
Proceedings Paper:
Plump, Detlef orcid.org/0000-0002-1148-822X and Hristakiev, Ivaylo (2017) Towards Critical Pair Analysis for the Graph Programming Language GP 2. In: James, Phillip and Roggenbach, Markus, (eds.) Recent Trends in Algebraic Development Techniques (WADT 2016), Revised Selected Papers. Lecture Notes in Computer Science . Springer , pp. 153-169.

https://doi.org/10.1007/978-3-319-72044-9_11

Towards Critical Pair Analysis for the Graph Programming Language GP 2

Ivaylo Hristakiev* and Detlef Plump

University of York, United Kingdom

Abstract. We present the foundations of critical pair analysis for the graph programming language GP 2. Our goal is to develop a static checker that can prove or refute confluence (functional behaviour) for a large class of graph programs. In this paper, we introduce *symbolic* critical pairs of GP 2 rule schemata, which are labelled with expressions, and establish the completeness and finiteness of the set of symbolic critical pairs over a finite set of rule schemata. We give a procedure for their construction.

1 Introduction

A common programming pattern in the graph programming language GP 2 [16] is to apply a set of attributed graph transformation rules as long as possible. To execute a set of rules $\{r_1, \ldots, r_n\}$ for as long as possible on a host graph, in each iteration an applicable rule is selected and applied. As rule selection and rule matching are non-deterministic, different graphs may result from such an iteration. Thus, if the programmer wants the loop to implement a function, a static analysis that establishes or refutes functional behaviour would be desirable.

GP 2 is based on the double-pushout approach to graph transformation with relabelling [8]. Programs can perform computations on labels by using rules labelled with expressions (also known as attributed rules). GP 2's label algebra consists of integers, character strings, and heterogeneous lists of integers and strings. Rule application can be seen as a two-stage process where rules are first instantiated, by replacing variables with values and evaluating the resulting expressions, and then applied as usual. Hence rules are actually rule schemata.

Conventional confluence analysis in the double-pushout approach to graph transformation is based on *critical pairs*, which represent conflicts in minimal context [15,5]. A conflict between two rule applications arises when one of the steps deletes or relabels an item matched by the other. In the presence of termination, one can check if all critical pairs are *strongly joinable*, and thus establish that the set of transformation rules is confluent.

However, the conventional notion of critical pairs is not directly applicable to GP 2 rule schemata. To construct such pairs, one needs to instantiate rule schemata to an (usually) infinite set of conventional graph transformation rules

* Supported by a Doctoral Training Grant from the Engineering and Physical Sciences Research Council (EPSRC) in the UK.

[12], and thus the analysis cannot be automated as part of a confluence checker. Furthermore, when constructing the labels of critical pairs, it has been observed [4, p. 198] that syntactic unification of the labels of overlapping graphs is not sufficient (as proposed by [10]). This is because the constructed set of critical pairs need not represent all conflicts. Instead, one has to take into account all equations valid in the attribute algebra. This problem is circumvented in [6,7] by imposing a severe restriction that avoids the need for unification altogether, namely to only allow rules labelled with variables or variable-free expressions.

In this paper, we do not use such restrictions. We rather define *symbolic* critical pairs which are labelled with expressions, and give an algorithm for their construction based on our unification algorithm for GP 2 expressions [11]. As a by-product of this construction, it is easy to show that a finite set of rule schemata gives rise to a finite set of symbolic critical pairs. We then prove that the generated critical pairs are complete in that they represent all conflicts of the given set of rule schemata. This proof is based on the completeness of the GP 2 unification algorithm.

We assume the reader to be familiar with basic notions of the double-pushout approach to graph transformation (see [4]).

Related Work. The approach of [14] also defines symbolic critical pairs in the context of symbolic graph transformation where symbolic graphs are transformed via symbolic rules (rules equipped with first-order logical formulas). Symbolic critical pairs represent all possible conflicts between such symbolic rules. However, it is important to stress the differences with our approach. No construction algorithm is given for these critical pairs whereas we give a construction for the GP 2 setting. In fact, that approach treats attribute algebras as a parameter, and thus a general construction algorithm cannot be given. Even so, a topic of future work is the relaxed notion of conflict where a minimal pair of derivations is critical if the pair does not commute when attribute semantics are taken into account.

The differences with critical pairs in the attributed setting of [4] are similar to the above. In this setting, graph attributes are represented via special *data* nodes and linked to ordinary graph nodes/edges via attribution edges, giving rise to infinite graphs. Attributed rules contain a data node for each term in the term algebra $T(X)$. The critical pair construction however is restricted to rules whose attributes are variables or variable-free, e.g. see [6]. An earlier version of the construction was based on computing a most general unifier [10], which renders the critical pairs incomplete. As above, attribute algebras are treated as parameters.

2 Graphs and Graph Programs

In this section, we present the approach of GP 2 [16,2], a domain-specific language for rule-based graph manipulation. The principal programming units of GP 2 are rule schemata $\langle L \leftarrow K \rightarrow R \rangle$ labelled with expressions that operate

on host graphs labelled with concrete values. The language allows to combine schemata into programs. The definition of GP 2's latest version, together with a formal operational semantics, can be found in [2].

2.1 Background

Labelled graphs. We start by recalling the basic notions of partially labelled graphs and their morphisms.

A (partially) labelled graph G consists of finite sets V_G and E_G of nodes and edges, source and target functions for edges $s_G, t_G \colon E_G \to V_G$, and a partial node/edge labelling function $l_G \colon V_G + E_G \to \mathcal{L}$ over a (possibly infinite) label set $\mathcal{L}$. Given a node or edge x, $l_G(x) = \bot$ expresses that $l_G(x)$ is undefined[1]. The graph G is totally labelled if l_G is a total function. The classes of partially and totally labelled graphs over a label set $\mathcal{L}$ are denoted by $\mathcal{G}_\bot(\mathcal{L})$ and $\mathcal{G}(\mathcal{L})$.

