UNIVERSITY of York

This is a repository copy of *Checking Graph Programs for Confluence*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/124164/

Version: Accepted Version

Proceedings Paper:

Plump, Detlef orcid.org/0000-0002-1148-822X and Hristakiev, Ivaylo (2018) Checking Graph Programs for Confluence. In: Seidl, Martina and Zschaler, Steffen, (eds.) Software Technologies: Applications and Foundations - STAF 2017 Collocated Workshops, Revised Selected Papers. Lecture Notes in Computer Science . Springer , pp. 92-108.

https://doi.org/10.1007/978-3-319-74730-9_8

Reuse

Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

White Rose
university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Checking Graph Programs for Confluence

Ivaylo Hristakiev and Detlef Plump

University of York, York, United Kingdom

Abstract. We present a method for statically verifying confluence (functional behaviour) of terminating sets of rules in the graph programming language GP 2, which is undecidable in general. In contrast to other work about attributed graph transformation, we do not impose syntactic restrictions on the rules except for left-linearity. Our checking method relies on constructing the symbolic critical pairs of a rule set using an E-unification algorithm and subsequently checking whether all pairs are strongly joinable with symbolic derivations. The correctness of this method is a consequence of the main technical result of this paper, viz. that a set of left-linear attributed rules is locally confluent if all symbolic critical pairs are strongly joinable, and our previous results on the completeness and finiteness of the set of symbolic critical pairs. We also show that for checking strong joinability, it is not necessary to compute all graphs derivable from a critical pair. Instead, it suffices to focus on the pair's persistent reducts. In a case study, we use our method to verify the confluence of a graph program that calculates shortest distances.

1 Introduction

A common programming pattern in the graph programming language GP 2 [18] 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 [6]. 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 [17,4]. A conflict between two rule applications arises, roughly speaking, when one of the steps cannot be applied to the result of 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.

In our previous paper [12], we developed the notion of symbolic critical pairs for GP 2 rule schemata which are minimal conflicting derivations, labelled with expressions. The set of such pairs is finite and complete, in the sense that they represent all possible conflicts that may arise during computation. Furthermore, we gave an algorithm for constructing the set of symbolic critical pairs induced by a set of schemata, which uses our E-unification algorithm of [9]. The approach does not place severe restrictions on labels appearing in rules, as the attributed setting of [3]. What remains to be shown is how to use such critical pairs in the context of confluence analysis.

In this paper, we present our method for statically verifying confluence of terminating sets of GP 2 rules. We introduce a notion of symbolic rewriting that allows us to rewrite the graphs of critical pairs, and show how it is used for confluence analysis. The correctness of our analysis is a consequence of the main technical result of this paper, namely that a set of left-linear attributed rules is locally confluent if all symbolic critical pairs are strongly joinable. We also show that for checking strong joinability, it is not necessary to compute all graphs derivable from a critical pair but it suffices to focus on the pair's persistent reducts. In a case study, we use our method to verify the confluence of a graph program that calculates shortest distances.

We assume the reader to be familiar with basic notions of the double-pushout approach to graph transformation (see [3]). The long version version of this paper [11] contains the technical proofs together with the full shortest distances case study.

2 Graphs and Graph Programs

In this section, we present the approach of GP 2 [18,1], 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 (or input graphs) labelled with concrete values. The language also allows to combine schemata into programs. The definition of GP 2's latest version, together with a formal operational semantics, can be found in [1]. We start by recalling the basic notions of partially labelled graphs and their morphisms.

Labelled graphs. A (partially) labelled graph G consists of finite sets V_G and E_G of nodes and edges (graph items for short), 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 an item x, $l_G(x) = \perp$ expresses that $l_G(x)$ is undefined. The graph G is totally labelled if l_G is a total function. The classes of partially and totally labelled graphs over $\mathcal{L}$ are denoted as $\mathcal{G}_\perp(\mathcal{L})$ 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, and is a graph morphism if it preserves labels of graph items, 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. A morphism g is an inclusion if $g(x) = x$ for all items x in G.

Note that inclusions need not preserve undefinedness. A morphism g is injective (surjective) if g_V and g_E are injective (surjective), and is an isomorphism (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 [7,6]. Composition of morphisms is defined componentwise. In this category not all pushouts exist, and not all pushouts along $\mathcal{M}$-morphisms are natural[1].

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/list concatenation for string/list expressions, `indegree` and `outdegree` operators for nodes. In pictures of graphs, graph items that are shown without a label are implicitly labelled with the empty list, while unlabelled items in interfaces are labelled with $\perp$ to avoid confusion.

Additionally, a label may contain an optional *mark* which is represented graphically as a colour. For example, the grey node of the rule schema `init` in Figure 3 has the label $(\mathtt{x} : \mathtt{0}, \mathtt{grey})$.

