some test cases from the AGG testsuite. Because AGG has no test cases for SPO graph transformation we have added some SPO specific testcases ourselves. We have also computed sets of critical pairs for some graph transformation systems, which haven then been used analyse whether these GTSs are (strictly) locally confluent.

8.2.3 Determining local confluence

Using existing features of GROOVE, we were able to analyse whether critical pairs are strictly locally confluent. Our case study has shown us that that this confluence analysis works very well: it finds many counterexamples if a GTS is not strictly locally confluent. We have also seen that many of these counterexamples are situations which cannot happen in practice, because some implicit constraints have not been satisfied. Such constraints can usually be formulated as positive or negative application conditions. The positive application conditions can be added to the left-hand sides of all rules to ensure that the rules only match if the constraints are satisfied. For negative application conditions, it is also possible to model these in the rules in GROOVE, however, critical pair and confluence analysis for rules with NACs is not yet possible.

8.3 A Last Minute Result

Unfortunately, just before finishing this thesis we have found a counterexample which shows that our pushout construction is incorrect (see Appendix A.2). This counterexample shows a situation where the pushout over two total $\boldsymbol{Graph^P}$-morphisms does not exist. As a consequence, this means that $(\boldsymbol{Graph^P}, \mathcal{M}, \mathcal{R})$ is not an SPO category because pushouts over $\mathcal{M}$ and $\mathcal{R}$ morphisms may fail to exist.

It turns out that the behaviour of the graph transformation tool GROOVE is the same as we have specified in our (incorrect) pushout construction. This means that GROOVE is able execute the incorrect transformation we have shown in Appendix A.2; however, the result of this transformation is not a pushout. This means that in particular a pushout construction for graphs with partial morphisms has to be reinvestigated. Unfortunately, the time constraints of this thesis made it impossible to carry this though. Instead, we have marked the affected results with footnotes.

A pushout construction for multi-sorted graphs (graphs which allow multiple edges between a pair of vertices) where edges are unlabelled has been shown by Ehrig et al. in [13]. It is likely that the definition for graphs and partial graph morphisms as given by Ehrig et al. can be used to form a strict SPO category for the transformation of graphs, in which therefore our main results can be applied.

8.4 Future work

Based on the work we have presented in this paper, there are several directions for future research:

1. Since we have defined HLR systems with NACs, we have presented and NAC SPO categories. Unfortunately we do not yet know whether graphs with partial morphisms form a NAC SPO category. Furthermore in order to be able to compute critical pairs a construction for pushout complements is needed.

2. The theory for critical pair and confluence analysis for HLR systems with NACs is not yet completely worked out. For example we believe that strict NAC-confluence of all conditional critical pairs (as we have discussed at the end of Chapter 4) is a sufficient condition for strict local confluence of a HLR system with NACs. However, it is not yet possible to decide when a conditional critical pairs is strictly NAC-confluent.

   Once all theory on HLR systems with NACs has been developed, an algorithm could be implemented in GROOVE, and our case study could be repeated. Using NACs it is possible to model constraints into the rules, such that rules are only applicable if these constraints hold. In our case study we have seen that many GTSs were not strictly locally confluent because some implicit constraints were assumed. These implicit constraints did not hold in the critical pairs we have found. In the new case study we will then see if local confluence can be decided for the graph transformation systems with constraints.

3. It is possible to model typed graphs with inheritance in GROOVE, this allows modelling of object-oriented systems. We do not yet know if a strict SPO category exists for typed graphs with inheritance.

4. The algorithm we have implemented computes all jointly surjective matches (where the host graph is as small as possible) for every pair of rules. Many of these jointly surjective matches do not lead to a critical pair. The performance of the algorithm could be improved if it would only compute those jointly surjective matches such that the transformations are parallel dependent.

5. GROOVE has support for many types of rules, such as nested rules (also called rules with nested application conditions). Since ideally we would want GROOVE to be able to analyse all kinds of graph transformation systems, it would be very useful if critical pair and confluence analysis for these rules can be performed using GROOVE.