A premorphism $g \colon G \to H$ consists of two functions $g_V \colon V_G \to V_H$ and $g_E \colon E_G \to E_H$ that preserve sources and targets. A graph morphism g is a premorphism that preserves labels of nodes and edges, that is $l_H\big(g(x)\big) = l_G(x)$ for all $x \in Dom(l_G)$. A morphism g preserves undefinedness if it maps unlabelled items of G to unlabelled items in H. Morphism g is an inclusion if $g(x) = x$ for all items x in G. Note that inclusions need not preserve undefinedness. Morphism g is injective (surjective) if g_V and g_E are injective (surjective), and is an isomoprhism (denoted by $\cong$) if it is injective, surjective and preserves undefinedness. The class of injective label preserving morphisms is denoted as $\mathcal{M}$ for short, and the class of injective label and undefinedness preserving morphisms is denoted as $\mathcal{N}$.

Partially labelled graphs and label-preserving morphisms constitute a category [9,8]. Composition of morphisms is defined componentwise. What is special about this category is that pushouts need not always exist, and not all pushouts along $\mathcal{M}$-morphisms are natural[2].

GP 2 labels. The types `int` and `string` represent integers and character strings. The type `atom` is the union of `int` and `string`, and `list` represents lists of atoms. Given lists l_1 and l_2, we write $l_1 : l_2$ for the concatenation of l_1 and l_2 (not to be confused with the list-cons operator in Haskell). Atoms are lists of length one. The empty list is denoted by `empty`. Variables may appear in labels in rules and are typed over the above categories. Labels in rule schemata are built up from constant values, variables, and operators - the standard arithmetic operators for integer expressions (including the unary minus), string concatenation for string expressions, `length` operator for list and string expressions, `indegree` and `outdegree` operators for nodes. In pictures of graphs, nodes or edges that are shown without a label are implicitly labelled with the empty list, while unlabelled items in interfaces are labelled with $\bot$ to avoid confusion.

[1] We do not distinguish between nodes and edges in statements that hold analogously for both sets.

[2] A pushout is natural if it is also a pullback.

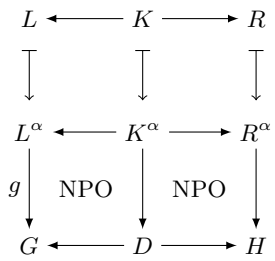


Fig. 1: A direct derivation

Additionally, a label may contain an optional *mark* which is one of `red`, `green`, `blue`, `grey` and `dashed` (where `grey` and `dashed` are reserved for nodes and edges, respectively). The mark component of labels is represented graphically rather than textually. For example, all edges of the rule schema `series` in Figure 2 have the label (`empty`, `dashed`).

Rule schemata and direct derivations. In order to compute with labels, it is crucial that nodes and edges can be relabelled. The double-pushout approach with partially labelled interface graphs is used as a formal basis [8].

To apply a rule schema to a graph, the schema is first instantiated by evaluating its labels according to some assignment α. An assignment α maps each variable occurring in a given schema to a value in GP 2's label algebra. Its unique extension α^* evaluates the schema's label expressions according to α. For short, we denote GP 2's label algebra as A. Its corresponding term algebra over the same signature is denoted as $T(X)$, and its terms are used as graph labels in rule schemata. Here X is the set of variables occurring in schemata. To avoid an inflation of symbols, we sometimes equate A or $T(X)$ with the union of its carrier sets.

A GP 2 rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$ consists of two inclusions $K \rightarrow L$ and $K \rightarrow R$ such that L and R are graphs in $\mathcal{G}(T(X))$ and K is a graph in $\mathcal{G}_\bot(T(X))$. Consider a graph G in $\mathcal{G}_\bot(T(X))$ and an assignment $\alpha\colon X \rightarrow A$. The instance G^α is the graph in $\mathcal{G}_\bot(A)$ obtained from G by replacing each label l with $\alpha^*(l)$. The instance of a rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$ is the rule $r^\alpha = \langle L^\alpha \leftarrow K^\alpha \rightarrow R^\alpha \rangle$.

A direct derivation via rule schema r and assignment α between host graphs $G, H \in \mathcal{G}(A)$ consists of two natural pushouts as in Figure 1. We denote such a derivation by $G \overset{r,g,\alpha}{\Longrightarrow} H$. Rules may also be applied to graphs in $\mathcal{G}(T(X))$. In this case assignments become substitutions $\sigma : X \rightarrow T(X)$. This will be useful later for critical pairs which are labelled over $T(X)$.

In [8] it is shown that in case the interface graph K has unlabelled items, their images in the intermediate graph D are also unlabelled by the condition that the pushouts are natural. Given a rule r and a graph G together with an injective match $g\colon L \rightarrow G$ satisfying the *dangling condition* (no node in $g(L) - g(K)$ is

incident to an edge in $G - g(L)$), there exists a unique double natural pushout [8, Theorem 1].

When a rule schema is graphically declared as done in Figure 2, the interface is represented by the node numbers in L and R. Nodes without numbers in L are to be deleted and nodes without numbers in R are to be created. All variables in R have to occur in L so that for a given match of L in a host graph, applying the rule schema produces a graph that is unique up to isomorphism.

Program constructs. A GP 2 program consists of declarations of rule schemata and macros, and a main command sequence which controls their application order. The language offers several operators for combining subprograms - the postfix operator '!' iterates a program as long as possible; sequential composition 'P; Q'; a rule set $\{r_1, \ldots, r_n\}$ tries to non-deterministically apply any of the schemata (failing if none are applicable); `if` C `then` P `else` Q allows for conditional branching (C,P,Q are arbitrary command sequences) meaning that if the program C succeeds on a copy of the host graph then P is executed on the original, if C fails then Q is executed on the original host graph.

Simple lists. The values of rule schema variables at execution time are determined by graph matching. To ensure that matches induce unique "actual parameters", expressions on the left-hand side of a rule schema must have a simple shape. A simple list expression [2] contains no arithmetic, length or degree operators, at most one occurrence of a list variable, at most one occurrence of a string variable per string expression. For example, `a:x` and `y:n:n` are simple expressions (`a,n` are atom variables; `x,y` are list variables) whereas `n` $*$ `2` or `x:y` are not simple.