Rule schemata and direct derivations. In order to compute with labels, it is necessary that graph items can be relabelled during computation. The double-pushout approach with partially labelled interface graphs is used as a formal basis [6]. This approach is also the foundation of GP 2.

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. A substitution σ maps variables to terms. 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 $\mathcal{G}_\perp(T(X))$. Consider a graph G in $\mathcal{G}_\perp(T(X))$ and an assignment $\alpha \colon X \rightarrow A$. The instance G^α is the graph in $\mathcal{G}_\perp(A)$ 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 $r^\alpha = \langle L^\alpha \leftarrow K^\alpha \rightarrow R^\alpha \rangle$.

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

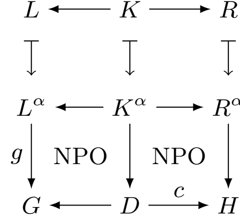


Fig. 1: A direct derivation

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 $G \overset{r,g,\alpha}{\Longrightarrow} H$. Later we will allow for rules to be applied to graphs in $\mathcal{G}(T(X))$. In [6] it is shown that in case the interface graph K has unlabelled items, their images in the intermediate graph D are also unlabelled. By [6, Theorem 1], given a rule r and a graph G together with an injective match $g\colon L \to G$ 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 as in Figure 1. The *track* morphism allows to "follow items through derivations": $track_{G \Rightarrow H} : G \to H$ is the partial premorphism defined by $track(x) =$ **if** $x \in D$ **then** $c(x)$ **else** $undefined$ where c is the inclusion $D \to H$, and $track_{G \Rightarrow^* H}$ of an arbitrary-length derivation is the composition of partial premorphisms. Note *track* may not preserve labels due to relabelling.

For an example rule schema and graph program, see the start of Section 5. When a rule schema is graphically declared as done in Figure 3, 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. The language GP 2 offers several operators for combining programs - 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 programs) meaning that if the program C succeeds on a copy of the input graph then P is executed on the original, if C fails then Q is executed on the original input graph.

Confluence. A set of rule schemata $\mathcal{R}$ is *confluent* if for all graphs G, H_1, H_2 with derivations $H_1 \Leftarrow^*_\mathcal{R} G \Rightarrow^*_\mathcal{R} H_2$ there is a graph M with $H_1 \Rightarrow^*_\mathcal{R} M \Leftarrow^*_\mathcal{R} H_2$. $\mathcal{R}$ is *locally* confluent if this property holds for direct derivations $H_1 \Leftarrow_\mathcal{R} G \Rightarrow_\mathcal{R} H_2$. Finally, $\mathcal{R}$ is *terminating* if there is no infinite sequence $G_0 \Rightarrow_\mathcal{R} G_1 \Rightarrow_\mathcal{R} \ \ldots$ of direct derivations.

Assumptions. Our previous results on critical pairs [12] involve several restrictions. Firstly, the proper treatment of GP 2 *conditional* rule schemata requires extra results about shifting of conditions along morphisms and rules, which we

do not treat here formally. We give an intuition of how to deal with conditions on labels in our shortest distances case study in Section 5. Secondly, we assume rule schemata to be *left-linear*, meaning no list variables are shared between items in schemata. This ensures that overlapping graphs with expressions results in a finite set of critical pairs. Thirdly, we allow interfaces in rules to contain edges and labels, which is a deviation from the GP 2 convention of unlabelled node-only interfaces. This reduces the number of potential conflicts.

3 Symbolic Critical Pairs

Confluence [17] 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 local nondeterminism. The main technique for confluence analysis is based on the study of critical pairs which are conflicts in minimal context.

In our previous paper [12], we defined critical pairs for GP 2 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. Hence, it is possible to foresee each conflict by computing all critical pairs statically. What is special about our critical pairs is that they show the conflict in the most abstract way. Informally, 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 symbolic critical pair if it is in conflict and minimal. Two direct derivations are independent if neither derivation deletes or relabels any common item, and in conflict if otherwise. Independent derivations have the Church-Rosser property as shown in [10] for the case of rule schemata.

Minimality of a pair of derivations 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. Formally, overlapping graphs L_1 and L_2 via premorphisms $m_1 : L_1 \to S, m_2 : L_2 \to S$ induces a system of unification problems:

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

The substitution σ above is taken from the complete set of unifiers of the above system computed by our unification algorithm of [9], and is used to instantiate the schemata to a critical pair.

Definition 1 (Symbolic Critical Pair [12]). *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 from a complete set of unifiers of $(\mathrm{EQ}(L_1 \xrightarrow{m_1} S \xleftarrow{m_2} L_2))$ 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 that the variables occurring in different rule schemata are distinct, which can always be achieved by variable renaming. The derivations have to be via left-linear rule schemata in order for our unification algorithm to work on the systems of unification problems EQ. For example critical pairs, see Section 5.

