Analysing Graph Programs for Confluence

Qualifying Dissertation

Ivaylo Hristakiev

September 5, 2014
Abstract

Graph transformation is concerned with rule-based manipulation of graphs. It is intrinsically non-deterministic in the sense that for a given input, several different outputs are possible. However, due to the richness of interactions between individual transformations, the computation can exhibit global determinism. This property is called confluence (or "don't care" nondeterminism) and allows us to view graph transformation as a function between inputs and outputs, an important property in application areas like model transformation.

Several extensions to graph transformations have been developed that enrich the computation model such as labels, application conditions and explicit control over transformations. However, the theory behind static confluence checking has not always kept up with this evolution.

The graph programming language GP, developed at York, is an experimental domain-specific language for high-level problem solving on graphs and graph-like structures, and implements the above extensions. The simple syntax and concise formal semantics of GP facilitate the reasoning on programs. However, it is not yet known how to obtain proofs of confluence for GP programs, which is the main aim of my research.

Confluence analysis opens up the possibility of several improvements. First, it impacts the run-time efficiency without compromising the semantics of transformations - if one cares only of the resulting outputs for a given input, then a confluent computation can only ever produce a unique result. Furthermore, if a confluent computation fails once, then all possible executions will fail. Moreover, if a programmer intended for a program to be functional, then a counter example to confluence can trigger a development iteration.

Current state of the art in confluence analysis is that of AGG and Groove. AGG is a development environment for attributed graph transformation systems that aims to provide tools for specifying and rapid prototyping of applications with complex, graph structured data. It supports confluence checking to the point of constructing conflicts and dependencies between rules, but without automated analysis. Groove is an object-oriented modelling environment supported by graph transformation. It is currently one of the fastest graph transformation system. A recent development added a confluence checker that is based generation and analysis of conflicts that is only partially correct.

Literature on graph transformation, graph programming and static confluence analysis is presented. Initial results of an attribution algorithm for GP labels are discussed together with early investigations of confluence analysis for graph programs.

A proposal is formed, based on the surveyed literature and these initial results. The overall goals are presented together with specific success criteria and strategy for achieving them.
# Contents

1 Introduction 4

2 Graph Transformation 5
   2.1 Graphs, Rules and Morphisms 5
   2.2 Pushouts and DPO with relabelling 8
   2.3 Embedding of transformations 11

3 Confluence 14
   3.1 Structural Confluence 14
   3.2 Critical Pairs 17
   3.3 Confluence and Attribution 21
   3.4 Confluence and Application Conditions 23
   3.5 Applications of Confluence 24
   3.6 Summary 25

4 Graph programming with GP 26
   4.1 GP labels and rule schemata 26
   4.2 Graph Programs 30
   4.3 Operational Semantics 32
   4.4 Summary 34

5 Preliminary results 35
   5.1 A Unification algorithm for GP 35
      5.1.1 Introduction 35
      5.1.2 Unification background 36
      5.1.3 Unification Algorithm 38
      5.1.4 Summary 40

6 Research Proposal 41
   6.1 Overall aims 41
   6.2 Modifying existing theory to GP rule schemata 41
   6.3 Implementing critical pair construction 41
   6.4 Determining strong joinability 42
   6.5 Confluence of graph programs 42
   6.6 Tool Structure 43
   6.7 Timeplan 44
1 Introduction

Graphs are natural way of specifying objects and their relationships in an intuitive way. They are used in many fields of computer science to model system state such as data and control flow diagrams, UML diagrams, Petri nets, hardware and software architectures [26, 38]. Graphs and graph theory have been subject to extensive research [3].

Graph transformation and graph grammars add to the static description of graphs by allowing us to express their dynamic evolution via graph transformation rules. Graph grammars have been applied to many fields of computer science such as model-driven software engineering [11, 16], database design [21], compiler construction [3], network architectures, distributed systems [57] and many more [15].

A transformation rule describes local changes to a graph, and a sequence of such rule applications contain the intended computational thinking. Rules are similar to pattern-matching and have a formal mathematical definition. Graph transformation has been subject to formal verification and analysis.

Graph transformation is non-deterministic - several rules may be applicable at the same time. This means that for any starting graph, there may be several different result graphs to which none of the rules apply. However, sometimes it may be the case that there is only one possible result due to how the rules interact with each other. This special property is called confluence and comes from the area of term and term graph rewriting.

A proof that a set of rules is confluent is done via constructing so-called critical pairs, conflicts in minimal context. Critical pairs can be detected and analysed statically and allow us to reason about all conflicts that may arise during computation.

Graph transformation has been extended in several ways. When equipped with sequential composition and as-long-as-possible iteration, it becomes computationally complete [23] which has been implemented in several graph transformation systems, one of them being York’s own GP [48, 37, 49]. GP was designed to be minimal and computationally complete with intuitive formal syntax and semantics. GP also allows the inserting of application conditions on rules that enable or forbid a rule from being applicable, which is another graph transformation extension. All of these extensions increase the richness of interactions between rules, and hence the increased complexity when proving confluence.

Confluence analysis for GP would be desirable for several reasons. The GP system is specified to compute all possible result graphs for a given input. If the program given is proven confluent, then only a single result graph should be computed, greatly increasing efficiency. Furthermore, GP has the explicit notion of failure, and if one possible execution of a confluent program produces failure, then all such executions will produce failure. Both of these improvements improve the execution run-time without compromising the program semantics. Furthermore, the behaviour of confluence in the presence of control constructs has no answer so far, and some application areas of graph transformation require both.

My research involves developing a static confluence checker for graph programs. The checker would be part of the GP compiler stack, currently being developed in C [6]. This would also involve extending the current theory of critical pairs to GP rule schemata and control constructs.

Chapter 2 is concerned with presenting the basics of graph transformation theory, and the double-pushout approach (DPO). Chapter 3 is concerned with the language GP and its main features - the lifting of rules to rule schemata and their combination using control constructs. Chapter 4 presents the property confluence, first in its basic form concerning unlabelled graphs, and later how it is extended over the above graph transformation features. Chapter 5 describes current work on static confluence checking for GP. Finally, Chapter 6 summarises the overall aims of the research and proposes a schedule for their implementation.
2 Graph Transformation

Graph transformation in its basic form is concerned with how to change one graph into another. These changes are called transformations (or derivations) and are guided by rules. Rules specify changes to a small substructure of the graph such as adding an edge between two nodes or removing a loop edge. Sometimes graph transformation is referred to as rule-based manipulation for this reason.

There exist several approaches to how exactly the rules operate. There are two main ones - the algorithmic (or set-theoretic) and the algebraic approaches [23]. The algebraic approach, on which this report is based, uses pushout constructions from category theory which model the gluing of graphs. The two variants, the single-pushout (SPO) and the double-pushout (DPO) approaches, handle some cases differently [13].

2.1 Graphs, Rules and Morphisms

We begin by defining graphs, and how graphs relate to each other by using structure-preserving mappings called morphisms. In our setting, edges are directed, nodes and edges have labels (unless specified), and parallel edges can exist. Graphs will be totally labelled, but in the case when the labels play no role we will omit them, formally represented by a \( \Box \) label which is not drawn.

**Definition 2.1** (Label alphabet). A label alphabet \( \mathcal{L} = (\mathcal{L}_V, \mathcal{L}_E) \) is a pair where \( \mathcal{L}_V \) is a set of node labels and \( \mathcal{L}_E \) is a set of edge labels.

**Definition 2.2** (Graph). A graph over a label alphabet \( \mathcal{L} \) is a system \( G = \langle V_G, E_G, s_G, t_G, l_G, m_G \rangle \) where

- \( V_G \) and \( E_G \) are finite set of nodes and edges
- \( s_G, t_G : E_G \to V_G \) are the source and target functions for edges
- \( l_G : V_G \to \mathcal{L} \) is a partial node labelling function
- \( m_G : E_G \to \mathcal{L} \) is a total edge labelling function

Given a node \( v \), \( l_G(v) = \perp \) expresses that \( l_G(v) \) is undefined. We say that node \( v \) is incident to edge \( e \) and vice versa if \( s_G(e) = v \) or \( t_G(e) = v \). Graph \( G \) is totally labelled if \( l_G \) is a total function. The classes of graphs and totally labelled graphs over \( \mathcal{L} \) are written as \( \mathcal{G}(\mathcal{L}_\perp) \) and \( \mathcal{G}(\mathcal{L}) \).

**Example 2.3** (A graph). Consider the graphs in Figure 1. The graph on the left is the pictorial representation of

\[
G_1 = \langle \{1, 2\}, \{e_1, e_2\}, s_{G_1} : (e_1 \mapsto 1, e_2 \mapsto 1), t_{G_1} : (e_1 \mapsto 2, e_2 \mapsto 2), l_{G_1} : \emptyset, m_{G_1} : (e_1 \mapsto \Box, e_2 \mapsto \Box) \rangle
\]

and the right one is a representation of

\[
G_2 = \langle \{3, 4\}, \{e_3, e_4, e_5\}, s_{G_2} : (e_3 \mapsto 3, e_4 \mapsto 3, e_5 \mapsto 3), t_{G_2} : (e_3 \mapsto 4, e_4 \mapsto 4, e_5 \mapsto 3), l_{G_2} : (1 \mapsto a, 2 \mapsto b), m_{G_2} : (e_3 \mapsto \Box, e_4 \mapsto \Box, e_5 \mapsto \Box) \rangle
\]

Both are over the label alphabet \( \mathcal{L} = \langle \{a, b\}, \{\Box\} \rangle \). We follow the convention that nodes are drawn as circles and edges as arrows. Node labels are drawn inside the circle and node identifiers are drawn next to the nodes. Edge labels are drawn next to the arrows. Edge identifiers are not usually drawn, but when they are relevant we add them in a id:label syntax next to the arrows. If the label is \( \Box \) then we just include the identifier in italics. Here \( G_1 \) is partially labelled and \( G_2 \) is totally labelled.
several ways to map $G$.

**Example 2.6.** (A premorphism and a graph morphism). From Example 2.3, we can find a premorphism $g : G \to H$ in $\mathcal{G}(\mathcal{L})$ that also preserves labels, i.e., $m_H \circ g_E = m_G$ and $l_H(g_V(v)) = l_G(v)$ for all $v \in \text{Dom}(l_G)$ such that $l_G(v) \neq \bot$.

**Definition 2.4** (Premorphism). A premorphism $g : G \to H$ consists of two functions $g_V : V_G \to V_H$ and $g_E : E_G \to E_H$ that preserve sources and targets:

$$s_H \circ g_E = g_V \circ s_G, \quad t_H \circ g_E = g_V \circ t_G.$$  

**Definition 2.5.** (Graph morphism). A graph morphism is a premorphism $g$ that also preserves labels, i.e., $m_H \circ g_E = m_G$ and $l_H(g_V(v)) = l_G(v)$ for all $v \in \text{Dom}(l_G)$ such that $l_G(v) \neq \bot$.

**Example 2.6.** (A premorphism and a graph morphism). From Example 2.3, we can find several ways to map $G_1$ to $G_2$:

$$f_1 = \{1 \mapsto 3, 2 \mapsto 4, \{e_1 \mapsto e_3, e_2 \mapsto e_4\}\}$$

$$f_2 = \{1 \mapsto 3, 2 \mapsto 4, \{e_1 \mapsto e_4, e_2 \mapsto e_3\}\}$$

$$f_3 = \{1 \mapsto 3, 2 \mapsto 3, \{e_1 \mapsto e_5, e_2 \mapsto e_5\}\}$$

We have that $f_1$ and $f_2$ are injective graph morphisms and $f_3$ is non-injective. All three preserve node and edge labels because $l_{G_1}$ is undefined for all nodes ($l_{G_1}(1) = \bot$ and $l_{G_1}(2) = \bot$) and edge labels are always $\bot$. Note that the mapping $h : G_2 \to G_1 = \{3 \mapsto 1, 4 \mapsto 2, \{e_3 \mapsto e_1, e_4 \mapsto e_2\}\}$ is not a proper graph morphism because nodes 3 and 4 end up unlabelled in $G_1$.

Graph morphisms can be composed: given morphisms $f : G \to H$ and $g : H \to T$ the composition $g \circ f$ is the morphism $(g_V \circ f_V, \ g_E \circ f_E)$ where $\circ$ denotes standard function composition.

A graph morphism is injective (surjective) if $g_V$ and $g_E$ are both injective (surjective). It is an isomorphism if it is injective, surjective, and maps unlabelled nodes to unlabelled nodes ($l_H(g(x)) = \bot$ for all nodes $v$ with $l_G(v) = \bot$), denoted by $G \cong H$. A graph morphism $g : G \to H$ is an inclusion if $g(x) = x$ for all nodes and edges in $G$.

A rule allows us to change the graph to which the rule is applied. In the setting of the double-pushout (DPO) approach, rule applications are local in the sense that all changes to the graph are as prescribed by the rule.

**Definition 2.7** (Rule). A rule $r = (L \leftarrow K \rightarrow R)$ over $\mathcal{L}$ consists of two inclusions $K \to L$ and $K \to R$ such that $L, K, R$ are graphs in $\mathcal{G}(\mathcal{L})$.

The left graph $L$ can be thought as the pattern that needs to be matched, and the right graph $R$ is what the pattern is replaced with. The interface $K$ is used to define how the right graph glues into the context of the rule application.
Example 2.8 (Example Rule). Figure 2 shows an example rule \( r \). It represents the replacement of a graph of two connected nodes with a loop on the first by a graph that is computed from the left graph by deleting both edges, and creating a new node labelled with 'b' that is connected to node 1. Note that nodes 1 and 2 are kept the same.