Assumptions. In this paper, we make several assumptions. First, we further restrict simple lists to not contain string concatenation (`.`) and unary minus (`-`) operators. These operators inflate the unification algorithm of [11] which we use for the construction of critical pairs, without posing a substantial challenge. Second, a proper treatment of GP 2 *conditional* rule schemata requires extra technicalities, and hence we consider unconditional schemata only. Third, we assume rule schemata to be *left-linear* (see Section 3).

2.2 Example: Recognition of series-parallel graphs

As a motivating example, consider a GP program that recognizes *series-parallel graphs*. These graphs have been introduced as models of electrical networks [3], and are interesting from a complexity point of view since many graph problems, some of which NP-complete, are solvable in linear time for these graphs e.g. maximum matching, maximum independent set, Hamiltonian completion.

These graphs are recognized by means of graph reduction: for a host graph G, apply a set of size-reducing rules `Reduce = {series, parallel}` as long as possible, obtaining a result graph H, then check whether H is isomorphic to $\bigcirc\!\!\longrightarrow\!\!\bigcirc$ (ignoring labels) to decide whether the original graph G is series-parallel.

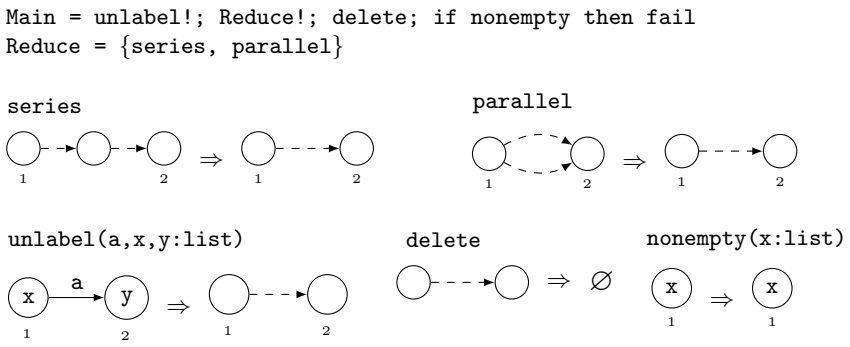


Fig. 2: GP 2 program recognizing series-parallel graphs.

A GP 2 program implementing the above algorithm is presented in Figure 2. Given a host graph G, the program works as follows. First, it removes all labels by applying the `unlabel` rule as long as possible (labels do not play a role in whether a graph is series-parallel or not). Applying `unlabel` amounts to non-deterministically selecting a subgraph of the host graph that matches `unlabel`'s left graph, relabelling the matched nodes with the empty list and recreating the connecting edge as `dashed` to avoid non-termination.

Afterwards, the `Reduce` rules are applied as long as possible. To determine whether the resulting graph has the correct shape, the program first attempts to delete the correct result graph (see above) then checks whether this yields the empty graph. If either the deletion or the non-empty check fails, then the program fails. In this context, termination of the program with a proper graph means the host graph G is series-parallel, and failure means G is not series-parallel.

However, if the non-deterministic reduction results in a graph other than ◯- - →◯, we need to be sure that no other reduction sequence ends in that graph. Therefore, the correctness of the above recognition algorithm depends on the *confluence* of the loops `unlabel!` and `Reduce!`.

Confluence [15] is a property of a rewrite system that ensures that any pair of derivations on the same host graph can be joined again thus leading to the same result, and is an important property for many kinds of graph transformation systems. A confluent computation is globally deterministic despite possible local non-determinism. The main technique for confluence analysis is based on the study of critical pairs which are conflicts in minimal context. However, the previous results in critical pair analysis do not cover rule schemata and GP. This raises the question of how to check whether loops such as `unlabel!` and `Reduce!` are confluent or not.

Infinity of conventional critical pairs. To construct critical pairs for the above case, we can consider the infinite set of all rules obtained by arbitrary instantiations of the schemata and compute conventional critical pairs over those (see,

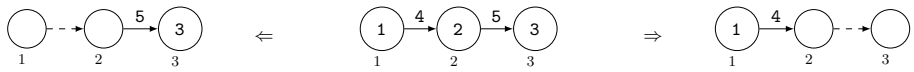


Fig. 3: A conventional critical pair of `unlabel` with itself.

for example, [4]). Since there would be an infinite number of rule instances to consider, the set of conventional critical pairs would also be infinite.

For example, consider the conflicting pair of derivations in Figure 3. The middle graph is obtained by overlapping the left-hand graph of `unlabel` with itself, and the graphs on either side are the results of applying the schema in conflicting ways. The pair is in conflict because both derivations relabel a common node (2) to the empty list. The instantiating assignment of `unlabel` and its copy is $\alpha = \{\mathtt{x1} \to 1, \mathtt{y1} \to 2, \mathtt{x2} \to 2, \mathtt{y2} \to 3, \mathtt{a1} \to 4, \mathtt{a2} \to 5\}$ where the variables are indexed to signify from which `unlabel` instance they originate.

3 Unification of GP 2 List Expressions

Below we review the problem of unifying GP 2 list expressions. The problem arises when having to overlap graphs labelled with expressions to compute critical pairs. Unification has a long history in the automated deduction community, see for instance [1] for an introduction. We use our AU-unification algorithm [11] as a solution, and its properties - namely completeness and termination. As mentioned in the Introduction, the use of our algorithm is motivated by the need to respect the axioms valid in the label algebra. (See Section 6 for more on the relation between critical pairs and unification.)

A substitution maps GP variables to expressions $\sigma : X \to T(X)$. For example, we write $\sigma = \{\mathtt{x} \mapsto \mathtt{x} + 1\}$ for the substitution that maps `x` (an integer variable) to $\mathtt{x} + 1$ and every other variable to itself. Applying a substitution σ to an expression t, denoted by $t\sigma$, means to replace every variable x in t by $\sigma(x)$ simultaneously. In the above example, $(\mathtt{x} : -\mathtt{x})\sigma = (\mathtt{x} + 1) : -(\mathtt{x} + 1)$. Composition of substitutions λ and σ is written as $\lambda \circ \sigma$ (λ *after* σ). Given substitutions $\sigma_1, \ldots, \sigma_n$ with pairwise disjoint domains, their composition $\sigma_1 \circ \ldots \circ \sigma_n$ is commutative. A substitution σ is *more general* on a set of variables X than a substitution θ if there exists a substitution λ such that $x\theta =_{\mathrm{AU}} (x\sigma)\lambda$ for all $x \in X$. In this case we write $\sigma \leqq_X \theta$ and say that θ is an instance of σ on X. Here $=_{\mathrm{AU}}$ is the equivalence relation on expressions generated by the axioms of associativity and unity of list concatenation $\mathrm{AU} = \{\mathtt{x} : (\mathtt{y} : \mathtt{z}) = (\mathtt{x} : \mathtt{y}) : \mathtt{z},\ \mathtt{empty} : \mathtt{x} = \mathtt{x},\ \mathtt{x} : \mathtt{empty} = \mathtt{x}\}$, where $\mathtt{x}, \mathtt{y}, \mathtt{z}$ are list variables. If one considers ordinary equality $=$, then the unification is called *syntactic*.