6. Attributed graphs in the rules in GROOVE use the term algebra. This allows rules to have variables and apply operations. Unfortunately it can be the case (which we have seen in our case study) that a critical pair using the term algebra is not (strictly) locally confluent, however the same critical pair is strictly locally confluent when a different algebra is used. At this point our critical pair detection algorithm uses the same algebra as the rule graphs. It would be better if it would be possible to decide if a HLR system is locally confluent if a certain (non-term) algebra is used.

7. Since GROOVE is also a model checking tool which can efficiently compute state spaces, it could be investigated how *confluence reduction* [39] could be applied to reduce the number of states that need to be evaluated. The work on confluence reduction uses a slightly different notion of confluence, however we believe that our definitions for parallel independence and critical pairs could be useful to implement confluence reduction.

Acknowledgements

I would like to thank Arend Rensink, for his excellent feedback and supervision. I also want to thank Leen Lambers, for her advice and all the insights she has given me on this subject. Finally I would like to thank my other supervisors Peter van Rossum, Jan Kuper and Mariëlle Stoelinga, for all their comments and advise.

Appendix A

Proofs

A.1 Pushout Construction in $\boldsymbol{Graph^P}$

In this section we will restate our pushout construction[1] and prove several propositions which follow from our pushout construction.

Construction 2.3.4 (Pushout in $\boldsymbol{Graph^P}$)**.** Let A, B, and C be graphs, let $f : A \to B$, and $g : A \to C$ be (partial) morphisms. We can construct the pushout $B \xrightarrow{g'} D \xleftarrow{f'} C$ as follows:

Define the relation $\sim$ on the disjoint union $U = V_B \mathbin{\dot\cup} V_C \mathbin{\dot\cup} E_B \mathbin{\dot\cup} E_C$ as follows: for all $a \in (V_A \mathbin{\dot\cup} E_A)$ we have $f(a) \sim g(a)$ if $f(a) \neq \bot$ and $g(a) \neq \bot$.

Let $[x] = \{y \in U \mid x \equiv y\}$ where $\equiv$ is the equivalence relation generated by $\sim$.

Now we can construct V_D as follows:

$$\begin{aligned} V_D = \{&[x] \mid x \in V_B \mathbin{\dot\cup} V_C \\ &\wedge \nexists a \in V_A : ((f(a) = \bot \wedge g(a) \equiv x) \vee (f(a) \equiv x \wedge g(a) = \bot))\} \end{aligned}$$

Before we can construct the set of edges, we first construct three sets: $E_{D,\text{add}}$ (edges that are being added), $E_{D,\text{del}}$ (edges that are being removed) and $E_{D,\text{all}} \subseteq (V_D \times Lab \times V_D)$ (all edges where the source and target exist in V_D).

$$E_{D,\text{add}} = \{([s], l, [t]) | (s, l, t) \in (E_B \mathbin{\dot\cup} E_C) \setminus (f_E(E_A) \mathbin{\dot\cup} g_E(E_A))\}$$

$$\begin{aligned} E_{D,\text{del}} = \{&([s], l, [t]) | (s, l, t) \in (E_B \mathbin{\dot\cup} E_C) \wedge \exists a \in E_A : (f(a) \equiv (s, l, t) \\ &\wedge g(a) = \bot) \vee (g(a) \equiv (s, l, t) \wedge f(a) = \bot)\} \end{aligned}$$

$$E_{D,\text{all}} = \{([s], l, [t]) | (s, l, t) \in (E_B \mathbin{\dot\cup} E_C) \wedge [s] \in V_D \wedge [t] \in V_D\}$$

The set E_D is constructed as follows:

$$E_D = E_{D,\text{all}} \setminus (E_{D,\text{del}} \setminus E_{D,\text{add}})$$