![Figure 2: Example rule](image)

Sometimes it will be easier to write rules in the shorthand \( \langle L \Rightarrow R \rangle \) instead of \( \langle L \leftarrow K \rightarrow R \rangle \). In that case we assume that \( K \) consists of only nodes that have the same labels as their images in \( L \) and \( R \).

![Figure 3: Example direct derivation](image)

Informally, an application of a rule \( r \) to a graph will remove the items in \( L - K \), preserve \( K \), add the items in \( R - K \).

**Definition 2.9** (Dangling condition). Let \( r = \langle L \leftarrow K \rightarrow R \rangle \) be a rule and \( G \) a graph in \( G(L) \). A graph morphism \( g : L \rightarrow G \) satisfies the dangling condition if no node in \( g(L) - g(K) \) is incident to an edge in \( G - g(L) \).

The morphism \( g \) is called the match of \( r \) in \( G \). The dangling condition guarantees that the graph \( D \) obtained by removing nodes in \( L - K \) is proper and has no edges without source or target.

**Definition 2.10** (Direct Derivation). Let \( r = \langle L \leftarrow K \rightarrow R \rangle \) be a rule, \( G \) a graph in \( G(L) \), and \( g : L \rightarrow G \) an injective graph morphism satisfying the dangling condition. The transformation of \( G \) to \( H \) via \( r \), denoted by \( G \Rightarrow_{r,g} H \) or \( G \Rightarrow_r H \), is a direct derivation if \( H \cong H' \) where \( H' \) is constructed as follows:

1. Remove all nodes and edges in \( g(L) - g(K) \), obtaining graph \( D \)
2. Add disjointly to $D$ all nodes and edges from $R - K$, keeping their labels, obtaining graph $H'$. For each edge $e$ in $R - K$, define source like this:

$$s_{H'}(e) = \begin{cases} s_R(e) & \text{if } s_R(e) \in V_R - V_K \\ g_V(s_R(e)) & \text{otherwise} \end{cases}$$

3. Define target analogously as source.

If $R$ is a set of rules, then $G \Rightarrow_R H$ means that there is some $r \in R$ such that $G \Rightarrow_r H$. A sequence of direct derivations

$$G \Rightarrow_R H_1 \Rightarrow_R H_2 \Rightarrow_R \ldots \Rightarrow_R H_n$$

can be written as $G \Rightarrow_R H_n$ (or $G \Rightarrow H_n$ if $R$ is obvious from the context), expressing that $H$ is derived from $G$ in 0 or more rule applications. A direct derivation of length 0 is the isomorphism relation $G \cong H$.

### 2.2 Pushouts and DPO with relabelling

The above definition of rule application is enough for operational purposes, but makes it difficult to reason about graph transformation at an abstract level. In the DPO approach, the definition can be restructured using the notion of a pushout from category theory. This allows us to use results about pushouts in the setting of graph transformation.

**Definition 2.11** (Pushout). Given a graph $A$ together with morphisms $b : A \to B$ and $c : A \to C$, a pushout over $(A,b,c)$ is a graph $D$ together with morphisms $b' : B \to D$ and $c' : C \to D$ such that

- **Commutativity**: the morphisms commute, i.e. $b' \circ b = c' \circ c$
- **Uniqueness**: for every pair of morphisms $b'' : B \to D'$ and $c'' : C \to D'$ such that the outer diagram commutes $(b'' \circ b = c'' \circ c)$, there exists a unique morphism $d : D \to D'$ such that the triangles commute $b'' = d \circ b'$ and $c'' = d \circ c'$.

A pushout is the formal way of gluing two objects $B$ and $C$ along a common object $A$. The graph $D$ is the disjoint union of $B$ and $C$ except on elements that are common with $A$, which are merged.

Pushouts have several important properties. First, every node in $D$ has a pre-image in either $B$ or $C$ (if in both, then it is also in $A$). Second, if $c : A \to C$ is injective (surjective), then $b' : B \to D$ is also injective (surjective). Third, $D$ is unique up to isomorphism - if another graph $D'$ was a pushout of $(A,b,c)$, then by the uniqueness property there would be morphisms from $D$ to $D'$ and from $D'$ to $D$, i.e. an isomorphism $D \cong D'$.

A pushout complement is the last definition we need in order to formally define rule applications. It is similar to that of pushouts:
Definition 2.12 (Pushout complement). Given a graph $A$ together with morphisms $c : A \to C$ and $c' : C \to D$, the pushout complement of $(A, c, c')$ is a graph $B$ together with morphisms $b : A \to B$ and $b' : B \to D$ such that $(D, b', c')$ is a pushout of $(A, b, c)$.

Algorithms for construction of pushouts and pushout complements in the category of totally labelled graphs can be found in [10].

A direct derivation can be viewed as a pushout complement construction followed by a pushout construction. It can be shown that this is equivalent to Definition 2.10, so we can redefine it:

Definition 2.13 (Direct Derivation). Let $r = \langle L \leftarrow K \rightarrow R \rangle$ be a rule, $G$ a graph in $\mathcal{G}(\mathcal{L})$, and $g : L \to G$ an injective graph morphism satisfying the dangling condition. The transformation of $G$ to $H$ via $r$ is given by the double-pushout diagram in Figure 4 with pushouts (1) and (2).

![Figure 4: The double-pushout (DPO) approach](image)

Figure 4: The double-pushout (DPO) approach

Even though the DPO approach allows us to further reason about graph transformation at an abstract level by using results about (de)composition of pushouts and pushout complements, it has the shortcoming of not handling relabelling very well. For example, if we wanted to relabel an edge, we can do so by deleting it and recreating it with the desired label. However, to relabel a node, we have to delete it and then recreate it. This is only possible if the node is isolated due to the dangling condition. Habel and Plump [24] solve this problem by requiring that pushouts (1) and (2) are also pullbacks. The paper also keeps uniqueness of direct derivations (up to isomorphism). Furthermore, the approach relaxes the requirement that all graphs are totally labelled. As we shall see later, in GP $L$ and $R$ are totally labelled and $K$ is unlabelled on the nodes that we wish to relabel.

Definition 2.14 (Pullback). Given a graph $D$ together with morphisms $b' : B \to D$ and $c' : C \to D$, a pullback over $(D, b, c)$ is a graph $A$ together with morphisms $b : A \to B$ and $c : A \to C$ such that

- **Commutativity**: the morphisms commute, i.e. $b' \circ b = c' \circ c$
- **Uniqueness**: for every pair of morphisms $b'' : A' \to B$ and $c'' : A' \to C$ such that the outer diagram commutes $(b'' \circ b' = c'' \circ c')$, there exists a unique morphism $a : A' \to A$ such that the triangles commute $b'' = b \circ a$ and $c'' = c \circ a$.

![Diagram for pullback](image)
Pullbacks are the dual construction of pushouts in category theory. Similarly, they represent an intersection of two objects over a common object. The results for pullbacks are similar to those of pushouts.

**Definition 2.15** (Natural Pushout). A natural pushout is a pushout that is also a pullback.

**Fact 1** (Uniqueness, Composition, and Decomposition of POs, PBs, NPOs). The composition and decomposition of POs, PBs and NPOs results again in a PO, PB and NPO respectively, i.e. given the following commutative diagram, the statements below are valid:

\[
\begin{array}{ccc}
A & \rightarrow & B \\
\downarrow & & \downarrow \\
C & \rightarrow & D \\
\end{array} \quad \begin{array}{ccc}
(1) & & (2) \\
\downarrow & & \downarrow \\
E & \rightarrow & F \\
\end{array}
\]

1. Pushout composition: if (1) and (2) are POs, then (1) + (2) is also a PO
2. Pushout decomposition: if (1) and (1) + (2) are POs, then (2) is also a PO
3. Pullback composition: if (1) and (2) are PBs, then (1) + (2) is also a PB
4. Pullback decomposition: if (2) and (1) + (2) are PBs, then (1) is also a PB
5. Natural pushout composition: if (1) and (2) are NPOs, then (1) + (2) is also a NPO

*Proof.*
For 1-4, see Sections 2.4 and 2.5 in [10]. For 5, it follows directly from 1 and 3 and definition of NPO.

**Definition 2.16** (Direct Derivation with relabelling). Let \( r = \langle L \leftarrow K \rightarrow R \rangle \) be a rule, \( L, R \in \mathcal{G}(\mathcal{L}), K \in \mathcal{G}(\mathcal{L}_\perp) \), \( G \) a graph in \( \mathcal{G}(\mathcal{L}) \), and \( g : L \rightarrow G \) an injective graph morphism satisfying the dangling condition. A direct derivation with relabelling is a direct derivation in the sense of Definition 2.10, but with the extra step:

4. For each unlabelled node \( v \) in \( K \), \( l_H(g_V(v)) \) becomes \( l_R(v) \)

The diagrammatic change is that pushouts (1) and (2) in Definition 2.13 have to be natural.

Habel and Plump [23] give a characterization of natural pushouts in the sense of the following lemma.

**Lemma 2.17.** The pushout in Definition 2.11 is natural if and only if for all elements \( x \in A \), \( l_A(x) = \perp \) implies \( l_B(b(x)) = \perp \) or \( l_C(c(x)) = \perp \).

This restriction means that for every node we want to relabel, first we omit a label in the interface \( K \), and second, we omit a label in the gluing graph \( D \). As a result, the graph \( H \) will be unique up to isomorphism, and will be totally labelled if \( G \) is totally labelled [24].
2.3 Embedding of transformations

In this section, we present the problem of under what conditions a graph transformation \( t : G_0 \Rightarrow G_n \) can be extended to a graph transformation \( t' : G'_0 \Rightarrow G'_n \) via an extension morphism \( k_0 : G_0 \rightarrow G'_0 \). This result is the basis to prove many other results in graph transformation. In particular, embedding is needed to prove the Local Confluence Theorem in Section 3.

The idea of embedding is to obtain an extension diagram (1) below, where the same sequence of transformations \( t_1, t_2, \ldots, t_n \) are applied in the same order in \( t \) and \( t' \):

\[
\begin{array}{c}
t : G_0 \Rightarrow \overset{*}{G_n} \\
k_0 \downarrow \hspace{1cm} \downarrow k_n \\
t' : G'_0 \Rightarrow \overset{*}{G'_n}
\end{array}
\]

This is not always possible. The idea is to first construct a boundary graph \( B \) and a context graph \( C \) for \( k_0 \), such that \( G'_0 \) is the gluing of \( G_0 \) and \( C \) along \( B \), i.e. \( G'_0 = G_0 +_B C \). For an extension to exist, \( t \) must preserve \( B \). This means that none of the transformations \( t_1, t_2, \ldots, t_n \) of \( t \) delete any item of \( B \).

**Definition 2.18 (Boundary and Context).** In the category of unlabelled graphs, the boundary object \( B \) of an injective graph morphism \( k : G \rightarrow G' \) consists only of nodes of \( G \) that are incident to ”dangling” edges, i.e. all nodes \( x \in G \) such that \( k(x) \) is adjacent to an edge in \( G' \setminus k(G) \). These nodes are needed to glue \( G \) to the context graph \( C = G' \setminus k(G) \cup k(b(B)) \) (the boundary \( B \) together with all elements that are to be deleted) in order to obtain \( G'' \) as the gluing of \( G \) and \( C \) via \( B \). \( C \) is called the context of \( k \).

**Example 2.19 (Context and Boundary).** Consider the graphs \( K \) and \( L \) below that are part of rule describing the deletion of node (4) and the corresponding edge.
Graph $B$ consists only of nodes of $G$ that are incident to "dangling" edges, i.e. just node (3). The context graph $C$ is the boundary $B$ together with all elements that are to be deleted - in this case nodes (3) and (4) and the edge between them.

The diagram is called an *initial pushout*. If the morphism $k$ is not injective, we have to add to the boundary $B$ all nodes and edges of $G$ that are merged in $G'$ (i.e. $k(x) = k(y)$) and those nodes that are the source or target of two edges that are equally mapped by $k$.

In order to define a boundary consistency in a formal way, first we consider *derived rules* (often referred to in the literature as *derived spans*). A derived rule describes the combined changes of a sequence of transformations by condensing it into a single transformation from $G_0$ to $G_n$.

**Definition 2.20** (Derived rule). The derived rule of an identical transformation $t : G \Rightarrow id \ G$ is defined by $der(t) = (G \leftarrow G \rightarrow G)$ with identical morphism.

The derived rule of a direct transformation $G \Rightarrow H$ is the rule $(G \leftarrow D \rightarrow H)$.

For a transformation $t : G_0 \Rightarrow G_n \Rightarrow G_{n+1}$, the derived rule is the composition via the pullback $P$ of the derived rules $der(G_0 \Rightarrow G_n) = (G_0 \leftarrow D' \rightarrow G_n)$ and $der(G_n \Rightarrow G_{n+1}) = (G_n \leftarrow D_n \rightarrow G_{n+1})$. This construction leads to the derived rule $der(t) = (G_0 \leftarrow D \rightarrow G_{n+1})$.

The derived rule of a transformation is unique up to isomorphism and does not depend on the order of the pullback constructions.