A *unification problem* is an equation P of the form $s =^? t$ where s and t are simple list expressions without common variables. A *unifier* of P is a substitution σ over the set of variables occurring in P (denoted as $\mathrm{Var}(P)$) such that $s\sigma =_{\mathrm{AU}} t\sigma$.

A set $\mathcal{C}$ of unifiers is a *complete set of unifiers* of a unification problem P if for each unifier θ there exists $\sigma \in \mathcal{C}$ such that $\sigma \leqq_{\mathrm{Var}(P)} \theta$. This essentially means that any substitution that is a unifier is an instance of some unifier in $\mathcal{C}$. The set $\mathcal{C}$ is also *minimal* if each pair of distinct unifiers in $\mathcal{C}$ are incomparable w.r.t. $\leqq_{\mathrm{Var}(P)}$. If a unification problem is not unifiable, then by convention $\varnothing$ is its minimal complete set of unifiers.

The paper [11] gives an algorithm for solving unification problems between GP 2 list expressions. The algorithm produces a finite complete set of unifiers for a given problem. The assumptions of the algorithm are: 1) simple expressions, as presented in Section 2; 2) *left-linearity* (see below); 3) infinite pool of fresh variables. We can summarize the results of [11] as the following theorem.

Theorem 1 (Unification algorithm). *There exists an algorithm solving the following problem:*

Input: A unification problem $s =^? t$ between simple list expressions s and t without common variables

Output: A finite complete set of unifiers $\mathrm{UNIF}(s =^? t)$

We write $\mathrm{UNIF}(P)$ for the set of unifiers returned by the unification algorithm. For a finite system of independent unification problems $(P_1, \ldots, P_n)$[3], the extension of UNIF is defined to be the set of unifiers obtained by combining the unifiers of each individual unification problem:

$$\mathrm{UNIF}(P_1, \ldots, P_n) = \left\{\sigma_1 \circ \ldots \circ \sigma_n \mid \sigma_i \in \mathrm{UNIF}(P_i),\ 1 \leq i \leq n\right\}$$

For example, if $\mathrm{UNIF}(P_1) = \{\alpha, \beta\}$ and $\mathrm{UNIF}(P_2) = \{\lambda\}$, then $\mathrm{UNIF}(P_1, P_2) = \{\alpha \circ \lambda,\ \beta \circ \lambda\}$. By the above result, this set is finite. It is also complete since it contains all combinations of unifiers.

Left-linearity. The left-linearity assumption states that no list variables are shared between items in a left-hand graph of a rule schema. This is sufficient to ensure that we can apply our generalized algorithm in the construction of critical pairs (Theorem 2) as the system of equations resulting from overlapping left-hand graphs will have a finite set of solutions. Without this assumption it is easy to construct two rule schemata that induce the system of equations $\{\mathtt{x} : 1 =^? \mathtt{y},\ 1 : \mathtt{x} =^? \mathtt{y}\}$. The system is not independent as the list variables `x` and `y` are shared between the two equations. We can solve each equation separately, but the composition of their unifiers does not produce a unifier for the system. In fact, this system has an *infinite* minimal complete set of solutions $\{\mathtt{x} \mapsto \mathtt{empty}, \mathtt{y} \mapsto 1\}$, $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1 : 1\}$, $\{\mathtt{x} \mapsto 1 : 1, \mathtt{y} \mapsto 1 : 1 : 1\}, \ldots$.

Unification example. The minimal complete set of unifiers of the problem $\langle \mathtt{a} : \mathtt{x} =^? \mathtt{y} : 2\rangle$ (where `a` is an atom variable and `x`,`y` are list variables) is $\{\sigma_1, \sigma_2\}$ with $\sigma_1 = \{\mathtt{a} \mapsto 2, \mathtt{x} \mapsto \mathtt{empty}, \mathtt{y} \mapsto \mathtt{empty}\}$ and $\sigma_2 = \{\mathtt{x} \mapsto \mathtt{z} : 2, \mathtt{y} \mapsto \mathtt{a} : \mathtt{z}\}$. We have $(\mathtt{a} : \mathtt{x})\sigma_1 = 2 : \mathtt{empty} =_{\mathrm{AU}} 2 =_{\mathrm{AU}} \mathtt{empty} : 2 = (\mathtt{y} : 2)\sigma_1$ and $(\mathtt{a} : \mathtt{x})\sigma_2 = \mathtt{a} : (\mathtt{z} : 2) =_{\mathrm{AU}} (\mathtt{a} : \mathtt{z}) : 2 = (\mathtt{y} : 2)\sigma_2$. Other unifiers such as $\sigma_3 = \{\mathtt{x} \mapsto 2, \mathtt{y} \mapsto \mathtt{a}\}$ are instances of σ_2, hence set of unifiers $\{\sigma_1, \sigma_2, \sigma_3\}$ is complete but not minimal.

[3] Two unification problems are independent if they do not share list variables.

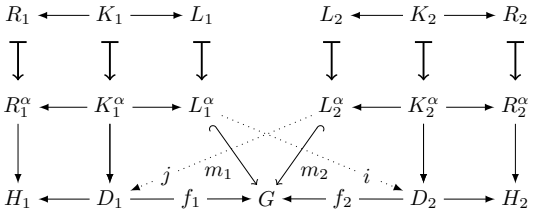
4 Symbolic Critical Pairs

In this section, we define the notions of independence and conflicts for rule schema rewriting as done in [12]. Then we develop the notion of symbolic critical pairs describing conflicts in minimal context. Symbolic critical pairs allow for the realization of a static confluence checker.