[1] Unfortunately, this construction is incorrect, as we show in Appendix A.2

The morphisms $f'_V : V_C \to V_D$ and $f'_E : E_C \to E_D$) are defined as follows: (g'_V and g'_E are defined analogously)

$$f'_V(c) = \begin{cases} [c] & \text{if } [c] \in V_D \\ \bot & \text{otherwise} \end{cases}$$

$$f'_E((s,l,t)) = \begin{cases} ([s],l,[t]) & \text{if } ([s],l,[t]) \in E_D \\ & \wedge(([s],l,[t]) \in E_{D,\text{del}} \Rightarrow (s,l,t) \notin g_E(E_A)) \\ \bot & \text{otherwise} \end{cases}$$

Before we show that our pushout construction is correct, we first show that f' and g' are jointly surjective.

Lemma A.1.1. *The morphisms f' and g', as defined in Construction 2.3.4 are jointly surjective.*

Proof. To show that the morphism pair (f', g') is jointly surjective, we need to show that the pairs (f'_V, g'_V) and (f'_E, g'_E) are jointly surjective.

We will first show that f'_V and g'_V are jointly surjective. Let $[x] \in V_D$. We know from the definition of V_D that there exists a $v \in V_B \dot\cup V_C$ such that $v \in [x]$. If $v \in V_C$, then we know from the definition of f'_V that $f'_V(v) = [x]$. Similarly if $v \in V_D$, we know that $g'_V(v) = [x]$. This means that f'_V and g'_V are indeed jointly surjective.

Next, we will show that f'_E and g'_E are jointly surjective. Let $e = ([s], l, [t]) \in E_D$. We will show that there exists a $b \in E_B$ such that $g'_E(b) = e$, or there exists a $c \in E_C$ such that $f'_E(c) = e$. We make a case distinction based on whether $e \in E_{D,\text{del}}$:

- $e \notin E_{D,\text{del}}$
  We know $E_D \subseteq E_{D,\text{all}}$ this means that there must exist an edge $e' = (v_s, l, v_t) \in E_B \dot\cup E_C$ such that $v_s \in [s]$ and $v_t \in [t]$. Assume that $e' \in E_C$ (the other case is symmetric). Since $e \notin E_{D,\text{del}}$, we know by definition of f'_E that $f'_E(e') = ([s], l, [t]) = e$

- $e \in E_{D,\text{del}}$
  Since $e \in E_D$ and $e \in E_{D,\text{del}}$, we know (by definition of E_D) that $e \in E_{D,\text{add}}$. Therefore there must exist an $e' = (v_s, l, v_t) \in (E_B \dot\cup E_C) \setminus (f_E(E_A) \dot\cup g_E(E_A))$ such that $v_s \in [s]$ and $v_t \in [t]$. Assume that $e' \in E_C$ (the other case is symmetric). Now the implication in the definition of f'_E is satisfied ($e' \notin g_E(E_A)$), this means that $f'_E(e') = e$.

For both cases we have shown that e is in the image of either f'_E or g'_E, therefore f'_E and g'_E are jointly surjective. We have also shown that f'_V and g'_V are jointly surjective therefore we can conclude that f' and g' are jointly surjective. □

Proposition 2.3.5. *$B \xrightarrow{f'} D \xleftarrow{g'} C$ as defined in Construction 2.3.4 is a pushout.*[2]

[2] Actually this construction is incorrect, in Appendix A.2 we show that pushouts do not exist over all spans.

Proof. Let A, B, and C be graphs, let $f : A \to B$, and $g : A \to C$ be (partial) morphisms. Let $B \xrightarrow{f'} D \xleftarrow{g'} C$ be constructed as in Construction 2.3.4. In order to show that $B \xrightarrow{f'} D \xleftarrow{g'} C$ is indeed a pushout, we must show that $f' \circ g = g' \circ f$ and we must show that the following universal property is fulfilled: for all objects X and morphisms $h : B \to X$ and $k : C \to X$ with $k \circ g = h \circ f$, there is a unique morphism $x : D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$.