**Definition 2.21** (Consistency). Given a transformation $t : G_0 \Rightarrow G_n$ with a derived rule $der(t) = (G_0 \leftarrow D \rightarrow G_n)$, a graph morphism $k_0 : G_0 \rightarrow G'_0$ is called *consistent* with respect to $t$ if there exist an initial pushout (1) over $k_0$ and a graph morphism $b$ with $d_0 \circ b = b_0$.

$$
\begin{array}{c}
\text{Graph } B \text{ consists only of nodes of } G \text{ that are incident to "dangling" edges, i.e. just node (3).} \\
The context graph } C \text{ is the boundary } B \text{ together with all elements that are to be deleted - in this case nodes (3) and (4) and the edge between them.}
\end{array}
$$
Informally this means that \( t \) does not change the boundary \( B \) of \( k_0 \).

It can be shown that consistency is both sufficient and necessary for the construction of extension diagrams.

**Theorem 2.22** (Embedding Theorem). Given a derivation \( t : G_0 \Rightarrow^* G_n \) and a morphism \( k_0 : G_0 \rightarrow G'_0 \) which is consistent with respect to \( t \), then there is an extension diagram over \( t \) and \( k_0 \).

The consistency condition is also necessary for the construction of extension diagrams. Furthermore, it is possible to directly construct \( G'_n \) from the derived rule of \( t \) and \( k_0 \) as a single pushout construction.

As already noted, embedding is used for the Local Confluence Theorem that is concerned with how critical overlays are embedded into bigger contexts.
3 Confluence

As mentioned already, the computational model of graph transformation is non-deterministic. However, in several application areas of graph transformation it is important that there exists exactly one final result. Given an input graph $G$, the final result of the computation on $G$ is a graph $H$ to which none of the transformation rules can be further applied. $H$ is called a normal form of $G$. If each graph has a unique normal form, then the order of transformations does not matter.

The property of which a non-deterministic transformation system produces deterministic results is called confluence. The notion originally comes from (abstract) rewriting systems [31, 30]. A system of transformation rules $R$ is confluent if for every pair of diverging transformations $H_1 \xrightarrow{\text{R}} G$ and $G \xrightarrow{\text{R}} H_2$, there is a common graph $H$ such that $H_1 \xrightarrow{\text{R}} H \xleftarrow{\text{R}} H_2$. Confluence ensures there exists at most one normal form.

It is well known that confluence is an undecidable property [45]. However, a sufficient lemma has been developed that studies minimal conflicting situations called critical pairs. They are useful in the sense that there exists a critical pair for every conflict that may arise during computation and hence can be used to argue that a rewriting system is confluent.

Confluence is an important property in several aspects. First, together with termination it guarantees the global determinism (functional behaviour) of transformations. This is important when transformations are used to implement functions. Second, on a more practical level, it alleviates the costs associated with non-determinism while keeping semantic equivalence. On a similar note, in the presence of failure, if a sequence of derivations fails, then any possible sequence from the same start graph will also fail.

Counter examples to confluence, i.e. pairs of rules which are dependent, can also be useful. An implementation may choose to apply independent rules as much as possible and then start creating decision points for backtracking, thus reducing memory footprint. Furthermore, the proof of non-confluence enables so-called completion that adds extra rules until confluence is proven while preserving semantic equivalence and termination.

The rest of the chapter is as follows - first we properly define what it means for two transformations to be in conflict; next, we define critical pairs and give algorithms for their computation; last, we extend critical pairs over attribution, typing and application conditions.

3.1 Structural Confluence

We start by investigating the proper definitions of confluence and its flavour local confluence.

**Definition 3.1** (Confluence). A pair of derivations $G \xrightarrow{\text{R}} H_1$ and $G \xrightarrow{\text{R}} H_2$ is confluent if there exist transformations and $H_1 \xrightarrow{\text{R}} G'$ and $H_2 \xrightarrow{\text{R}} G'$. A graph transformation system $\mathcal{R}$ (a set of rules) is confluent if for all graphs $G$, all pairs of derivations from $G$ are confluent.

Informally, this means that two diverging transformations are confluent if they can be joined again.

**Remark 1.** For the rest of this section we will only consider unlabelled graphs ($\mathcal{G}(L_\bot)$), i.e. will only be interested in the structural aspect of confluence. Later on we will discuss how attribution affects confluence.

---

1 One such application is model transformation. See [32] for a comprehensive comparison between existing model transformation systems, one of the criteria for which is the existence of confluence analysis.

2 See [10] for a comprehensive introduction to confluence.
A weaker version of confluence is called local confluence, which restricts the pair of derivations in the definition to direct derivations (derivations of length 1). According to a general result for rewriting systems, it is sufficient to consider local confluence, given that the system is terminating.

**Definition 3.2 (Local Confluence).** A pair of direct derivations \( G \Rightarrow_{R} H_1 \) and \( G \Rightarrow_{R} H_2 \) is **locally confluent** if there exist transformations and \( H_1 \Rightarrow_{R} G' \) and \( H_2 \Rightarrow_{R} G' \). A graph transformation system \( R \) is **locally confluent** if for all graphs \( G \), all pairs of direct derivations from \( G \) are locally confluent.

The difference between confluence and local confluence is illustrated in Figure 6.

![Figure 6: Confluence and Local Confluence](image)

Local confluence is strictly weaker because it does **not** imply confluence, as shown in the following example.

**Example 3.3 (Local confluence does not imply confluence).** Consider the locally confluent system with four different direct transformations.

\[
A \leftarrow B \rightarrow C \rightarrow D
\]

The pairs \( A \Leftrightarrow B \Rightarrow C \) and \( B \Leftrightarrow C \Rightarrow D \) are locally confluent because they can be joined by \( C \Rightarrow B \Rightarrow A \) and \( B \Rightarrow C \Rightarrow D \) respectively.

However, we can see that \( A \) and \( C \) are two unique normal forms of \( B \) (cannot be transformed further) - the pair \( A \Leftrightarrow B \Rightarrow D \) cannot be joined. Similarly, the pair \( A \Leftrightarrow C \Rightarrow D \) cannot be joined. Therefore, the system is not confluent.

**Definition 3.4 (Termination).** A system of rules \( R \) is **terminating** if there is no infinite sequence of transformations

\[
G_0 \Rightarrow_{R} G_1 \Rightarrow_{R} G_2 \Rightarrow_{R} \ldots
\]

Termination ensures the existence of at least one normal form. However, termination is also an undecidable property [46]. Some termination criteria for graph transformation systems have been shown in [14].

**Lemma 3.5 (Newman’s Lemma).** Every terminating and locally confluent graph transformation system is also confluent.

The above example is non-terminating because there exists an infinite sequence

\[
B \Rightarrow C \Rightarrow B \Rightarrow \ldots
\]

Hence, the local confluence of the system does not imply confluence. However, one might suspect that the cycle between \( B \) and \( C \) is responsible which turns out to not be the case - even for acyclic relations, local confluence does not imply confluence [1 Section 2.7].
In order to reason about local confluence, we will distinguish two types of pairs of transformations - those that are parallelly dependent and those that are parallelly independent. The reason for this is that two parallelly independent transformations are always interchangeable and can be applied in any order, whereas pairs that are dependent may be a source of true nondeterminism.

Definition 3.6 (Parallel independence). Two direct transformations \( G \Rightarrow r_1, g_1 \) \( H_1 \) and \( G \Rightarrow r_2, g_2 \) \( H_2 \) are parallel independent if the second derivation can be applied to \( H_1 \) and vice versa, i.e.

\[
 g_1(L_1) \cap g_2(L_2) \subseteq g_1(K_1) \cap g_2(K_2)
\]

where \( g_1 \) and \( g_2 \) are the matches of the rules in \( G \). An equivalent definition [10, Fact 3.18] is to say the pair is independent iff exist morphisms

\[
h_{12} : L_1 \to D_2 \text{ such that } f_2 \circ h_{12} = g_1
\]

and

\[
h_{21} : L_2 \to D_1 \text{ such that } f_1 \circ h_{21} = g_2
\]

A pair of transformations is parallel dependent if it is not parallel independent.

It has been shown that two parallelly independent transformations can be applied in any order. This is the famous Local Church-Rosser Theorem - rule \( r_2 \) can be applied to the result of the first transformation \( H_1 \) via the same match \( g_2 \) (and vice versa).

Theorem 3.7 (Local Church-Rosser Theorem). Given two parallelly independent transformations \( G \Rightarrow r_1, g_1 \) \( H_1 \) and \( G \Rightarrow r_2, g_2 \) \( H_2 \), there is a graph \( G' \) together with transformations \( H_1 \Rightarrow r_2, g_2' \) \( G' \) and \( H_1 \Rightarrow r_1, g_1' \) \( G' \) such that the diagram below commutes [see [10, Section 3.3]].

A parallel dependent pair of derivations is also called a conflict. Conflicts can be characterized depending on the model - when dealing with "plain" (unlabelled) graphs and rules, conflicts are called delete-use because one of the rules in the pair deletes something which the other needs. In the presence of attribution (labels) and application conditions, there can be more types of conflicts such as forbid-produce.
3.2 Critical Pairs

Proving a transformation system is confluent may be an impossible task just by looking at the definition - joinability must hold for all graphs $G$ and all derivable graphs $H_1$ and $H_2$ from $G$. The presence of conflicts makes the task even more difficult as the Church-Rosser theorem cannot be applied, and the resulting system of rules may not be confluent.

In order to check confluence in finite time, the potentially infinite set of conflicting pairs has to be reduced to a finite set of representatives. This is the aim of constructing critical pairs, pairs of dependent transformations in minimal context.

Definition 3.8 (Critical pair). A pair of direct derivations $T_1 \Leftarrow r_1, g_1 S r_2, g_2 \Rightarrow T_2$ is a critical pair if:

1. The graph $S$ is an overlapping of the left-hand sides of the two rules, i.e. minimal
   \[ S = g_1(L_2) \cup g_2(L_2) \]

2. The derivations are not parallely independent
   \[ g_1(L_1) \cap g_2(L_2) \nsubseteq g_1(K_1) \cap g_2(K_2) \]

3. If $r_1 = r_2$, then $g_1 \neq g_2$ must hold

The minimality condition means that the matches $g_1$ and $g_2$ are jointly surjective and that each item in $G$ has a pre-image in $L_1$ or $L_2$. In other words, $G$ does not contain unnecessary context. The dependency condition ensures that the transformations are not directly applicable to the result graphs $H_1$ or $H_2$. The extra condition of different matches ensures that $H_1$ is not trivially isomorphic to $H_2$.

Example 3.9 (Example Critical Pair). Consider the rule $r_1$ below.

\[
\begin{array}{c}
\text{1} \\
\text{2} \\
\end{array} \xleftarrow{r_1} \begin{array}{c}
\text{1} \\
\text{2} \\
\end{array}
\]

The following diagram represents a critical pair:

\[
\begin{array}{c}
\text{1} & \text{2} & r_1, g_1 \Leftarrow r_2, g_2 \Rightarrow \\
\text{1} & \text{2} & \text{1} & \text{2}
\end{array}
\]

with matches $g_1 = L \rightarrow S : (1 \rightarrow 1, 2 \rightarrow 2)$ and $g_2 = L \rightarrow S : (1 \rightarrow 2, 2 \rightarrow 1)$ respectively. $S$ is minimal because it contains no extra context that is unused by the matches, i.e. $g_1$ and $g_2$ are jointly surjective. The pair of derivations is a conflict because both delete the proper edges of $S$ and the rules cannot be applied to the results $T_1$ and $T_2$ at all. We also have that $g_1 \neq g_2$, which is needed since the rules involved are the same rule.

It can be shown that each pair of parallely dependent transformation is an embedding of a critical pair.
Lemma 3.10 (Completeness of critical pairs). For each pair of parallel dependent direct transformations \(H_1 \leftarrow r_{1,g_1}^1 G r_{2,g_2}^2 \Rightarrow H_2\), there is a critical pair \(T_1 \leftarrow r_{1,g_1}^1 S r_{2,g_2}^2 \Rightarrow T_2\) such that \(S\) is a subgraph of \(G\).

\[
\begin{array}{ccc}
T_1 & \xleftarrow{r_{1,g_1}^1} & S & \xrightarrow{r_{2,g_2}^2} & T_2 \\
\downarrow & & m & & \downarrow \\
H_1 & \xleftarrow{r_{1,g_1}^1} & G & \xrightarrow{r_{2,g_2}^2} & H_2
\end{array}
\]

Both squares above are extension diagrams in the sense of subsection 2.3.

Apart from reasoning about conflicts, critical pairs have another important property - in rewrite systems where the rules consist of finite graphs (such as in graph transformation) the set of critical pairs is finite because the left-hand sides of two rules can be overlapped in only finitely many ways. This is in contrast to the number of pairs of conflicting derivations which is infinite in general.

If the set of critical pairs is empty, then the above lemma already implies local confluence of the graph transformation system because all pairs of derivations will be parallely independent. Otherwise, in order to show local confluence, it is sufficient to examine all critical pairs and determine if they are strongly joinable. Strong joinability means that the critical pair is

- joinable - exists graph \(S'\) together with transformations \(T_1 \Rightarrow S'\) and \(T_2 \Rightarrow S'\)
- strictness - the largest subgraph \(N\) of \(S\) that is preserved by both steps in the critical pair is preserved by the joining derivations \(T_1 \Rightarrow S'\) and \(T_2 \Rightarrow S'\).

Joinability without the strictness condition has been shown to not be enough for local confluence [45] because the isomorphism between the two resulting graphs from the joining derivations may be destroyed when the critical pair is embedded into context.