Independence of schema derivations. Two schema derivations are independent if neither derivation deletes or relabels any common item. This can be expressed as an 'existence-of-morphisms' condition. Independent derivations can be interchanged, leading to the same result. This property is known as the Local Church-Rosser Theorem, shown in [12] for the case of rule schemata.

In the rest of the paper we assume that the variables occurring in different rule schemata are distinct, which can always be achieved by variable renaming.

Definition 1 (Independence of derivations). Two rule schema direct derivations $G \overset{r_1,m_1,\alpha}{\Longrightarrow} H_1$ and $G \overset{r_2,m_2,\alpha}{\Longrightarrow} H_2$ are *independent* if the plain derivations with relabelling $G \overset{r_1^\alpha,m_1}{\Longrightarrow} H_1$ and $G \overset{r_2^\alpha,m_2}{\Longrightarrow} H_2$ are independent, meaning that there exist morphisms $i : L_1^\alpha \to D_2$ and $j : L_2^\alpha \to D_1$ such that $f_2 \circ i = m_1$ and $f_1 \circ j = m_2$.



Two direct derivations are in *conflict* if they are not independent. There are different types of conflict that can arise between two direct derivations. One option is to have that either derivation deletes graph elements which are used by the other (delete-use conflict). The other option is that one derivation relabels graph elements used by the other (relabelling conflict).

Example of conflict. Figure 4 shows two direct derivations $H_1 \Leftarrow G \Rightarrow H_2$ that use instances of the rule schema `unlabel`. The derivations are in conflict - there are no morphisms $L_1 \to D_2$ and $L_2 \to D_1$ with the desired properties. The problem is that node 2 gets relabelled. Note that the edge from node 1 to 3 is never matched and is therefore preserved during both derivations.

Symbolic critical pairs

Critical pairs allow for the static confluence analysis of rule schema rewriting. Each conflict that may occur during the graph transformation is represented by a critical pair. Hence, it is possible to foresee each conflict by computing all critical pairs statically. Each pair of rule schemata induces a set of critical pairs.

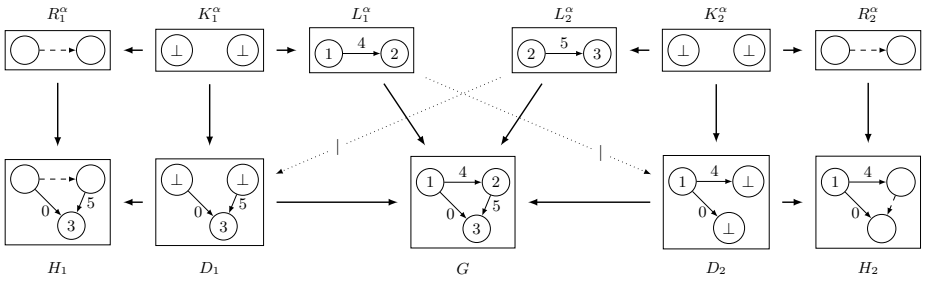


Fig. 4: Conflict due to relabelling.

We define critical pairs that are labelled with expressions rather than from a concrete data domain. Each symbolic critical pair represents a possibly infinite set of conflicting host graph derivations. What is special about our critical pairs is that they show the conflict in the most abstract way.

A pair of derivations $T_1 \overset{r_1,m_1,\sigma}{\Longleftarrow} S \overset{r_2,m_2,\sigma}{\Longrightarrow} T_2$ between graphs labelled with expressions is a critical pair if it is in conflict and minimal. Minimality means the pair of matches (m_1, m_2) is jointly surjective – the graph S can be considered as a suitable overlap of L_1^σ and L_2^σ. Two items $x \in L_1$ and $y \in L_2$ are overlapped if $m_1(x) = m_2(y)$, which induces a unification problem $l(x) =^? l(y)$ between their labels. Formally, overlapping graphs L_1^σ and L_2^σ induces a system of unification problems:

$$\mathrm{EQ}(L_1^\sigma \xrightarrow{m_1} S \xleftarrow{m_2} L_2^\sigma) = \{l_{L_1^\sigma}(a) \overset{?}{=} l_{L_2^\sigma}(b) \mid (a,b) \in L_1^\sigma \times L_2^\sigma \text{ with } m_1(a) = m_2(b)\}$$

The substitution σ is taken from a complete set of unifiers of the above system of problems and is used to instantiate the schemata. To avoid a circular definition, the system can instead be constructed over the induced premorphisms $L_i \to S$ rather than over $L_i^\sigma \to S$, $i = 1, 2$.

To disambiguate between the critical pairs of our approach and conventional critical pairs found in literature (e.g. [15]), we introduce them as *symbolic*[4].

Definition 2 (Symbolic Critical Pair). *A symbolic critical pair is a pair of direct derivations $T_1 \overset{r_1,m_1,\sigma}{\Longleftarrow} S \overset{r_2,m_2,\sigma}{\Longrightarrow} T_2$ on graphs labelled with expressions such that:*

(1) σ is a substitution in $\mathrm{UNIF}(\mathrm{EQ}(L_1 \xrightarrow{m_1} S \xleftarrow{m_2} L_2))$ *where L_1 and L_2 are the left-hand graphs of r_1 and r_2, m_1 and m_2 are premorphisms, and*
(2) the pair of derivations is in conflict, and
(3) $S = m_1(L_1^\sigma) \cup m_2(L_2^\sigma)$, meaning S is minimal, and
(4) $r_1^\sigma = r_2^\sigma$ implies $m_1 \neq m_2$. □

We assume the derivations are via left-linear rule schemata for UNIF to return a finite set of unifiers. Terms appearing in left-hand graphs are restricted to simple lists with an optional mark component as presented in Section 2.

[4] The paper [14] introduces symbolic critical pairs in the setting of symbolic graph transformation where graphs are combined with first-order logic formulas.