First we show that $f' \circ g = g' \circ f$. We will first prove this for all vertices of A. Let $v \in V_A$. We make a case distinction on whether a is in the domain of f_V and/or g_V:

- $a \in \text{dom}(f_V)$ and $a \in \text{dom}(g_V)$
  By definition of $\sim$ we have $f_V(a) \sim g_V(a)$ therefore $[f_V(a)] = [g_V(a)]$, now by definition of f'_V and g'_V we have $f'_V(g_V(a)) = g'_V(f_V(a))$.

- $a \notin \text{dom}(f_V)$ and $a \in \text{dom}(g_V)$
  We know that $g'_V(f_V(a)) = \bot$, so we will show that $f'_V(g_V(a)) = \bot$. Because $f_V(a) = \bot$ and $g_V(a) \in [g_V(a)]$ the property expressed by the existential quantifier in the definition of V_D is not satisfied. We can conclude that $[g_V(a)] \notin V_D$, which means that $f'_V(g_V(a)) = \bot$

- $a \in \text{dom}(f_V)$ and $a \notin \text{dom}(g_V)$
  Analogous to the previous case

- $a \notin \text{dom}(f_V)$ and $a \notin \text{dom}(g_V)$
  We have $f_V(a) = \bot$ and $g_V(a) = \bot$ therefore $f'_V(g_V(a)) = g'_V(f_V(a)) = \bot$

Now we will show that $f' \circ g = g' \circ f$ also holds for any edge $e \in E_A$. Again we make a case distinction on whether e is in the domain of f_E and/or g_E:

- $e \in \text{dom}(f_E)$ and $e \in \text{dom}(g_E)$
  Because morphisms must preserve the sources and targets, we know that $s(e)$ and $t(e)$ also have an image under f and g. This means $[f_V(s(e))] = [g_V(s(e))]$ and $[f_V(t(e))] = [g_V(t(e))]$. Let $s = f_V(s(e))$, $t = f_V(t(e))$ and $l = l(e)$. We separate two cases, based on whether $([s], l, [t]) \in E_{D,\text{del}}$:
  - $([s], l, [t]) \in E_{D,\text{del}}$
    The implication in the definition of f'_E and g'_E is not satisfied, since clearly $f_E(e) \in f_E(E_A)$ and $g_E(e) \in g_E(E_A)$, therefore $f'_E(g_E(e)) = g'_E(f_E(e)) = \bot$
  - $([s], l, [t]) \notin E_{D,\text{del}}$
    The implication in the definition of f'_E and g'_E is satisfied and we can observe that $f'_E(g_E(e)) = g'_E(f_E(e))$.

- $e \notin \text{dom}(f_E)$ and $e \in \text{dom}(g_E)$
  We know that $g'_E(f_E(e)) = \bot$, so we will show that $f'_E(g_E(e)) = \bot$. Morphisms preserve sources and targets, this means that $g_V(s(e)) \neq \bot$ and $g_V(t(e)) \neq \bot$ Let $s = g_V(s(e))$, $t = g_V(t(e))$ and $l = l(e)$. Because $f(e) = \bot$ we know that $([s], l, [t]) \in E_{D,\text{del}}$. We can see that the implication in the definition of f'_E is false. This means that $f'_E(g_E(e)) = \bot$.