In Example 3.9, the critical pair is joinable because \(T_1 \cong T_2\) with isomorphism \(iso = T_1 \rightarrow T_2 : (1 \rightarrow 2, 2 \rightarrow 1)\). However, it is not strongly joinable because the isomorphism makes the joining derivations not commute on both nodes 1 and 2. It is easy to see a counter example to confluence by embedding the critical pair into larger context by adding an extra edge:

Both \(H_1\) and \(H_2\) are non-isomorphic normal forms of \(G\). Therefore, the rule \(r_1\) does not represent a confluent graph transformation system.

Strong joinability allows us to state the so-called Critical Pair Lemma.

Theorem 3.11 (Local Confluence Theorem, Critical Pair Lemma). A graph transformation system is locally confluent if all its critical pairs are strongly joinable.
Critical Pair Construction Algorithm

The algorithm for computing the set of all critical pairs can be derived from the definition \[35, 36\]. First, we need to compute all jointly surjective matches for the two rules, i.e. suitable gluings of the left-hand sides. Second, we need to analyse if the transformations given by the two rules and computed matches are indeed parallel dependent. If that is the case, then that is indeed a critical pair.

\[ \text{input} : \text{Two rules } r_1 = \langle L_1 \leftarrow K_1 \rightarrow R_1 \rangle \text{ and } r_2 = \langle L_2 \leftarrow K_2 \rightarrow R_2 \rangle \]

\[ \text{output} : \text{A set of critical pairs } CP \]

1. \( CP \leftarrow \emptyset \)
2. Compute all jointly surjective matches \((g_1 : L_1 \rightarrow S, g_2 : L_2 \rightarrow S)\) of \( r_1 \) and \( r_2 \)
3. \( \text{foreach} \ (g_1, g_2) \ \text{do} \)
4. \( \text{if } r_1 = r_2 \text{ and } g_1 = g_2 \ \text{then skip;} \)
5. \( \text{Compute } t_1 = (S \xrightarrow{g_1} T_1) \text{ and } t_2 = (S \xrightarrow{g_2} T_2) \)
6. \( \text{if } t_1 \text{ and } t_2 \text{ are parallelly dependent then} \)
7. \( CP \leftarrow CP \cup \{(t_1, t_2)\} \)
8. \( \text{end} \)
9. \( \text{end} \)

Algorithm 1: Simple algorithm for computing all critical pairs

The most complex part of the algorithm is the computation of the jointly-surjective matches since their number will grow very large if the left-hand sides of the two rules are large \[54\, \text{Section 7.2}\]. The problem is in fact from set theory and is defined inductively - let \( V \) be the disjoint union of two sets of vertices \( V_1 \) and \( V_2 \), and call \((X, x : V \rightarrow X)\) an overlapping of \( V \) where \( x \) is a total surjective function:

- **Base case** If \( V = \{1\} \) there is only one overlapping, depicted in Figure 7 (a) \[\text{3}\]. For \( V = \{1, 2\} \) there are two possibilities - either the nodes are separate or are merged together as in Figure 7 (b)

- **Inductive case** - pick a vertex \( v \in V \) and compute the overlappings for \( V' = V \setminus v \).

For every such overlapping \((X', x : V' \rightarrow X')\) create the overlapping \((X, x : V \rightarrow X)\) in the following way:

- Take \( X = X' \) and merge \( v \) with one of the existing vertices, i.e. define the mapping \( x \) such that \( x(v) \) is in \( X' \) and equal to \( x' \) on all other vertices. (There are \( |X| \) ways to do this)

- Take \( X = X' \cup \{v\} \) and add \( v \) disjointly - \( x(v) = v \) and \( x = x' \) for all other vertices. (There is only one way to do this)

In Figure 7(c) we can see the overlappings for the case when \( V = \{1, 2, 3\} \). Let \( v = 3 \), then \( V' = V \setminus \{3\} = \{1, 2\} \) for which we already have the overlappings in Figure 7(b). The first overlapping in (b) has only one vertex so we can add node 3 in two ways - either merge it with node 1,2 or add it separately. This is depicted in the first two boxes of Figure 7(c). For the right overlapping of \( V' \) in (b), three overlappings can be created - merge node 3 with either of the nodes 1 and 2, or add it disjointly. This results in the later 3 overlappings of Figure 7(c).

\[3\] Hence, we can consider vertices and edges separately

\[4\] The node ids represent the mapping between \( V \) and \( X \) so it is left implicit
Figure 7: Computing jointly surjective overlappings

The number of overlappings is bounded by the $n$-th Bell number

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

where $v(n)$ is the total number of possible overlappings (injective and non-injective) of $n$ vertices.

Note that the algorithm can be easily modified to consider only injective matches - in the inductive subcase 1, merge the node $v$ only with nodes from the opposite rule. This decreases the number of overlappings. In the case where $V_1 = \{1, 2\}$ and $V_2 = \{3\}$ Figure 7(c) will have only the blue overlappings as answer because in the red overlappings nodes 1 and 2 are merged but originate from the same vertex set.

Lambers et al. [35] present two possible optimizations to the above algorithm for computing critical pairs. First, if both rules $r_1$ and $r_2$ are non-deleting, then they will always be parallely independent. A rule $r = \langle L \xleftarrow{l} K \rightarrow R \rangle$ is non-deleting if the morphism $l : K \rightarrow L$ is an isomorphism. There are different ways of checking this, and the authors prefer the construction of the context graph $C$ of $l$ (see Definition 2.18).

The second optimizations involves a deleting and a non-deleting rule. Instead of computing all possible overlappings, in this case it is possible to compute only those ones which lead to a critical pair. This is achieved by requiring that an element that is deleted by the deleting rule to be overlapped with an item of the non-deleting rule. The authors define certain conditions under which the pair of rules are guaranteed to be in conflict. In practice, this involves constructing a pullback object of the context of the deleting rule and the left-hand side of the non-deleting rule (and hence reusing the computation of the first optimization).

In a further paper [36], the authors define essential critical pairs, which are a subset of all critical pairs, and are complete in the sense that all critical pairs are strongly joinable if and only if all essential critical pairs are strongly joinable. However, the set of essential critical pairs is not significantly smaller than the set of all critical pairs.

Ruud Welling [54, Section 6.2] shows further improvements by attempting to determine if the strong joinability of critical pair $a$ implies the strong joinability of critical pair $b$. However, they only consider non-parallel edges, and their results are not applicable to our setting.
3.3 Confluence and Attribution

Confluence in its basic sense is *structural* in the sense that the graphs considered are unlabelled (Remark [1]). However, in many cases where graph transformation is used as a modelling technique, the domain is *attributed* (*labelled*) graphs that represent diagrams with textual, numerical, semantic annotations.

In their seminal paper, Heckel et al. [28] develop the theory of critical pairs to the setting of attribution. The paper introduces typed, vertex-attributed graph transformation systems and establish a critical pair lemma[5]. Their running example considers a translation between simple UML state-charts into CSP for automated verification via CSP model checker, and this process needs to be functional.

Attributes in this setting are represented as algebraic terms over an term algebra $T_{\Sigma}(X)$ [4, Chapter 3]. Attributed graphs are bipartite - they contain a *graph* part $G$ together with an *data* part $A$. Node and edge attributes are represented by as a relation between nodes and edges from the graph part to the nodes in the data part. Graphs will be usually infinite due to the data part containing all possible values for each type, e.g. if the data type $\mathbb{N}$ of natural numbers then each $n \in \mathbb{N}$ will be a separate data vertex. In practice, the data part is kept constant during computation, and unneeded values are not represented in the current state.

The main focus of the paper is how to decide attribution in the minimal graph $S$ of the critical pair. For the structure of $K$, the normal construction is used as presented in the previous section. This produces two jointly surjective matches $g_1$ and $g_2$ for the graph parts of the critical pair. For the data part of $K$, the minimality condition in the definition of critical pairs generalizes the idea of a most general unifier - a substitution $\sigma : X \rightarrow T_{\Sigma}(X)$ with as little instantiation of variables as needed to equate two terms.

In the simplest case, the attribute for a vertex $v$ in $K$ is obtained by computing the most general unifier of all pre-images of $v$ under $g_1$ and $g_2$. If a mgu does not exist, then there is no critical pair for that given pair of matches $(g_1, g_2)$. Otherwise, the mgu is used for the attribution of $v$, namely $\sigma(t)$ where $t$ is the attribute of the pre-image of $v$ (any of them).

For example, if $v$ is attributed with the terms $\text{add}(0, x)$ and $\text{add}(y, s(0))$ in $L_1$ and $L_2$ respectively[6], then the most-general unifier is $\sigma = \langle y \rightarrow 0, x \rightarrow s(0) \rangle$, and the attribute of $v$ is $\sigma(0 + x) = \text{add}(0, s(0))$.

\[ \begin{array}{c}
L_1 \\
\text{add}(0, x) \\
\downarrow g_1 \\
S \\
\downarrow g_2 \\
\text{add}(0, s(0))
\end{array} \begin{array}{c}
L_2 \\
\text{add}(y, s(0)) \\
\downarrow o_1 \\
G \\
\downarrow m \\
1
\end{array} \]

---

5. Typed graphs can be seen as a generalization of labelled graphs [10, Fact 2.9].

6. Here $\text{add}$ is the addition function of arity 2, $x$ and $y$ are variables, and $s(x)$ is the successor function over the natural numbers
Graph nodes in the above diagrams are represented as black circles, data nodes are rectangles labelled with terms, and attribution is represented by dashed edges. The left-hand sides of some pair of rules are overlayed in the minimal graph $S$. The only node in $S$ is attributed by the unified term as explained above. The complete rules are not shown but this overlay can easily be made critical by setting $K_1$ or $K_2$ to the empty graph $\emptyset$ (delete-use conflict) or changing the attribution in $R_1$ or $R_2$ (change-attribute conflict).

The resulting critical pair represents a pair of conflicting transformations in which the graph $G$ is attributed with the ground term $1$. If the critical pair is later proven to be strongly joinable, then the transformations from $G$ will be strongly joinable too.

However, the above construction has been shown to be incomplete in [10, p.198]! If you consider a graph $G'$ isomorphic to $G$, but attributed with the term e.g. $2$, then there are morphisms $o_1$ and $o_2$ by setting $\langle x \to 2, y \to 1 \rangle$, but there is no way to extend the graph $S$ to $G'$. Also, there are no other critical pairs with the same structural conflict because most general unifiers are unique should they exist.

The reason for the above negative result is that syntactic unification is not enough for arbitrary term algebras $T_2(X)$. For a given structural critical pair, it may be the case that there are an infinite number of unique attributions for the minimal overlap $S$, which in turns implies an infinite number of critical pairs.

[10, Section 9.4] consider the special case where the left-hand sides of the rules are attributed only by variables and prove that their simple attribution of the critical pair is complete by the so-called epi-mono $E_2\cdot M_2$ pair factorization.

Golas et al. [20] further extend the theory by allowing negative application conditions (NACs) and inheritance on top of attribution and typing. Attribution of left-hand sides of rules is only by variables or terms without variables, but this condition is not justified explicitly nor is an algorithm shown how to attribute the critical overlay. The paper however further restricts the set of critical pairs to abstract critical pairs, a representative of all potential critical pairs, which is shown to be complete given the mentioned restrictions.
3.4 Confluence and Application Conditions

Graph conditions allow us to restrict the application of rules by equipping them with an extra graph that must or must not exist in the match of a rule. First they emerged as a general concept of restriction [12], and were extended to negative application conditions [25] and nested application conditions [22]. The dangling condition in the DPO approach can be thought as an example of a negative application condition (and expressed as one).

An application condition over a graph $L$ is intuitively represented by a boolean formula $\exists a$ (or $\forall a$) and is defined in terms by a morphism $a : P \rightarrow C$. A morphism $m : L \rightarrow G$ over $L$ satisfies the application condition if $C$ does (not) occur in $G$.

**Definition 3.12** (Application condition). An application condition $ac_L$ over a graph $L$ is inductively defined as follows:

- $true$ is an application condition over $P$
- For every morphism $a : P \rightarrow C$ and every application condition $ac_C$ over $C$, $\exists (a, ac_C)$ is an application condition over $P$
- For every application conditions $ac_1$ and $ac_2$ over $P$, $\neg ac_1$ and $ac_1 \land ac_2$ are application conditions over $P$.

**Definition 3.13** (Satisfiability of an application condition). Satisfaction of application conditions is defined inductively:

- Every morphism satisfies $true$
- A morphism $g : L \rightarrow G$ satisfies an application condition $\exists (a : L \rightarrow C, ac_C)$, denoted by $g \models ac_L$, if there exists an injective morphism $q : C \rightarrow G$ such that $q \circ a = g$ and $q \models ac_C$

$$
\begin{array}{c}
L \xrightarrow{a} C \\
g \xrightarrow{\exists q} G \\
q \models ac_C
\end{array}
$$

- A morphism $g : L \rightarrow G$ satisfies an application condition $\neg c$ if $g$ does not satisfy $c$ and satisfies $ac_1 \land ac_2$ if it satisfies both $ac_1$ and $ac_2$

Application conditions add to the expressive power of graph transformation. However, their use poses additional problems for constructing critical pairs - a pair of transformations may be independent in the sense of Definition 3.6, but one of the rules creates a structure that is forbidden by the other rule’s application condition. A more elaborate version of independence and confluence was developed in [34, 33].