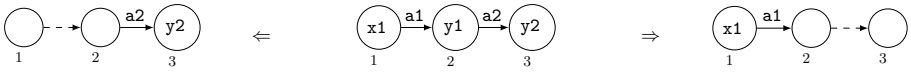


Fig. 5: A symbolic critical pair of `unlabel` with itself.

Critical pairs of the Series-Parallel Program. An example symbolic critical pair of the rules in Figure 2 is shown in Figure 5 where the middle graph is obtained by overlapping the left-hand graph of the `unlabel` schema with itself, and the graphs on either side are the results of applying the schema in conflicting ways. The pair is in conflict because both derivations relabel a common node (2) to the empty list. Note that this symbolic critical pair looks very similar to the pair of conflicting derivations in Figure 3. In fact, they are related by the instantiation $\lambda = \{\mathtt{x1} \to \mathtt{1}, \mathtt{y1} \to \mathtt{2}, \mathtt{y2} \to \mathtt{3}, \mathtt{a1} \to \mathtt{4}, \mathtt{a2} \to \mathtt{5}\}$ where the variables are indexed (as usual) to signify from which `unlabel` instance they originate.

There are 4 more symbolic critical pairs obtained by self-overlapping `unlabel`. In addition, the set `Reduce` gives rise to 3 symbolic critical pairs. Both loops `unlabel!` and `Reduce!` can be shown to be locally confluent by analysing these critical pairs under a suitable notion of critical pair joinability (to be published elsewhere). It follows that the program in Figure 2 is correct.

5 Construction and Finiteness of Symbolic Critical Pairs

We give an algorithm for the construction of symbolic critical pairs that respects the equations of GP's label algebra, namely the associativity and unit laws of list concatenation. The construction is given as the following theorem.

Theorem 2 (Construction of Symbolic Critical Pairs). *Given left-linear rule schemata $r_1 = \langle L_1 \leftarrow K_1 \rightarrow R_1 \rangle$ and $r_2 = \langle L_2 \leftarrow K_2 \rightarrow R_2 \rangle$, the following construction computes all symbolic critical pairs of r_1 and r_2:*

1. *Compute all overlaps of L_1 and L_2, giving rise to pairs of jointly surjective premorphsisms (m_1, m_2) into an unlabelled graph S.*
2. *For each overlap check that m_1 and m_2 satisfy the dangling condition w.r.t. r_1 and r_2.*
3. *For each overlap compute the set of unifiers* $\mathrm{UNIF}(\mathrm{EQ}(L_1 \overset{m_1}{\to} S \overset{m_2}{\leftarrow} L_2))$.
4. *For each unifier σ from the above set and its overlap do the following:*
   (a) *Instantiate r_1 and r_2 via σ to obtain the rules r_1^σ and r_2^σ, and if $r_1^\sigma = r_2^\sigma$ check that $m_1 \neq m_2$.*
   (b) *Define the labelling function of S as*

$$l_S(x) = \begin{cases} l_{L_1^\sigma}(x') & \text{if } \exists x' \in L_1^\sigma \text{ such that } m_1(x') = x \\ l_{L_2^\sigma}(x') & \text{if } \exists x' \in L_2^\sigma \text{ such that } m_2(x') = x \end{cases}$$

   (c) *Construct the derivations $T_1 \overset{r_1, m_1, \sigma}{\Longleftarrow} S \overset{r_2, m_2, \sigma}{\Longrightarrow} T_2$.*

(d) If the pair of derivations is in conflict, then it is a symbolic critical pair.

Proof. We show that the above construction produces exactly all symbolic critical pairs according to Definition 2. The construction computes only symbolic critical pairs - when Step 4.d is reached, the pair of derivations exists, is minimal and in conflict, and is labelled using one of the substitutions returned by UNIF. The construction computes all critical pairs since all overlaps and all unifiers per overlap are considered. □

The construction of symbolic critical pairs is similar to that of conventional critical pairs. The most important difference occurs when overlapping graph nodes or edges since unification needs to be considered. This process terminates in finite time - overlapping finite graphs produces a finite number of overlaps (Step 1), left-linearity allows for UNIF to produce a finite set of substitutions (Step 3), all other components are either finite checks or constructing a pair of direct derivations. Recall that the number of conventional critical pairs is infinite in general due to the infinite number of rules that a schema represents. Consequently, the computation of symbolic critical pairs is much more suitable for automation as part of a confluence checker.

Corollary 1 (Finiteness of Symbolic Critical Pairs). *For each pair of left-linear rule schemata r_1 and r_2, the set of symbolic critical pairs induced by r_1 and r_2 is finite.*

Proof sketch. Since the above construction computes all critical pairs and terminates, then the set of symbolic critical pairs must be finite. □

6 Completeness of Symbolic Critical Pairs

Completeness of critical pairs means that each pair of conflicting direct derivations is an instance of a symbolic critical pair. Formally, we state the result as a theorem. We start by discussing the link between unification of expressions and completeness of critical pairs.

Critical Pairs and Unification. As shown in Section 2.2, one needs to consider critical pairs labelled with expressions rather than concrete values. This means one needs an algorithm to compute their labels which are expressions resulting from overlapping left-hand graphs of rules. If one does not severely restrict the shape of labels, this computation involves unification. However, the type of unification becomes crucial when considering whether the constructed critical pairs are complete, as we show below.

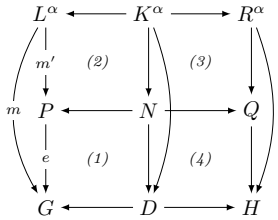
Consider two rule schemata with left-hand sides $\textcircled{\texttt{1:x}}$ and $\textcircled{\texttt{y:1}}$ where `x` and `y` are `list` variables. Overlapping these graphs induces the unification problem $P = \langle \texttt{1} : \texttt{x} =^? \texttt{y} : \texttt{1} \rangle$. This problem can be syntactically unified via its most general unifier $\sigma = \{\texttt{x}, \texttt{y} \to \texttt{1}\}$. However, the problem has an infinite number of

AU-unifiers: $\{\mathtt{x}, \mathtt{y} \to \mathtt{empty}\}$, $\{\mathtt{x}, \mathtt{y} \to \mathtt{1}\}$, $\{\mathtt{x}, \mathtt{y} \to \mathtt{1} : \mathtt{1}\}$, ..., none of which are instances of σ except one. As a consequence, critical pairs labelled using most general unifiers are incomplete. To our knowledge this problem has been first observed in [4, p. 198] in the context of attributed graph transformation. Their solution is to restrict the shape of labels/attributes to variable-free or variable-only terms which avoids the need for unification.