- $e \in \text{dom}(f_E)$ and $e \notin \text{dom}(g_E)$
  Analogous to the previous case

- $e \notin \text{dom}(f_E)$ and $e \notin \text{dom}(g_E)$
  We have $f_E(e) = \perp$ and $g_E(e) = \perp$ therefore $f'_E(g_E(e)) = g'_E(f_E(e)) = \perp$

Since for all cases $f'(g(a)) = g'(f(a))$, we know that $f' \circ g = g' \circ f$

For a given graph X with morphisms $h : B \to X$ and $k : C \to X$ such that $k \circ g = h \circ f$ we define $x : D \to X$ such that

$$x_V([d]) = \begin{cases} h(b) & \text{if } \exists b \in V_B : b \equiv d \\ k(c) & \text{if } \exists c \in V_C : c \equiv d \end{cases}$$

$$x_E(([s], l, [t])) = \begin{cases} h((b_1, l, b_2)) & \text{if } \exists b_1, b_2 \in V_B : b_1 \equiv s \wedge b_2 \equiv t \\ k((c_1, l, c_2)) & \text{if } \exists c_1, c_2 \in V_C : c_1 \equiv s \wedge c_2 \equiv t \end{cases}$$

First we will show that x is well-defined. The following reasoning holds for both edges and vertices. Because f' and g' are jointly surjective, we know that at least one of the cases of x_V or x_E will occur. If both occur then $\equiv$ implies the existence of $a_1, \ldots, a_n \in A$ with $f(a_1) = b$, $g(a_1) = g(a_2)$, $f(a_2) = f(a_3)$, ..., $g(a_{n-1}) = g(a_n) = c$, which implies $h(b) = k(c)$ using $h \circ f = k \circ g$. Similarly, $b_1 \equiv b_2$ implies $h_V(b_1) = h_V(b_2)$ and $c_1 \equiv c_2$ implies $k_V(c_1) = k_V(c_2)$. Therefore x is well-defined.

We must also show that $x \circ g' = h$ and $x \circ f' = k$. This clearly follows from the definition[3].

The last thing we must show is that x is unique. We know (Lemma A.1.1) that f' and g' are jointly surjective, which implies (Proposition 2.2.13) that f' and g' are jointly epimorphic. Therefore we can conclude that x is unique. □

Proposition 3.1.8. *Let $r : L \to R$ and $m : L \to G$ be* ***Graph^P****-morphisms, such that $m \in \mathcal{M}$, where $\mathcal{M}$ is the class of total graph morphisms. Then r is $\mathcal{M}$-*preserving *w.r.t. m if and only $m(x) = m(y)$ implies $x, y \in \text{dom}(r)$ or $x, y \notin \text{dom}(r)$*[4].

Proof. We prove this proposition using Construction 2.3.4.

($\Rightarrow$) First we assume that $m(x) = m(y)$ implies $x, y \in \text{dom}(r)$ or $x, y \notin \text{dom}(r)$. To show that r is $\mathcal{M}$-preserving w.r.t. m, we much show that m' is total, i.e. we need to show that every vertex and edge in R is has an image under m'.

- Let $v \in V_R$. If v has no preimage under r_V, then we know that $[v] = \{v\}$ and $[v] \in V_H$. Therefore $m'_V(v) = [v]$. If v has one or more preimages under r_V, then we know that m_V is defined for every vertex in $r_V^{-1}(v)$ (because m is total), and because of our assumption ($m(x) = m(y)$ implies $x, y \in \text{dom}(r)$ or $x, y \notin \text{dom}(r)$) there does not exist an $a \in V_L$ such that $m_V(a) \equiv v$ and $r_V(a) = \perp$. This means that $[v] \in V_H$ and $m'_V(v) = [v]$.

- Let $e = (s, l, t) \in E_R$, we will show that e is has an image under m'_E. We have already shown that all vertices in V_R have an image in V_H, this means s and t have an image in V_H. If e has no preimage under r_E then

[3] Actually this is not the case, this is where we have made a mistake in our proof, in Appendix A.2 we show a counterexample

[4] Unfortunately, we can not be sure that the pushout over r and m exist, therefore we can not be sure if r is (strictly) $\mathcal{M}$-preserving w.r.t. m, see Section 8.3 and Appendix A.2.

we know that $([s], l, [t]) \in E_{D,\text{add}}$, which means $([s], l, [t]) \in E_D$, so we have $m'_E(e) = ([s], l, [t])$. If e has one or more preimages under r_E then we know (because we assumed $m(x) = m(y)$ implies $x, y \in \text{dom}(r)$ or $x, y \notin \text{dom}(r)$) that there does not exist an $a \in E_L$ such that $m_E(a) \equiv e \wedge r_E(a) = \bot$. This means that $([s], l, [t]) \notin E_{D,\text{del}}$, and because the source and target exist in V_D we have $([s], l, [t]) \in E_D$, therefore $m'(e) = ([s], l, [t])$.