**Definition 3.14** (Parallel independence with NACs). Two direct transformations $G \Rightarrow_{r_1,g_1} H_1$ with NAC $ac_{L_1} : L_1 \rightarrow N_1$ and $G \Rightarrow_{r_2,g_2} H_2$ with NAC $ac_{L_2} : L_2 \rightarrow N_2$ are parallel independent iff

- $\exists h_{12} : L_1 \rightarrow D_2$ such that $f_2 \circ h_{12} = g_1$ and $g_2 \circ h_{12} \models ac_{L_1}$ and
- $\exists h_{21} : L_2 \rightarrow D_1$ such that $f_1 \circ h_{21} = g_2$ and $g_1 \circ h_{21} \models ac_{L_2}$
As already mentioned, for each morphism \( h \) breaking the first condition induces a delete-use conflict and breaking the second condition induces a produce-forbid conflict.

Now that there are extra sources of conflicts, this changes the basic algorithm for critical pair construction in the following way - if the computed pair of transformations \( t_1 \) and \( t_2 \) are not in delete-use conflict, then check if either is in produce-forbid conflict, thus adding an extra layer of complexity.

Ehrig et al. [18] consider confluence of rules equipped with arbitrary nested application conditions. Their results rely on shifting conditions over morphisms and rules - in the diagram above, instead of having two conditions over \( L_1 \) and \( L_2 \), two other conditions \( ac_G \) and \( ac_G^* \) over \( G \) are constructed that ensure any extension of the critical pair by a morphism \( m : G \to G' \) is also parallelly dependent.

However, since \( ac_G \) and \( ac_G^* \) are also arbitrarily nested, constructing a set of critical pairs in that setting turns out to be an undecidable problem due to nesting being equivalent to first-order graph formulas. Showing strong joinability in this case turns out to also be undecidable.

### 3.5 Applications of Confluence

Confluence has been used in various fields of computer science to reason about properties of computation.

Reduction systems are a powerful way of finitely representing an infinite set of graphs with some special property [2, 7]. They consist of a set of rules and a finite number of accepting graphs. A reduction system defines a language of graphs \( L \) such that every normal form of a graph \( G \) of \( L \) is an accepting graph. Classical examples of graph reduction consists trees, series-parallel graphs, flowcharts. However, recognition of such languages can be a costly process since all normal forms are to be computed. In the case when the reduction rules are confluent, this becomes much more efficient as confluence implies uniqueness of normal forms.

In [10, Ch. 14], the authors use graph transformation to implement the model transformation between UML Statecharts and Petri Nets. One of the main properties of model transformation is its functional behaviour, so the underlying graph transformation system has to be proven locally confluent and terminating. The tool AGG is used, and the graph transformation is in the context of typed attributed graphs. The source and target languages are given as typed graphs, which is not a restriction as any structure such as a model can be represented by typed attributed graphs. The supporting analysis of critical pair construction in AGG is used to reason about local confluence and hence the functional behaviour of the transformation.

Conflicts between requirements of different stakeholders may be difficult and expensive to resolve during software development. Hausmann et al. [27] propose a method for detecting such conflicts as early as possible by means of graph transformation. The authors use representations of UML use case, activity and collaboration diagrams as typed attributed graphs and transformation rules between such graphs. The resulting critical pairs analysis can be used to annotate use cases and activity diagrams with extra information regarding conflicts.
and dependencies and to trigger a re-iteration of the requirements model to eliminate the undesired effects. Note that confluence is not strictly required, only the detection of potential conflicts between use cases. Furthermore, not all such conflicts represent an error - if a requirements analyst decides that two use cases or activities are meant to happen alternatively, the conflict simply reflects this requirement at the object level.

Software refactoring is a common technique for improving the structure of object-oriented code while preserving external behaviour \[39\]. Existing tools only offer help with the automated application of such refactorings. The developer has to choose interactively which refactorings he would like to apply, and use a refactoring tool to apply these refactorings. Mens et al. \[40\] propose a method for the automated detection of implicit conflicts and dependencies between refactorings. Their method is based on critical pair analysis which would allow a tool to suggest refactorings that are more appropriate in a given context and offer explanation why. The authors model refactorings as rules in the setting of typed attributed graphs, and use AGG to compute all possible conflicts between them. Then, based on the number of dependencies, some rules are suggested before others.

Visual languages are a programming paradigm where users create programs that have spatial relationships between elements rather than being textual. Their parsing can be described as a graph transformation system. Bottoni et al. \[9\] propose an efficient parsing algorithm such that critical pair analysis is used to delay decision between conflicting rule applications as far as possible. This means applying non-conflicting rules first and reducing the graph as much as possible before creating decision points for the backtracking.

### 3.6 Summary

Confluence is an important property when trying to argue that a computation in a non-deterministic model of computation exhibits deterministic behaviour. We have presented the main notions of parallel independence and conflicts, and shown how they allow us to automate proofs of confluence. The extensions to static confluence checking have been presented, namely how attribution and application conditions affect the types of conflicts that may arise during computation. Some applications of confluence have been presented.
4 Graph programming with GP

The computational model of graph transformation has been extended in several ways. One of them was to allow the explicit control of rule applications. Habel and Plump [23] showed that when equipped with sequential composition and as-long-as-possible iteration constructs, graph transformation becomes computationally complete in the sense of Turing. This led to the design of GP, a minimal complete language for graph transformation that allows high-level problem solving on graphs [51], on which we concentrate for the rest of this document.

GP incorporates many of the extensions to graph transformation as explained in the introduction. First, it allows for labels to be typed over integer, string and list expressions. Second, it lifts graph transformation rules to rule schemata that are blueprints for rules at run-time. Third, it allows the composition of rule schemata into graph programs using the above-mentioned constructs of sequential composition, as-long-as-possible iteration and non-deterministic choice. Forth, it allows for the specification of textual application conditions on rules that restrict/enable the rule application.

The rest of the section is organized as follows - first we define GP’s building blocks (conditional) rule schemata and what graphs they operate on. Then, we give the abstract syntax of graph programs and give example GP programs. Finally, we briefly outline the operational semantics of the core and derived commands.

4.1 GP labels and rule schemata

In GP, there is a difference between the labels in program rules and labels of graphs on which the rules operate. The first are labelled over integer, string and list expressions and may contain variables of those types. The latter are labelled over integers, strings and lists of integers and strings and and are provided as input to programs. Both kinds of labels can also be marked.

Let $\mathbb{Z}$ be the set of integers, Char be a finite set of characters and $B = \{\text{true}, \text{false}\}$.

Definition 4.1 (Host graph label alphabet). The label alphabet $\mathcal{L} = (\mathbb{Z} \cup \text{Char}^*)^* \times B$ is called the label alphabet for host graphs, where each label consists of a list of atoms (integers or strings) and a mark (true or false).

Sometimes the mark component is omitted, e.g. writing $l_G(v) = 25$ means that label of node $v$ in graph $G$ has list component 25 with an unspecified mark component.

Labels in GP rule schemata comprise of an expression and a mark, as opposed to labels in host graphs. Each expression is built up from constant symbols in $\mathbb{Z}$ and Char, variables, and function symbols. Variables are typed, each drawn from a disjoint set of variables: IVar, SVar, AVar and LVar respectively for denoting integer, string, atom and list variables. Their union is denoted as Var. The subtype hierarchy for expressions is presented in Figure 8.

```
list          (Z ∪ Char*)*
atom          Z ∪ Char*
string        Z ∪ Char*       Char
int           Z ∪ Char*       Char
str           Z ∪ Char*       Char
```

Figure 8: Subtype hierarchy for expressions
The grammar presented in Figure 9 defines how expressions are built using basic components. It is assumed that Node is the set of node identifiers occurring in a GP rule schema, which must be the same for the left and the right graph. The intended meaning of symbols is their usual meaning, like ‘+’ for addition, ‘.’ for string concatenation and ‘:’ for list concatenation.

RSLabel ::= List Mark
Mark ::= true | false
List ::= empty | Atom | LVar | List ‘:’ List
Atom ::= Integer | String | AVar
String ::= “” {Char} “” | SVar | String ‘.’ String
Integer ::= Digit {Digit} | IVar | ‘-’ Integer | Integer ArithOp Integer | (indeg | outdeg) ‘(’ Node ‘)’
ArithOp ::= ‘+’ | ‘-’ | ‘*’ | ‘/’

Figure 9: Abstract syntax of rule schema labels

Let RS denote the alphabet of all expressions generated by the syntactic class RSLabel of the above grammar, then \( G(RS) \) is the set of all symbolic graphs over RS and \( G(RS_\perp) \) is the set of all partially labelled symbolic graphs over \( RS_\perp \).

Rule schemata are rules in the sense of Definition 2.7, but with \( L \) and \( R \) being labelled with expressions and \( K \) consisting of unlabelled nodes only. They represent possibly infinite sets of rules over \( G(L) \), obtained by assigning values to variables and evaluating expressions. Because this is done during execution time and is determined by graph matching, is required that expressions in the left graph of a rule schema must have a simple shape.

**Definition 4.2 (Simple list).** An expression \( e \in \text{List} \) is simple if
1. \( e \) contains no arithmetic operators
2. \( e \) contains at most one occurrence of a list variable
3. each occurrence of a string expression in \( e \) contains at most one occurrence of a string variable

For example, if \( x, y \in \text{LVar}, a \in \text{AVar}, s, t \in \text{SVar} \) are variables, then the expressions \( a:x \) and "no", \( s:y:t \) are simple whereas \( x:y \) and \( s.t \) are not simple.

**Definition 4.3 (Rule schema).** A rule schema \( \langle L \leftarrow K \rightarrow R \rangle \) consists of two inclusions \( K \rightarrow L \) and \( K \rightarrow R \) such that \( L, R \) are graphs in \( G(RS) \) and \( K \) is a graph in \( G(RS_\perp) \) consisting of unlabelled nodes only. It is required that all list expressions in \( L \) are simple and that all variables occurring in \( R \) also occur in \( L \) which ensures that for a given match applying \( r \) produces a unique graph (up to isomorphism).

Rule schemata are instantiated by evaluating their labels according to some assignment \( \alpha : \text{Var} \rightarrow \mathcal{L} \).

**Definition 4.4 (Assignment).** An assignment is a family of mappings \( \alpha = (\alpha_X)_{X \in \{I,S,A,L\}} \) :
- \( \alpha_I : \text{IVar} \rightarrow \mathbb{Z} \)
- \( \alpha_S : \text{SVar} \rightarrow \text{Char}^* \)
- \( \alpha_A : \text{AVar} \rightarrow \mathbb{Z} \cup \text{Char}^* \)

\( \text{Var} = \text{LVar} \cup \text{AVar} \cup \text{SVar} \cup \text{IVar} \)
traverse(x,y,z:list; a,b:int)

\begin{figure}
\centering
\begin{tikzpicture}
  \node (x) at (0,0) [circle,draw] {$x$};
  \node (y) at (1,0) [circle,draw] {$y$};
  \node (z) at (2,0) [circle,draw] {$z$};
  \node (a) at (0,-1) [circle,draw] {$a$};
  \node (b) at (1,-1) [circle,draw] {$b$};
  \draw [->] (x) to node [above] {$a$} (y);
  \draw [->] (y) to node [above] {$b$} (z);
  \draw [->] (a) to node [below] {$a+b$} (b);
  \draw [->] (x) to node [below] {$1$} (a);
  \draw [->] (y) to node [below] {$2$} (b);
  \draw [->] (z) to node [below] {$3$} (b);
\end{tikzpicture}
\caption{Example rule schema}
\end{figure}

\begin{itemize}
\item $\alpha_L : \text{LVar} \rightarrow (\text{Z} \cup \text{Char}^*)^*$
\end{itemize}

We call $r^{g,\alpha}_g = \langle L^{g,\alpha} \leftarrow K \rightarrow R^{g,\alpha}_g \rangle$ the \textit{instance} of $r$ with respect to $g$ and $\alpha$, where $L^{g,\alpha}$ and $R^{g,\alpha}_g$ are obtained from $L$ and $R$ by replacing each label $l$ with $l^{g,\alpha}$ which involves evaluation of expressions. See \cite{15} for a complete example. Note that $r^{g,\alpha}_g$ is a rule over $L$.

We can restrict the application of a rule schema by a textual \textit{application condition}. This extension is inspired by application conditions in graph transformation that allow for the specification of a structure that must or must not exist for the rule to be applied. Application conditions have been subject to extensive research as their existence has implications for static properties of transformations like confluence (\cite{18,19,17}, subsection 3.4).

In GP, rule schemata can be equipped with a condition as specified by the grammar in Figure 11. Here $\text{Node}$ denotes a set of node identifiers occurring in the rule schema.

\begin{figure}
\centering
\begin{grammar}
  \text{Condition} ::= \text{Type} \ ('\text{List} ') | \text{List} \ ('\text{=} | ' \text{\textbackslash=} ') \ \text{List} \\
  \text{Type} ::= \text{int} | \text{string} | \text{atom} \\
  \text{IntRel} ::= '<' | '<=' | '>' | '>='
\end{grammar}
\caption{Abstract syntax of rule schema conditions}
\end{figure}

The Type predicate expresses that a list expression must evaluate to a certain concrete type. The List comparison allows for comparing the values of two list expressions, and similarly for integer comparison. The \textit{edge} predicate gives a structural restriction on existence of an edge between two nodes in the matched graph (the third optional parameter specifies the label of that edge). This is more useful when negated, forbidding the rule application in contexts where edges (with a particular label) exist outside of the match.

\textbf{Definition 4.5} (Conditional rule schema). A \textit{conditional rule schema} is a pair \langle $r$, $\Gamma$ \rangle where $r$ is a rule schema and $\Gamma$ is a condition generated by the Condition grammar in Figure 11 such that all variables occurring in $\Gamma$ also occur in the left-hand side $L$ of $r$.