Our solution to this problem involves using our complete AU-unification algorithm - solving P produces the AU-unifiers $\{\mathtt{x}, \mathtt{y} \to \mathtt{empty}\}$ and $\{\mathtt{x} \to \mathtt{x}' : \mathtt{1}, \mathtt{y} \to \mathtt{1} : \mathtt{x}'\}$ where $\mathtt{x}'$ is a fresh list variable. (Our algorithm also produces σ making the result set non-minimal.) Consider any assignment α such that $(\mathtt{1} : \mathtt{x})\alpha =_{\mathrm{AU}} (\mathtt{y} : \mathtt{1})\alpha$. By Theorem 1, there exists a unifier $\sigma \in \mathrm{UNIF}(P)$ and instantiating substitution λ such that $\alpha = \lambda \circ \sigma$. Thus, a symbolic critical pair labelled using σ can be instantiated via λ to a critical pair of host graph derivations. Consequently, completeness of our AU-unification algorithm allows for greater representational power when it comes to critical pairs.

Restriction Lemma. In the following, we present a restriction construction, formulated only for direct derivations, which is in some sense the inverse of extending a derivation to a larger context. This construction is necessary for the proof of Theorem 3.

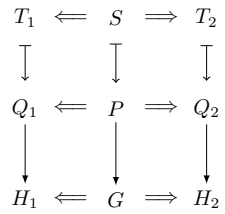
Lemma 1 (Restriction). *Given a direct derivation $G \overset{r,m,\alpha}{\Longrightarrow} H$, a morphism $e : P \to G \in \mathcal{N}$, and a match $m' : L^\alpha \to P \in \mathcal{N}$ such that $m = e \circ m'$, then there is a direct derivation $P \overset{r,m',\alpha}{\Longrightarrow} Q$ leading to the (extension) diagram below.*



Proof. See the long version of this paper [13].

Completeness of symbolic critical pairs. Next we state our Completeness Theorem. The technical aspects of its proof (see [13]) are concerned with the properties of partially labelled graphs $\mathcal{G}_\perp$ and the classes of horizontal and vertical morphisms in direct derivations ($\mathcal{M}$ and $\mathcal{N}$). These basic properties have already been studied in [9,8]. Below we give a proof sketch, including only of the important steps.

Theorem 3 (Completeness of Symbolic Critical Pairs). *For each pair of conflicting rule schema applications $H_1 \overset{r_1,m_1,\alpha}{\Longleftarrow} G \overset{r_2,m_2,\alpha}{\Longrightarrow} H_2$ between left-linear schemata r_1 and r_2 there exists a symbolic critical pair $T_1 \overset{r_1}{\Longleftarrow} S \overset{r_2}{\Longrightarrow} T_2$ with (extension) diagrams between $H_1 \Leftarrow G \Rightarrow H_2$ and an instance of $T_1 \Leftarrow S \Rightarrow T_2$.*



Proof sketch. We start by decomposing the pair of matches $(m_1 : L_1^\alpha \to G, m_2 : L_2^\alpha \to G)$ (Figure 6) to obtain a graph $P = m_1(L_1^\alpha) \cup m_2(L_2^\alpha) = m_1'(L_1^\alpha) \cup m_2'(L_2^\alpha)$ together with jointly surjective matches $(m_1' : L_1^\alpha \to P, m_2' : L_2^\alpha \to P)$ and morphism $e : P \to G \in \mathcal{N}$.

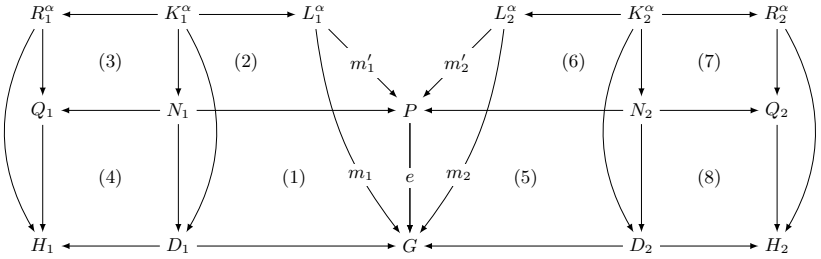


Fig. 6: Decomposed pushouts

Next we apply Lemma 1 twice to obtain the restricted derivations $P \Rightarrow Q_1$ and $P \Rightarrow Q_2$. It is not difficult to show that $Q_1 \Leftarrow P \Rightarrow Q_2$ is minimal and in conflict using the commutativity of (1), the properties of (m_1', m_2'), and Definition 1 (e.g. see the proof of Lemma 6.22 in [4]). This concludes the first part of the proof.

For the second part we will show that $Q_1 \Leftarrow P \Rightarrow Q_2$ is an instance of a symbolic critical pair $T_1 \Leftarrow S \Rightarrow T_2$. We use the fact that the assignment α is an AU-unifier for the system of equations $\mathrm{EQ}(L_1, L_2, m_1, m_2)$ and therefore, by Theorem 1 (r_1 and r_2 are left-linear), α is an instance of a unifier $\sigma \in \mathrm{UNIF}(\mathrm{EQ}(L_1, L_2, m_1', m_2'))$ such that $\alpha = \lambda \circ \sigma$ where λ is some assignment.

Next we construct the symbolic critical pair $T_1 \Leftarrow S \Rightarrow T_2$. The graphs have the same node/edge sets as $Q_1 \Leftarrow P \Rightarrow Q_2$ but different labels. First, instantiate L_1 and L_2 via σ to obtain graphs L_1^σ and L_2^σ. Then define $S = m_1'(L_1^\sigma) \cup m_2'(L_2^\sigma)$. This definition is sound because σ is a unifier. It is easy to show that $P \cong S^\lambda$ using $\alpha = \lambda \circ \sigma$.