($\Leftarrow$) Conversely, let r be $\mathcal{M}$-preserving w.r.t. m, which means m' is total. We will show that this means that $m(x) = m(y)$ implies $x, y \in \text{dom}(r)$ or $x, y \notin \text{dom}(r)$. We will prove this by contradiction, let $x, y \in L$ such that $m(x) = m(y)$ and assume to the contrary that $x \in \text{dom}(r)$ and $y \notin \text{dom}(r)$. We know that $m(x) \equiv m(y)$. Now we make a case distinction based on whether x and y are vertices or edges:

- Vertices: Since $r_V(y) = \bot$, $[m(y)] \notin V_H$, we know that $m'_V(r_V(x)) = \bot$.
- Edges: Let $(s, l, t) = m(x)$, then we know $([s], l, [t]) \in E_{D,\text{del}}$. This means that $m'_E(r_E(x)) = \bot$.

For both vertices and edges we have seen that m' not total, which is a contradiction. We may conclude $x, y \in \text{dom}(r)$ or $x, y \notin \text{dom}(r)$, therefore r is $\mathcal{M}$-preserving w.r.t m. □

A.2 Incorrectness of Pushout Construction in *Graph*^{P}

Unfortunately, when finishing this thesis, we discovered a counterexample which shows that our pushout construction in ***Graph***^{P} (Construction 2.3.4) is incorrect. In this section we will show a situation where a pushout in ***Graph***^{P} does not exist.

Figure A.1 shows morphisms $f : A \to B$ and $g : A \to C$. Construction 2.3.4 results in the cospan $C \xrightarrow{f'} D \xleftarrow{g'} B$ (as shown in Figure A.1). If $C \xrightarrow{f'} D \xleftarrow{g'} B$ is indeed the pushout over f and g, then the following pushout property must hold:

> For all objects X and morphisms $h : B \to X$ and $k : C \to X$ with $k \circ g = h \circ f$, there is a unique morphism $x : D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$.

We will show that this is not the case. Consider the object X and the morphisms $h : B \to X$ and $k : C \to X$ as shown in Figure A.1. We can see that $k \circ g = h \circ f$ therefore there must exist a morphism $x : D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$. However there does not exist a morphism x with these properties.

The problem is the edge $(1, \epsilon, 3) \in B$ which is mapped to $(1, \epsilon, (2{,}3)) \in X$ under h. If we follow the construction of the edge component x in Proposition 2.3.5, then we would obtain the total morphism $x_E : D \to X$ which maps the edge $(1, \epsilon, (2{,}3)) \in D$ to the edge $(1, \epsilon, (2{,}3)) \in X$. However, we can see that $x_E \circ g'_E \neq h_E$, because h_E is not defined for the edge $(1, \epsilon, 2) \in B$, but $x_E \circ g'_E$ is defined for the same edge.