Figure 10 shows a conditional rule schema \texttt{traverse} that links two indirectly connected nodes with a direct edge. The rule schema identifier is written above the two graphs along with any variable declarations. Multiple variables can have the same type, separated by a semi-colon. The condition is written below the two graphs following the keyword \texttt{where}. Here it ensures that in the context of the match there is not an already edge connecting the two nodes 1 and 2 and hence the rule cannot be matched on the same graph in the result.

From now on we will refer to conditional rule schemata as rule schemata and be explicit in the cases when the condition matters.
Definition 4.6 (Rule schema application). The application of a rule schema \( r = (L \leftarrow K \rightarrow R, \Gamma) \) to a host graph \( G \in \mathcal{G}(\mathcal{L}) \), denoted by \( G \Rightarrow_{r,g} H \) (or just \( G \Rightarrow_r H \)) informally is done as follows:

1. Match \( L \) with a subgraph of \( G \), ignoring labels, by means of a premorphism \( g : L \rightarrow G \)
2. Check whether there is an assignment \( \alpha \) of variables to values such that after evaluating the simple expressions in \( L \), \( g \) is label-preserving
3. Check whether the rule schema condition evaluates to \text{true} under the given premorphism \( g \) and assignment \( \alpha \)
4. Apply the rule \( r^{g,\alpha} \), obtained from \( r \) by evaluating all expressions in \( L \) and \( R \), to \( G \) according to Definition 2.10 to obtain graph \( H \).

We write \( G \Rightarrow_{r,g,\alpha,g} H \) to denote the application of \( r^{g,\alpha} \) to \( G \) with match \( g \). Given a set \( \mathcal{R} \) of rule schemata, we write \( G \Rightarrow_{\mathcal{R}} H \) if \( G \Rightarrow_r H \) for some rule schema \( r \) in \( \mathcal{R} \).

Example 4.7 (Example direct derivation). Figure 12 shows the rule schema \texttt{traverse} being applied to a labelled graph \( G \). The application starts by finding an injective premorphism \( g : L_{\text{traverse}} \rightarrow G \) - there are two of them:

\[ g_1 = \langle 1 \mapsto 5, 2 \mapsto 6, 3 \mapsto 7 \rangle \]
\[ g_2 = \langle 1 \mapsto 5, 2 \mapsto 8, 3 \mapsto 7 \rangle \]

If \( g_2 \) is chosen as a match, then the next step fails because there is no assignment of the integer variable \( a \) to the string ”a”. Therefore, the only match is \( g_1 \) and the corresponding assignment is

\[ \alpha = \langle x \mapsto 0:1, y \mapsto 2, z \mapsto 3, a \mapsto 5, b \mapsto 6 \rangle \]

The rule schema condition \texttt{not edge}(1,3) is checked next - it is \text{true} because \( g(1) \) and \( g(3) \) in \( G \) (respectively nodes 5 and 7) do not have an edge between them.

The derivation process can proceed by \textit{instantiating} \texttt{traverse} with \( \alpha \) and \( g \) which is shown in the middle of the figure. The structural information in \( g \) is not needed in this case, but it may be needed due to the \texttt{indegree}/\texttt{outdegree} operators existing in the GP Label syntax. Note that \( R^{\alpha,g} \) contains \textit{evaluated} labels instead of expressions, i.e. the value 11 instead of 5+6 for node 3 (denoted in the diagram by \( \alpha^* \)).

Once \texttt{traverse}^{\alpha,g} has been obtained, the process can proceed as a DPO direct derivation with relabelling (Definition 2.16). The resulting graph \( H \) is shown in the bottom right. Note that none of the nodes change their labels, hence the rule instance interface does not have unlabelled nodes (not explicitly shown).
traverse: \[ a \xrightarrow{1} x \xrightarrow{2} y \xrightarrow{3} z \] \Rightarrow \[ a \xrightarrow{1} x \xrightarrow{2} y \xrightarrow{3} z \]

\[ \alpha \downarrow \]

\[ \alpha \star \downarrow \]

traverse^{a,g}: \[ 0:1 \xrightarrow{5} 2 \xrightarrow{6} 3 \] \Rightarrow \[ 0:1 \xrightarrow{5} 2 \xrightarrow{6} 3 \]

\[ g \downarrow \]

\[ h \downarrow \]

\[ G: \]

\[ 0:1 \xrightarrow{5} 2 \xrightarrow{6} 3 \]

\[ 5 \xrightarrow{1} 6 \xrightarrow{2} 7 \]

\[ "a" \xrightarrow{8} \]

\[ g_V = \langle 1 \mapsto 5, 2 \mapsto 6, 3 \mapsto 7 \rangle \]

\[ \alpha = \langle x \mapsto 0:1, y \mapsto 2, z \mapsto 3, a \mapsto 5, b \mapsto 6 \rangle \]

Figure 12: Example rule schema application

4.2 Graph Programs

A graph program consists is a set of rule schemata declarations together with some program text (possibly organized in macros). As mentioned at the start of this section, the control constructs of sequential composition and as-long-as-possible iteration are enough for computational completeness. However, GP offers a few extra ones for usability purposes. The syntax of programs is in Figure 13.

A program is a list of declarations, which are either a Rule Schema declaration, a macro declaration, or a main program declaration. The main program signifies the program text to be executed and contains several Command clauses (generalized in Command Sequences). A command sequence is the sequential composition of commands:

- RuleSetCall - either a single rule schema or a non-deterministic choice of rule schemata. RuleID represents the set of rule identifiers, unique for each rule schema.
- if-then-else - a conditional branching construct; executes the then clause or the else clause depending on the result of the condition execution. See below for full semantic explanation
- try-then-else - another conditional branching construct; see below
- ComSq! - the looping construct; the body is executed as many times as possible
- ComSq or ComSq - non-deterministic choice of two programs


|

| Prog ::= Decl {Decl} | Decl ::= RuleDecl | MacroDecl | MainDecl |
| MacroDecl ::= MacroID '=' ComSq |
| MainDecl ::= main '=' ComSq |
| ComSq ::= Com {';' Com} |
| Com ::= RuleSetCall | MacroCall |
| RuleSetCall ::= RuleID | '{' [RuleID {',' RuleID} ] '} |
| MacroCall ::= MacroID |

Figure 13: Abstract syntax of graph programs

- **skip** - do nothing; equivalent to executing the empty rule which is always applicable
- **fail** - manually trigger program termination. Equivalent to executing the empty set of rule schemata {}

The **skip**, **fail** and **or** commands can be simulated by the others.

**Example 4.8** (Transitive closure). Consider the example GP program below that computes the transitive closure of an integer labelled graph. It uses the rule schema from [Figure 10].

```plaintext
main = traverse!

traverse(x,y,z:list; a,b:int)

where not edge(1,3)
```

Figure 14: Example GP program

The rule schema is applied as long as possible. At each step, two nodes that are indirectly connected are joined with a direct edge. The program can be shown to terminate and to be correct.
4.3 Operational Semantics

GP has formal semantics in the style of Plotkin’s structural operational semantics [44]. Each programming construct has several related inference rules that induce a transition relation → on configurations. A configuration represents the current state of computation and is either a graph (a proper result) or a command sequence and a graph (unfinished computation) or a special element fail (failure state).

Definition 4.9 (Transition relation). The small-step transition relation

\[ \rightarrow \subseteq \left( \text{ComSq} \times \mathcal{G}(\mathcal{L}) \right) \times \left( \left( \text{ComSq} \times \mathcal{G}(\mathcal{L}) \right) \cup \mathcal{G}(\mathcal{L}) \cup \{\text{fail}\} \right) \]

on configurations defines the single-step computation on graphs. The transitive and reflexive-transitive closures are written as →⁺ and →∗ respectively.

A configuration λ is said to be terminal if there is no configuration δ such that λ → δ.

Below are the inference rules for GP’s core and derived commands (“derived” because they can be defined by core commands). An inference rule has a premise and a conclusion, separated by a horizontal line. The rules contain (implicitly) universally quantified metavariables for command sequences and graphs - R stands for a call in RuleSetCall, C, P, P’, Q stand for command sequences in ComSq and G, H stand for graphs in G(\mathcal{L}).

Definition 4.10 (Semantic inference rules for core commands). Figure 15 defines the inference rules for the core GP commands. The notation \( G \not\Rightarrow_{\mathcal{R}} H \) expresses that for a graph \( G \in \mathcal{G}(\mathcal{L}) \), there is no graph \( H \) such that \( G \Rightarrow_{\mathcal{R}} H \).

Consider the rule [call1] - intuitively it reads “for all sets of conditional rule schemata \( \mathcal{R} \) and graphs \( G, H \in \mathcal{G}(\mathcal{L}) \), \( G \not\Rightarrow_{\mathcal{R}} H \) implies \( (\mathcal{R}, G) \to H \)”. Note the definition of the if-then-else - the alternative \( Q \) is executed if the conditional program \( C \) fails. This is the concept of negation as failure that comes from logic programming. Also, the programs \( P \) and \( Q \) are executed on the original graph \( G \), which allows for the hiding of “destructive” tests. The alternative try-then-else executes the consequent \( P \) the proper result of executing \( C \).

Failing executions of if-then-else, try-then-else and as-long-as-possible are notable from the point of view of nondeterminism - if the guard \( C \) (or lopp body \( P \) respectively) may result in both a proper result graph and a fail, then the choice of which alternative to take (resp. to continue looping or pass control back) is nondeterministic.

Figure 16 defines the inference rules for the derived commands of GP.

Similarly as above, the or command is nondeterministic on the sense that the evaluation may choose which of the two programs to execute. The meaning of the GP commands can be defined by the semantic function \([ ]\) which assigns to each program \( P \) the function \([P]\) that maps an input graph \( G \) to the set of all possible results of executing \( P \) on \( G \). The application of \([P]\) to \( G \) is denoted by \([P]G\). This set may also contain the special symbols fail and ⊥. The first indicates that the program can end in failure, and the second means that the program may diverge (does not terminate) or "gets stuck”.

Definition 4.11 (Divergence). A program \( P \) diverges from graph \( G \) if there is an infinite sequence

\[ (P,G) \to (P_1,G_1) \to (P_2,G_2) \to \ldots \]

Definition 4.12 (Getting stuck). A program \( P \) gets stuck from graph \( G \) if there is terminal configuration \( (Q,H) \) such that \( (P,Q) \to^+ (Q,H) \)
A program can get stuck in two situations:

- it contains an if-then-else (or try-then-else) such that the condition $C$ can diverge on some graph $G$ and can neither produce a proper result graph nor fail

- it contains a loop $P!$ such that the body $P$ contains the above property.

The evaluation of such programs gets stuck because none of the inference rules for the respective constructs are applicable.

**Definition 4.13** (Semantic function). The semantic function $\llbracket \cdot \rrbracket : \text{ComSq} \rightarrow (\mathcal{G}(\mathcal{L}) \rightarrow 2^{\mathcal{G}(\mathcal{L}) \cup \{\text{fail}^1, \bot\}})$ is defined by

$$\llbracket P \rrbracket G = \{X \in (\mathcal{G}(\mathcal{L}) \cup \{\text{fail}\}) \mid \langle P, G \rangle \rightarrow^+ X \} \cup \{\bot \mid P \text{ can diverge or get stuck from } G\}$$

**Definition 4.14** (Semantic equivalence). Two programs $P$ and $Q$ are semantically equivalent, denoted by $P \equiv Q$, if $\llbracket P \rrbracket = \llbracket Q \rrbracket$.

For example, the following equivalences can be proven:

- **skip** $\equiv$ **null**, where **null** is the empty rule schema $\emptyset \Rightarrow \emptyset$ that is always applicable
\[ \langle P \text{ or } Q, G \rangle \rightarrow \langle P, G \rangle \]  
\[ \langle P \text{ or } Q, G \rangle \rightarrow \langle Q, G \rangle \]

\[ \langle \text{skip}, G \rangle \rightarrow G \]  
\[ \langle \text{fail}, G \rangle \rightarrow \text{fail} \]

\[ \frac{\langle C, G \rangle \rightarrow^+ H}{\langle \text{if} \ C \text{ then } P, G \rangle \rightarrow \langle P, G \rangle} \]  
\[ \frac{\langle C, G \rangle \rightarrow^+ \text{fail}}{\langle \text{if} \ C \text{ then } P, G \rangle \rightarrow G} \]

\[ \frac{\langle C, G \rangle \rightarrow^+ H}{\langle \text{try} \ C \text{ then } P, G \rangle \rightarrow \langle P, H \rangle} \]  
\[ \frac{\langle C, G \rangle \rightarrow^+ \text{fail}}{\langle \text{try} \ C \text{ then } P, G \rangle \rightarrow G} \]

Figure 16: Inference rules for derived commands

- \( \text{fail} \equiv \{\} \), where \( \{\} \) is the empty set of rule schemata
- \( \text{if } C \text{ then } P \equiv \text{if } C \text{ then } P \text{ else null} \), for all programs \( C \) and \( P \)
- \( \text{try } C \text{ then } P \equiv \text{try } C \text{ then } P \text{ else null} \), for all programs \( C \) and \( P \)
- \( \text{try } C \text{ else } P \equiv \text{try } C \text{ then null else } P \), for all programs \( C \) and \( P \)

4.4 Summary

We have presented the intuition and theoretical background behind GP, a domain specific language for graph transformation and computation. Its main features allow for the intuitive specification of powerful graph programs without having to resort to low-level implementation concerns. Furthermore, the language has complete formal semantics that facilitate high-level reasoning about properties of graph computation.
5 Preliminary results

This section is mainly based on our paper accepted by the Fifth International Workshop on Graph Computation Models in York (GCM2014) [29].