We proceed by constructing the derivation $S \overset{r_1, m_1', \sigma}{\Longrightarrow} T_1$ - the double-pushout is (9+10) of Figure 7 together with the instantiation squares right above it. The same construction is applied to obtain $S \Rightarrow T_2$. By Definition 2, it follows that $T_1 \overset{r_1, m_1', \sigma}{\Longleftarrow} S \overset{r_2, m_2', \sigma}{\Longrightarrow} T_2$ is a symbolic critical pair – (m_1', m_2') are jointly surjective, σ is a unifier, and it can be shown the derivations are in conflict since $Q_1 \Leftarrow P \Rightarrow Q_2$ is in conflict. □

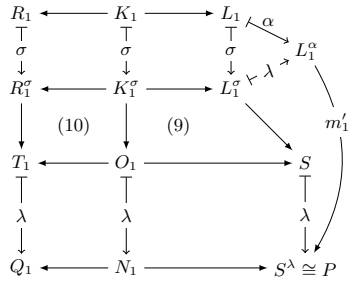


Fig. 7: Construction of $S \Rightarrow T_1$.

Example of completeness. Consider the pairs of derivations in Figure 3, Figure 4 and Figure 5. They form the layers of the diagram in Theorem 3. The morphism $e : P \to G$ is an inclusion where G contains the extra edge from node 1 to 3. The assignment λ linking the symbolic critical pair to its instance is $\lambda = \{\texttt{x1} \to 1, \texttt{y1} \to 2, \texttt{y2} \to 3, \texttt{a1} \to 4, \texttt{a2} \to 5\}$.

7 Conclusion and Future Work

We have presented the foundations of critical pair analysis for the graph programming language GP 2. Our goal is to develop a static checker that can verify or refute confluence (functional behaviour) for a large class of graph programs. We have introduced *symbolic* critical pairs of GP 2 rule schemata, which are labelled with expressions, and established the completeness and finiteness of the set of symbolic critical pairs over a finite set of rule schemata. We have given a procedure for constructing that set.

We are currently working on proving the Local Confluence Theorem for GP 2, which establishes local confluence of sets of rule schemata for the case that all symbolic critical pairs are strongly joinable. The precise definition of joinability is an interesting problem from an algorithmic point of view, and so is the development of a procedure that determines program confluence by analysing critical pairs. Another interesting topic is the role of SMT solvers for deciding label equivalences and implications in the context of isomorphism checking and joinability analysis.

References

1. Baader, F., Snyder, W.: Unification theory. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 2, pp. 445–532. Elsevier and MIT Press (2001)
2. Bak, C.: GP 2: Efficient Implementation of a Graph Programming Language. Ph.D. thesis, University of York (2015), `http://etheses.whiterose.ac.uk/id/eprint/12586`
3. Duffin, R.J.: Topology of series-parallel networks. Journal of Mathematical Analysis and Applications 10(2), 303–318 (1965), doi:10.1016/0022-247X(65)90125-3

4. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science, Springer (2006), doi:10.1007/3-540-31188-2
5. Ehrig, H., Golas, U., Habel, A., Lambers, L., Orejas, F.: $\mathcal{M}$-adhesive transformation systems with nested application conditions: Part 2: Embedding, Critical Pairs and Local Confluence. Fundamenta Informaticae 118(1-2), 35–63 (2012), doi:10.3233/FI-2012-705
6. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: Proc. International Conference on Graph Transformations (ICGT 2004). pp. 161–177 (2004), doi:10.1007/978-3-540-30203-2_13
7. Golas, U., Lambers, L., Ehrig, H., Orejas, F.: Attributed graph transformation with inheritance: Efficient conflict detection and local confluence analysis using abstract critical pairs. TCS 424, 46–68 (2012), doi:10.1016/j.tcs.2012.01.032
8. Habel, A., Plump, D.: Relabelling in graph transformation. In: Proc. International Conference on Graph Transformation (ICGT 2002). LNCS, vol. 2505, pp. 135–147. Springer (2002), doi:10.1007/3-540-45832-8_12
9. Habel, A., Plump, D.: $\mathcal{M},\mathcal{N}$-adhesive transformation systems. In: Proc. International Conference on Graph Transformation (ICGT 2012). LNCS, vol. 7562, pp. 218–233. Springer (2012), doi:10.1007/978-3-642-33654-6_15
10. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: Corradini, A., Ehrig, H., Kreowski, H., Rozenberg, G. (eds.) Proc. International Conference on Graph Transformation (ICGT 2002). LNCS, vol. 2505, pp. 161–176. Springer (2002), doi:10.1007/3-540-45832-8_14
11. Hristakiev, I., Plump, D.: A unification algorithm for GP 2. In: Graph Computation Models (GCM 2014), Revised Selected Papers. Electronic Communications of the EASST, vol. 71 (2015), `http://journal.ub.tu-berlin.de/eceasst/article/view/1002`, doi:10.14279/tuj.eceasst.71.1002
12. Hristakiev, I., Plump, D.: Attributed graph transformation via rule schemata: Church-Rosser theorem. In: Graph Computation Models (GCM 2014), part of Software Technologies: Applications and Foundations (STAF 2016), Revised Selected Papers. LNCS, vol. 9946, pp. 145–160. Springer (2016), doi:10.1007/978-3-319-50230-4_11
13. Hristakiev, I., Plump, D.: Towards critical pair analysis for the graph programming language GP 2 (long version) (2017), `https://www.cs.york.ac.uk/plasma/publications/pdf/HristakievPlump.WADT16.Long.pdf`
14. Kulcsár, G., Deckwerth, F., Lochau, M., Varró, G., Schürr, A.: Improved conflict detection for graph transformation with attributes. In: Proc. Graphs as Models (GaM 2015). EPTCS, vol. 181, pp. 97–112 (2015), doi:10.4204/EPTCS.181.7
15. Plump, D.: Confluence of graph transformation revisited. In: Processes, Terms and Cycles: Steps on the Road to Infinity: Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday, LNCS, vol. 3838, pp. 280–308. Springer (2005), doi:10.1007/11601548_16
16. Plump, D.: The design of GP 2. In: Proc. International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011). EPTCS, vol. 82, pp. 1–16 (2012), doi:10.4204/EPTCS.82.1