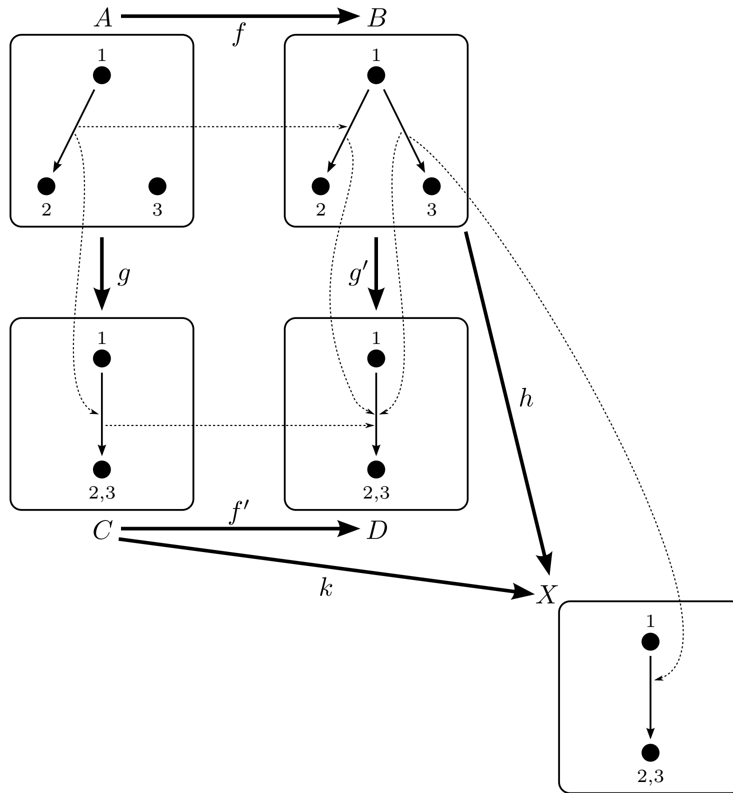


Figure A.1: $C \xrightarrow{f'} D \xleftarrow{g'} B$ is not the pushout over f and g. The dotted lines depict the edge component of the graph morphisms; the numbers show how the vertices are mapped under the graph morphisms.

It turns out there does not exist a pushout in this situation. Our proof for the pushout construction (Proposition 2.3.5) was incorrect. A consequence of this is that $(\boldsymbol{Graph^P}, \mathcal{M}, \mathcal{R})$ is not an SPO category (in the sense of Definition 2.4.1), because given an $\mathcal{M}$-morphism $m : L \to G$ and an $\mathcal{R}$-morphism $r : L \to R$, the pushout over m and r may not exist.

Bibliography

[1] U. Assmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In *Compiler Construction*, pages 121–135. Springer, 1996.

[2] S. Awodey. *Category theory.* Oxford University Press, 2010.

[3] R. Bardohl. A visual environment for visual languages. *Science of Computer Programming*, 44(2):181–203, 2002.

[4] P. Barroso and A. Furtado. Implementing a data definition facility driven by graph grammars. *Computer Languages*, 3(2):65–74, 1978.

[5] E. T. Bell. Exponential polynomials. *The Annals of Mathematics*, 35(2):258–277, 1934.

[6] D. Berend and T. Tassa. Improved bounds on bell numbers and on moments of sums of random variables. *Probability and Mathematical Statistics*, 30(2):185–205, 2010.

[7] V. Claus, H. Ehrig, and G. Rozenberg. *Graph-grammars and their application to computer science and biology*, volume 73. Springer, 1979.

[8] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of theoretical computer science*, volume B, pages 243–320. Elsevier, 1989.

[9] H. Ehrig, K. Ehrig, J. De Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. *Termination criteria for model transformation.* Springer, 2005.

[10] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation.* Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[11] H. Ehrig and C. Ermel. Semantical correctness and completeness of model transformations using graph and rule transformation. In *Graph Transformations*, pages 194–210. Springer, 2008.

[12] H. Ehrig, C. Ermel, F. Hermann, and U. Prange. On-the-fly construction, correctness and completeness of model transformations based on triple graph grammars. In A. Schrr and B. Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 241–255. Springer Berlin Heidelberg, 2009.

[13] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation: Part II: Single pushout approach and comparison with double pushout approach. In *Handbook of graph grammars and computing by graph transformation* [36], pages 247–312.

[14] H. Ehrig and M. Löwe. Categorical principles, techniques and results for high-level-replacement systems in computer science. *Applied Categorical Structures*, 1(1):21–50, 1993.