5.1 A Unification algorithm for GP

5.1.1 Introduction

A common programming pattern in the graph programming language GP [49] is to apply a set of graph transformation rules as long as possible. To execute such a loop \(\{r_1, \ldots, r_n\}!\) on a host graph, in each iteration an applicable rule \(r_i\) is selected and applied. As rule selection and rule matching are nondeterministic, different graphs may result from the loop. Thus, if the programmer wants the loop to implement a function, a useful tool would be a static analysis that establishes or refutes functional behaviour.

The above loop is guaranteed to produce a unique result if the rule set \(\{r_1, \ldots, r_n\}\) is terminating and confluent. However, conventional confluence analysis via critical pairs [47] assumes rules with constant labels whereas GP employs rule schemata whose graphs are labelled with expressions.

We are not aware of algorithms that check confluence over non-trivial attribute algebras such as that of GP which includes list concatenation and Peano arithmetic. The problem is that the equational theory of an attribute algebra needs to be taken into account when constructing critical pairs and checking their joinability.

For example, [28] presents a method of constructing critical pairs in the case where the equational theory of the attribute algebra is represented by a convergent term rewriting system. The algorithm first computes normal forms of the attributes of overlayed nodes and subsequently constructs the most general unifier of the normal forms. This has been shown to be incomplete [10, p.198] in that the constructed set of critical pairs need not represent all possible conflicts and the resulting confluence analysis is not even sufficient. For, the most general unifier produces identical attributes—but it is necessary to find all substitutions that make attributes equivalent in the equational theory subsection 3.3.

As a simple example, consider the program in Figure 17 for calculating shortest distances. The program expects input graphs with non-negative integers as edge labels, and arbitrary lists as node labels. There must be a unique marked node (drawn shaded) whose shortest distance to each reachable node has to be calculated.

\[
\text{main = init; \{add, reduce\}!}
\]

\[
\text{init(x: list)} \quad \text{add(x,y: list;m,n: int)}
\]

\[
\begin{align*}
\text{x} & \Rightarrow \text{x:0} \\
\text{x;m} & \Rightarrow \text{y} \\
\text{x;m} & \Rightarrow \text{y:m+n}
\end{align*}
\]

\[
\text{reduce(x,y: list;m,n,p: int)}
\]

\[
\begin{align*}
\text{x;m} & \Rightarrow \text{y:p} \\
\text{x;m} & \Rightarrow \text{y:m+n}
\end{align*}
\]

where \(m + n < p\)

Figure 17: GP program for calculating shortest distances
The rule schemata \texttt{init} and \texttt{add} append distances to the labels of nodes that have not been visited before, while \texttt{reduce} decreases the distance of nodes that can be reached by a path that is shorter than the current distance.

To construct the conflicts of the rule schemata \texttt{add} and \texttt{reduce}, their left-hand sides are overlayed. For example, the structure of the left-hand graph of \texttt{reduce} can match the following structure in two different ways:

Consider a copy of \texttt{reduce} in which the variables have been renamed to $x', m'$, etc. To match \texttt{reduce} and its copy differently requires solving the system of equations \{x:m =? y':p', y:p =? x'm'}

\begin{align*}
\sigma &= \{x' \mapsto y, m' \mapsto p, y' \mapsto x, p' \mapsto m\}
\end{align*}

is a most general solution. It gives rise to the following critical pair:

In general though, equations can arise that have several independent solutions. For example, the equation $n : x =? y : 2$ (with $n$ of type \texttt{int} and $x, y$ of type \texttt{list}) has the minimal solutions

\begin{align*}
\sigma_1 &= \{x, y \mapsto \text{empty}, n \mapsto 2\} \quad \text{and} \quad \sigma_2 = \{x \mapsto z:2, y \mapsto n:z\}
\end{align*}

where \text{empty} represents the empty list and $z$ is a list variable.

The problem of attributing critical pairs is orthogonal to that of their construction, and yet crucial for producing a sufficient algorithm for proving confluence.

Seen algebraically, we need to solve equations modulo the associativity and unit laws

\begin{align*}
\text{AU} &= \{x : (y : z) = (x : y) : z, \text{empty} : x = x, x : \text{empty} = x\}.
\end{align*}

This problem is similar to word unification \cite{word_unification}, which attempts to solve equations modulo associativity. Solvability of word unification is decidable, albeit in PSPACE \cite{word_unification}, but there is not always a finite complete set of solutions. The same holds for AU-unification. Fortunately, GP’s syntax for left-hand sides of rule schemata forbids labels with more than one list variable. We conjecture that this guarantees that left-hand overlays induce equation systems possessing finite complete sets of solutions.

\subsection{Unification background}

A substitution is a family of mappings from variables to expressions $\sigma = (\sigma_X)_{X \in \{I,S,A,L\}}$ where $\sigma_I : \text{IVar} \rightarrow \text{Integer}$, $\sigma_S : \text{SVar} \rightarrow \text{String}$, $\sigma_A : \text{AVar} \rightarrow \text{Atom}$, $\sigma_L : \text{LVar} \rightarrow \text{List}$. Here \text{Integer}, \text{String}, \text{Atom} and \text{List} are the sets of expressions defined by the GP label grammar of Figure 9. For example, if $z \in \text{LVar}$, $x \in \text{IVar}$ and $y \in \text{SVar}$, then we write $\sigma = \{x \mapsto x + 1, z \mapsto y : -x : y\}$ for the substitution that maps $x$ to $x + 1$, $z$ to $y : -x : y$ and every other variable to itself.
Applying a substitution \( \sigma \) to an expression \( t \), denoted by \( t \sigma \), means to replace every variable \( x \) in \( t \) by \( \sigma(x) \) simultaneously. In the above example, \( \sigma(z : -x) = y : -x : y : -(x + 1) \).

By \( \text{Dom}(\sigma) \) we denote the set \( \{ x \in \text{Var} \mid \sigma(x) \neq x \} \) and by \( \text{Vran}(\sigma) \) the set of variables in the range of \( \sigma \). A substitution \( \sigma \) is idempotent if \( \text{Dom}(\sigma) \cap \text{Vran}(\sigma) = \emptyset \).

**Definition 5.1** (Unification problem). A unification problem is a finite multiset of equations

\[
P = \{ s_1 \? t_1, \ldots, s_n \? t_n \}
\]

between simple list expressions.

The symbol \( \? \) signifies that the equations must be solved rather than having to hold for all values of variables.

Consider the equational axioms for associativity and unity,

\[
\text{AU} = \{ x : (y : z) = (x : y) : z, \text{ empty} : x = x, x : \text{ empty} = x \}
\]

and let \( =_{\text{AU}} \) be the equivalence relation on terms generated by these axioms.

**Definition 5.2** (Unifier). Given a unification problem \( P = \{ s_1 \? t_1, \ldots, s_n \? t_n \} \), a unifier of \( P \) is a substitution \( \sigma \) such that

\[
s_1 \sigma =_{\text{AU}} t_1 \sigma, \ldots, s_n \sigma =_{\text{AU}} t_n \sigma.
\]

The set of all unifiers of \( P \) is denoted by \( \mathcal{U}(P) \). We say that \( P \) is unifiable if \( \mathcal{U}(P) \neq \emptyset \).

A substitution \( \sigma \) is more general on a set of variables \( X \) than a substitution \( \theta \) if there exists a substitution \( \lambda \) such that \( x \theta =_{\text{AU}} x \sigma \lambda \) for all \( x \in X \). In this case we write \( \sigma \leq_X \theta \) and say that \( \theta \) is an instance of \( \sigma \) on \( X \). Substitutions \( \sigma \) and \( \theta \) are equivalent on \( X \), denoted by \( \sigma =_X \theta \), if \( \sigma \leq_X \theta \) and \( \theta \leq_X \sigma \).

**Definition 5.3** (Complete Set of Unifiers). A set \( \mathcal{C} \) of substitutions is a complete set of unifiers of a unification problem \( P \) if

1. \( \mathcal{C} \subseteq \mathcal{U}(P) \), that is, each substitution in \( \mathcal{C} \) is a unifier of \( P \),
2. for each \( \theta \in \mathcal{U}(P) \) there exists \( \sigma \in \mathcal{C} \) such that \( \sigma \leq_X \theta \), where \( X = \text{Var}(P) \).

Set \( \mathcal{C} \) is also minimal if it satisfies

3. each two substitutions in \( \mathcal{C} \) are incomparable with respect to \( \leq_X \), that is, for all \( \sigma, \sigma' \in \mathcal{C} \),

\[
\sigma \leq_X \sigma' \implies \sigma = \sigma'.
\]

If a unification problem \( P \) is not unifiable, then the empty set is a minimal complete set of unifiers of \( P \).

The application of a substitution \( \sigma = (s \rightarrow t) \) to a unification problem \( P \), denoted by \( P[s \rightarrow t] \) is obtained by applying the substitution to every term in \( P \), i.e.

\[
P\sigma = \{ s_1 \sigma \equiv t_1 \sigma, \ldots, s_n \sigma \equiv t_n \sigma \}
\]

We call a variable \( x \) solved in \( P \) if it occurs exactly once in \( P \), namely on the left-hand side of some equation \( x = L \) where \( x \notin \text{Var}(L) \) and \( \text{type}(x) \geq \text{type}(L) \).

**Definition 5.4** (Solved form). A unification problem \( P = \{ x_1 \? t_1, \ldots, x_n \? t_n \} \) is in solved form if the variables \( x_i \) are pairwise distinct and solved in \( P \). In this case we define the substitution

\[
\overline{P} = \{ x_1 \mapsto t_1, \ldots, x_n \mapsto t_n \}.
\]
For example, if \( a \) is an atom variable and \( x \) a list variable, then the problems \{x = a\} and \{x = 1 : a\} are in solved form whereas \{x = a : x\}, \{a : 1 = 2 : 1\} and \{a = x\} are not solved. For simplicity, we replace \( = \) with \( = \) in unification problems from now on.

An example unification problem is \{a : x = y : 2\} where \( a \) is an atom variable and \( x \) and \( y \) are list variables. The minimal complete set of unifiers is \{\( \sigma_1, \sigma_2 \)\} where

\[
\sigma_1 = \{a \mapsto 2, x \mapsto \text{empty}, y \mapsto \text{empty}\} \quad \text{and} \quad \sigma_2 = \{x \mapsto z : 2, y \mapsto a : z\}.
\]

We have \( \sigma_1(a : x) = 2 : \text{empty} \overset{\text{AU}}{=} 2 = \sigma_2(y : 2) \) and \( \sigma_2(a : x) = a : z : 2 = \sigma_2(y : 2) \). Other unifiers such as \( \sigma_3 = \{x \mapsto 2, y \mapsto a\} \) are instances of \( \sigma_2 \).

### 5.1.3 Unification Algorithm

We start with some notational conventions for the rest of this section:

- \( L, M \) stand for simple expressions,
- \( x, y, z \) stand for variables of any type (unless otherwise specified),
- \( a, b \) stand for simple string or integer expressions, or atom variables,
- \( s, t \) stand for simple string or integer expressions, or atom variables, or list variables,
- the symbol \( \cup \) denotes multiset union.

**Preprocessing.** Given a unification problem \( P \), we rewrite the terms in \( P \) using the rules

\[
L : \text{empty} \rightarrow L \quad \text{and} \quad \text{empty} : L \rightarrow L
\]

where \( L \) ranges over list expressions. These reduction rules are applied exhaustively before any of the transformation rules. For example,

\[
x : \text{empty} : 1 : \text{empty} \rightarrow x : 1 : \text{empty} \rightarrow x : 1.
\]

We call this process *normalization*. In addition, the rules are applied to each instance of a transformation rule (that is, once the formal parameters have been replaced with actual parameters) before it is applied, and also after each transformation rule application.

**The algorithm.** Given a normalized problem \( P \), the algorithm constructs the search tree of \( P \) where each node is labelled with a unification problem (the root is \( P \)), each success leaf represents a solution. The tree is explored in a breath-first manner and is constructed by picking an equation of \( P \) and examining its head. Depending on the type of symbols at the head, several possible problems are possible, each with the head removed. If the head of an equation is of type:

- \( (x, y) \) or \( (y, x) \) where \( x \) and \( y \) are variables of compatible types, then \( P \) has five children
  - \((P\sigma)_{\text{tail}}\) where \( \sigma = \{x \rightarrow y\} \) (\( x \) and \( y \) are syntactically equated)
  - \((P\sigma)_{\text{tail}}\) where \( \sigma = \{y \rightarrow xy\} \) (\( y \) starts with \( x \))
  - \((P\sigma)_{\text{tail}}\) where \( \sigma = \{x \rightarrow yx\} \) (\( x \) starts with \( y \))
  - \(P\sigma\) where \( \sigma = \{x \rightarrow \text{empty}\} \) (\( x \) is eliminated)
  - \(P\sigma\) where \( \sigma = \{y \rightarrow \text{empty}\} \) (\( y \) is eliminated)

- \( (x, a) \) or \( (a, x) \) where \( x \) and \( a \) are a variable and an atom expression of compatible types, then \( P \) has three children
  - \((P\sigma)_{\text{tail}}\) where \( \sigma = \{x \rightarrow a\} \) (\( x \) and \( a \) are syntactically equated)
\[(P\sigma)_{\text{tail}} \text{ where } \sigma = \{x \rightarrow az\} \text{ (} x \text{ starts with } a, \text{ and is renamed)}\]
\[- P\sigma \text{ where } \sigma = \{x \rightarrow \text{empty}\} \text{ (} x \text{ is eliminated)}\]

- \((a, b) \text{ or } (b, a)\) where \(a\) and \(b\) are atom expressions, then \(P\) has one child
  \[- (P\sigma)_{\text{tail}} \text{ where } \sigma = \{a \rightarrow b\} \text{ (} a\text{ and } b\text{ are syntactically equated)}\]

- \((x, \text{empty})\) or \((\text{empty}, x)\) where \(x\) is a list variable, then \(P\) has one child
  \[- P\sigma \text{ where } \sigma = \{x \rightarrow \text{empty}\} \text{ (} x \text{ is eliminated)}\]

- \((\text{empty}, \text{empty})\) then the equation is eliminated. If \(P\) has no other unsolved equations, then this node is a success leaf.

The above classification is just the intuition behind the actual rules. The full version is effectively a rule-based term rewrite systems on unification problems. See the paper for all the details.

An example tree traversed by the algorithm is shown in Figure 18. Nodes are labelled with unification problems and edges represent applications of transformation rules (see the paper for full details). The root of the tree is the problem \(\{a : x = y : 2\}\) to which several rules can be applied - the head is \((a, y)\) which is the second subcase from above so \(a\) and \(y\) can be equated, or \(y\) can start with \(a\), or \(y\) can be eliminated. The resulting problems are each processed in turn. Eventually, the unifiers

\[
\begin{align*}
\sigma_1 &= \{x \mapsto 2, y \mapsto a\} \\
\sigma_2 &= \{x \mapsto z : 2, y \mapsto a : z\} \\
\sigma_3 &= \{a \mapsto 2, x \mapsto \text{empty}, y \mapsto \text{empty}\}
\end{align*}
\]

are found, which represent a complete set of unifiers of the initial problem. Note that the set is not minimal because \(\sigma_1\) is an instance of \(\sigma_2\).

![Figure 18: Unification example](image-url)
Related Work  The algorithm is similar to the A-unification (word unification) algorithm presented in [55] which looks only at the head of an equation. That algorithm terminates for the special case that the input problem has no repeated variables, and is sound and complete. Our approach can be seen as an extension from A-unification to AU-unification, to handle the unit equations, and presented in the rule-based style of [5]. In addition, our algorithm deals with GP’s subtype system.

Termination, Soundness and Completeness  We show that the unification algorithm terminates if the input problem contains no repeated list variables, where termination of the algorithm follows from termination of the relation ⇨.

We first demonstrate that the algorithm need not terminate on unification problems with repeated list variables. A counterexample is the unification problem \{x:1 = 1:x\} which initiates the following infinite sequence:

\[
\begin{align*}
\{x:1 = 1:x\} \Rightarrow& \text{Subst3} \quad \{x = 1:z_1, \ z_1:1 = x\} \\
\Rightarrow& \text{Subst1} \quad \{x = 1:z_1, \ z_1:1 = 1:z_1\} \\
\Rightarrow& \text{Subst3} \quad \{x = 1:z_1, \ z_1 = 1:z_2, \ z_2:1 = z_1\} \\
\Rightarrow& \text{Subst1} \quad \{x = 1:1:z_2, \ z_1 = 1:z_2, \ z_2:1 = 1:z_2\} \\
\Rightarrow& \text{Subst3} \quad \{x = 1:1:z_3, \ z_1 = 1:z_2, \ z_2 = 1:z_3, \ z_3:1 = z_2\} \\
\Rightarrow& \text{Subst1} \quad \{x = 1:1:1:z_3, \ z_1 = 1:1:z_3, \ z_2 = 1:z_3, \ z_3:1 = 1:z_3\} \\
\Rightarrow& \text{Subst3} \quad \ldots
\end{align*}
\]

Note that \{x:1 = 1:x\} has an infinite number of solutions that are mutually incomparable: \{x \mapsto \text{empty}\}, \{x \mapsto 1\}, \{x \mapsto 1:1\}, \ldots We remark that the A-unification algorithm of [55] also diverges on this problem.

**Theorem 5.5** (Termination). If \(P\) is a unification problem without repeated list variables, then there is no infinite sequence \(P \Rightarrow P_1 \Rightarrow P_2 \Rightarrow \ldots\)

In order to show that the unification algorithm is sound, we need a preliminary lemmata.

**Lemma 5.6.** If \(P \Rightarrow P'\), then \(U(P) \supseteq U(P')\)

**Theorem 5.7** (Soundness). If \(P \Rightarrow^+ P'\) with \(P'\) in solved form, then \(\overrightarrow{P'}\) is an idempotent unifier of \(P\).

**Proof.** A simple induction with Lemma 5.6 shows that \(\overrightarrow{P'}\) must be a unifier of \(P\).

5.1.4 Summary

This section presented groundwork for a static confluence analysis of GP programs. We have constructed a rule-based unification algorithm for systems of equations with left-hand expressions of rule schemata, and have shown that the algorithm always terminates and is sound.

Future work includes proving that our unification algorithm always delivers a complete set of solutions, that is, that every unifier of the input problem is an instance of some unifier in the computed set of solutions. Next, to establish a Critical Pair Lemma in the sense of [47], a notion of independent rule schema applications has to be developed, as well as restriction and embedding theorems for derivations with rule schemata. In addition, since critical pairs contain graphs labelled with expressions, checking joinability of critical pairs will require sufficient conditions under which equivalence of expressions can be decided. This is because the theory of GP’s label algebra includes the undecidable theory of Peano arithmetic.
6 Research Proposal

In this section, I lay out my immediate and long-term research goals. I detail the overall aims of my proposed research and present a strategy for achieving them, along with current progress through the strategy.

6.1 Overall aims

The overall goals of my research is to implement a confluence analysis tool for GP programs as part of the GP compiler stack. To reach this goal, we have broken it down into the following aims:

1. How can the existing theory of finding critical pairs for rule-based transformation systems be modified for the case of sets of GP rule schemata?
2. How can we implement an algorithm to find critical pairs?
3. For which cases can we determine if a critical pair is strongly joinable?
4. How can we extend the theory of confluence in graph transformation systems to graph programs?

6.2 Modifying existing theory to GP rule schemata

The first important step is to come up with an algorithm for computing critical pairs between GP rule schemata. The theory of computing the structure of critical pairs has been developed and presented in the case of unlabelled graphs. However, in the case of labelled graphs, there are no algorithms that compute a complete set of attributed critical pairs. Also, the notions of parallel independence and embedding/restriction have to be extended so that the Critical Pair Lemma can be restated for the case of GP rule schemata.

Validation The algorithm for deciding attribution has to be proven sound and complete. Also, parallel independence must be sufficient to prove the restated Local Church-Rosser theorem (Theorem 3.7). This will allow us to prove the Critical Pair Lemma.

6.3 Implementing critical pair construction

This goes together with my first aim. Based on the definition of critical pairs, this step involves identifying pairs of rule schemata to be checked for independence and joinability and then implementing the construction in the GP compiler stack.

Validation In order to test whether our algorithm is correct, we have to construct test cases of GP rule schemata involves our above definitions of parallel dependence and independence. When a pair is independent, the algorithm should not give false-positives, and when a pair is dependent, it should compute all critical pairs that represent that conflict.

Furthermore, testing the efficiency of the algorithm would be of importance. A possible benchmark would be Ruud Welling's test cases (see below). Although these are used for the evaluation of Groove's which includes generation *and* joinability analysis, we might be able to compare our implementation with theirs. Another option is to create similar test cases for AGG and compare efficiency that way since AGG only supports critical pair generation (without joinability analysis).
6.4 Determining strong joinability

Strong joinability of a critical pair cannot be decided for all cases due to termination. To perform the analysis, this will require state-space search (limited to certain depth) from the two result graphs in the critical pair in order to determine strong joinability. When this is the case, the critical pair is strongly joinable.

However, the algebra of GP labels contains the undecidable theory of Peano arithmetic - given two unrestricted terms involving addition and multiplication, it cannot be computed if they will ever hold the same value. However, a decidable subset of Peano arithmetic called Pressburg arithmetic (does *not* contain multiplication) is a likely restriction at this step to make joinability analysis feasible.

Validation  Local confluence cannot be decided in all cases. However, we can test our implementation on graph transformation systems that have already been proven to be confluent. Ruud Welling [54] already gave a very comprehensive set of test cases where the number of critical pairs is sufficiently large in realistic scenarios such as the Counting Philosophers problem. Furthermore, these test cases have been used to objectify the efficiency of Groove’s [52] local confluence analysis.

6.5 Confluence of graph programs

At this stage the general confluence of GP programs has to be investigated. Sufficient conditions have to be developed that imply the confluence of a program by (non)-confluence of its component programs. This will probably involve exploiting the well-defined structural semantics of GP control constructs. I envisage some sort of relation between the domain of programs and true/false that is inductively defined over the GP operators. For example, if programs $P_1$ and $P_2$ are confluent, then their composition $P_1; P_2$ will also be confluent.

In the case when a subprogram is not confluent, it will be useful to investigate if the outer program can still be confluent. Several GP programs consist of a marking stage where a random node in the host graph is picked, and the rest of the program does some sort of computation while propagating this mark to other nodes (e.g. computing connectivity). The marking phase is truly nondeterministic, but the whole program is confluent.

Validation  In order to test this, we will have to use known programs that exhibit confluence such as automata minimization, serial-parallel checking, reduction systems. See [50] for a recent compilation of GP2 programs, most of which are confluent and a good starting point for confluence test cases.

The main properties of the confluence algorithm are correctness and efficiency. Regarding correctness, the algorithm has to give the correct answer for confluent programs and to be able to compute counter examples to programs for which one exists. Regarding efficiency, the algorithm will be ‘good’ if its time and space complexity compare to that of existing tools. Therefore, the notion of validating if the algorithm is ‘good’ is two-fold.

Current work in confluence analysis is done in AGG and Groove. AGG supports typed attributed graph transformation with NACs and without explicit control. Their critical pair algorithm only consists of the generation phase. Joinability is left to the user.

Groove has a partially correct implementation in the setting of the SPO approach [54]. Their setting is that of attributed graphs and currently do provide joinability analysis with a static bound on the length of joining derivations. The immediate question is how to make a more elaborate, possibly dynamic, search space search instead of using a static bound.
However, in their setting a last minute result showed the solution to be only partially correct, but their work on confluence checking is open source and likely to be relevant.

6.6 Tool Structure

The above theoretic extensions are to be implemented in the GP programming system. Below is a diagram of the relevant 'boxes' to perform the confluence analysis.

- **Rule Analysis** - This segment determines for which pairs of rules to compute critical pairs. The simplest case is a RuleSetCall \( \{r_1, r_2, \ldots, r_n\} \) where each pair of rules give rise to potential critical pairs. When the programs are more involved, the theory extension from subsection 6.5 will extend this part of the tool.

- **Structural CPs** - This part will be concerned with computing the structural critical pairs for each pair of rules when ignoring labels. See section 3.2 for the relevant algorithms.

- **Unification** - Given the basic graphs in the critical pair, this part will attempt to unify the attributes to be overlayed and compute the labels in the critical pair in the sense of section 5. If no unification is possible, then the structural critical pair cannot arise in practice, and should be discarded.

- **Strong Joinability** - For each critical pair, some computable notion of joinability has to be determined at this stage. Note that AGG gives up at this stage, and Groove has a static limit on how long to search for joining derivations. Furthermore, if a pair is found to be strongly joinable before considering labels, then an extra step has to be done to see if the labels can indeed be made equivalent. Note that as already mentioned this step is difficult due to the undecidability of Peano arithmetic, so alternatives have to be considered (e.g. restricting the algebra for which equivalence can be decided).

- **Confluence ‘answer’** - Given the results of joinability analysis of the critical pairs, then either a non-confluence, confluence or 'no-answer' has to be reported back to the user or to the rest of the GP system, together with any supporting information like a confluence counter example or which pairs of rules are independent.

The validation of each step is as discussed above, and the relevant design, implementation and testing phases have been taken into account in the timeplan below.

Since the GP2 compiler is currently still work in progress, implementing my confluence checker would pose difficulties. It might be the case that some of the activities need to be reshuffled or modified in order to account for missing or late relevant parts of the compiler. Second, in the presence of no GP2 compiler, I realistically have several options: - either

- implement my solution in the GP1 environment, or

- implement the relevant parts of the GP2 compiler which are needed for me, or
or use some other graph transformation system to use as supporting environment

The first option is undesirable because the original GP compiler is poorly documented, and is the reason for developing GP2 anew. The second option requires more time w.r.t. actual results because I would be burdened with extra work. The last option seems like a 'Plan B' but plausible due to the existence of graph representation standards like GPX that allow for the communication between different graph transformation systems.

6.7 Timeplan

To conclude this report, I propose a schedule for the rest of my PhD. This is pictorially represented as a Gantt chart below which includes the course milestones. Some milestones can be done in parallel with main work.

The timeplan posses the question of how to recover if things go wrong. The two big phases of developing program analysis or implementing strong joinability might become too big. In order to get a reasonable thesis submissions, this implies that one of them might be chopped or become partial. The strong joinability one seems like the more likely candidate as the novel question of my PhD is about how confluence propagates with control constructs, and thus seems more valuable to me.
Furthermore, the existing smaller cases offer extra flexibility because as seen from the 'boxes' representation, their implementation is largely independent of each other and can be switched around.

References