[15] H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: An algebraic approach. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 167–180. IEEE, 1973.

[16] G. Engels, R. Heckel, G. Taentzer, and H. Ehrig. A combined reference model-and view-based approach to system specification. *International Journal of Software Engineering and Knowledge Engineering*, 7(04):457–477, 1997.

[17] G. Engels, R. Heckel, G. Taentzer, and H. Ehrig. A view-oriented approach to system modelling based on graph transformation. In *Software Engineering ESEC/FSE'97*, volume 1301 of *Lecture Notes in Computer Science*, pages 327–343. Springer, 1997.

[18] G. Engels, C. Lewerentz, W. Schfer, A. Schörr, and B. Westfechtel, editors. *Graph Transformations and Model-Driven Engineering*, volume 5765 of *Lecture Notes in Computer Science*. Springer, 2010.

[19] U. Golas, L. Lambers, H. Ehrig, and F. Orejas. Attributed graph transformation with inheritance: Efficient conflict detection and local confluence analysis using abstract critical pairs. *Theoretical Computer Science*, 2012.

[20] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A graph-oriented object database model. *Knowledge and Data Engineering, IEEE Transactions on*, 6(4):572–586, 1994.

[21] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM systems journal*, 45(3):451–461, 2006.

[22] H. Kastenberg. Towards attributed graphs in groove: Work in progress. *Electronic Notes in Theoretical Computer Science*, 154(2):47–54, 2006.

[23] L. Lambers. Adhesive high-level replacement systems with negative application conditions. Technical report, Technische Universität Berlin, 2007.

[24] L. Lambers, H. Ehrig, and F. Orejas. Conflict detection for graph transformation with negative application conditions. *Graph Transformations*, pages 61–76, 2006.

[25] L. Lambers, H. Ehrig, and F. Orejas. Efficient detection of conflicts in graph-based model transformation. *Electronic Notes in Theoretical Computer Science*, 152:97–109, 2006.

[26] L. Lambers, H. Ehrig, and F. Orejas. Efficient conflict detection in graph transformation systems by essential critical pairs. *Electronic Notes in Theoretical Computer Science*, 211:17–26, 2008.

[27] M. Löwe and J. Müller. Critical pair analysis in single-pushout graph rewriting. In *Colloquium on Graph Transformation and its Application in Computer Science*, pages 71–77, 1995.

[28] T. Mens. On the use of graph transformations for model refactoring. In R. Lmmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 219–257. Springer Berlin Heidelberg, 2006.

[29] U. G. Montanari. Separable graphs, planar graphs and web grammars. *Information and Control*, 16(3):243–267, 1970.

[30] M. Nagl. Formal languages of labelled graphs. *Computing*, 16(1-2):113–137, 1976.

[31] D. Plump. Hypergraph rewriting: Critical pairs and undecidability of confluence. *Term graph rewriting: theory and practice*, 15:201–213, 1993.

[32] D. Plump. Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33(2):201–209, 1998.

[33] D. Plump. Confluence of graph transformation revisited. In *Processes, Terms and Cycles: Steps on the Road to Infinity*, pages 280–308. Springer, 2005.

[34] T. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5(6):560–595, 1971.

[35] A. Rensink. The groove simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance*, pages 479–485. Springer, 2004.

[36] G. Rozenberg and H. Ehrig. *Handbook of graph grammars and computing by graph transformation*, volume I: Foundations. World Scientific Singapore, 1997.

[37] A. Sanfeliu and K.-S. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man and Cybernetics*, 13(3):353–362, 1983.

[38] G. Taentzer. Agg: A graph transformation environment for modeling and validation of software. In *Applications of Graph Transformations with Industrial Relevance*, pages 446–453. Springer, 2004.

[39] M. Timmer. *Efficient modelling, generation and analysis of Markov automata*. PhD thesis, University of Twente, 2013.