Properties of Symbolic Critical Pairs. Below we present the properties of critical pairs as proven in [12]. Symbolic critical pairs are complete, meaning that each pair of conflicting direct derivations is an instance of a symbolic critical pair. Additionally, the set of symbolic critical pairs is finite.

Theorem 1 (Completeness and Finiteness of Critical Pairs [12]). *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* $T_1 \overset{r_1}{\Leftarrow} S \overset{r_2}{\Rightarrow} T_2$ *with the (extension) diagrams (1) and (2) between* $H_1 \Leftarrow G \Rightarrow H_2$ *and an instance of* $T_1 \Leftarrow S \Rightarrow T_2$*. Moreover, the set of symbolic critical pairs induced by* r_1 *and* r_2 *is finite.*

$$\begin{array}{ccccc} T_1 & \Longleftarrow & S & \Longrightarrow & T_2 \\ \downarrow & & \downarrow & & \downarrow \\ Q_1 & \Longleftarrow & P & \Longrightarrow & Q_2 \\ \downarrow & (1) & \downarrow & (2) & \downarrow \\ H_1 & \Longleftarrow & G & \Longrightarrow & H_2 \end{array}$$

4 Symbolic Rewriting and Joinability

Symbolic critical pairs consist of graphs labelled with expressions. This is necessary for making the set of critical pairs finite as GP 2's infinite label algebra induces an infinite set of conflicts at the instance (host) level. How to rewrite such graphs is what we focus on next as the current GP 2 framework only defines rewriting of host graphs.

In this section we propose *symbolic rewriting* of GP 2 graphs to overcome the above limitation. This allows for the representation of multiple host graph direct derivations. The overall aim is to use symbolic rewriting for establishing the (strong) joinability of critical pairs.

4.1 Symbolic Rewriting

Informally, symbolic rewriting introduces a relation on rule graphs ($\Rrightarrow$) where matching is done by treating variables as typed symbols/constants. What is special in our setting is that rules cannot introduce new variables. Furthermore, since application conditions cannot usually be checked for satisfiability as values for variables are not known at analysis time, they are only recorded as assumptions to be resolved later. This type of rewriting is very similar to symbolic graph transformation, e.g. as in [16].

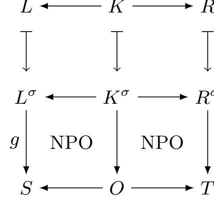


Fig. 2: Symbolic direct derivation.

Isomorphism and label equivalence. Since we now consider graphs in $\mathcal{G}(T(X))$ involving GP 2 label expressions, we relax the definition of isomorphism presented in Section 2 by replacing label equality with *equivalence*. Furthermore, since graph isomorphism is central to the discussion of joinability of critical pairs, we define how isomorphism relates to a critical pair's set of persistent nodes.

Two graphs $G, H \in \mathcal{G}(T(X))$ are E-isomorphic (denoted by $G \cong_E H$) if there exists a bijective premorphism $i : G \to H$ such that $l_H(i(x)) \approx_E l_G(x)$ for all items of G. Here $\approx_E$ is the equivalence relation on GP2 expressions given by all the equations valid in GP 2's label algebra of integer arithmetic and list/string concatenation.

For an example of why a more general notion of isomorphism is needed, consider the schemata r_1: `m:n` $\Rightarrow$ `m+n` and r_2: `m:n` $\Rightarrow$ `n+m` which both match a node labelled with a list of two integers (`m` and `n`) but relabel the node to (syntactically) different expressions. The derivations `n+m` $\Leftarrow_{r_1}$ `m:n` $\Rightarrow_{r_2}$ `m+n` represent a symbolic critical pair (conflict due to relabelling). The resulting graphs are normal forms, and isomorphic only if one considers the commutativity of addition.

Symbolic derivation. The essence of symbolic rewriting is to allow rule schemata to be applied to graphs labelled with expressions, i.e. graphs in $\mathcal{G}(T(X))$. In the terminology of Section 2, assignments become substitutions $\sigma : X \to T(X)$. We call such a derivation *symbolic*. For example, the critical pairs in Figure 4 involve such symbolic derivations. Operationally, constructing symbolic derivations involves obtaining a substitution σ for the variables of L given a premorphism $L \to S$, and then constructing a direct derivation with relabelling as in Section 2.

Definition 2 (Symbolic direct derivation). *A symbolic direct derivation via rule schema r, substitution σ between graphs $S, T \in \mathcal{G}(T(X))$ consists of two natural pushouts via match $g : L^\sigma \to S$ as in Figure 2.*

We denote symbolic derivations by $S \overset{r,g,\sigma}{\Rrightarrow} T$. Note that variables occurring in S cannot be modified. This kind of rewriting is incomplete in that not all host graph derivations can be represented by symbolic derivations.

Symbolic derivations allow for the representation of multiple host graph direct derivations, and can be seen as transformations of specifications. The propo-

sition below states that the application of symbolic rule schema coincides, in some sense, with respect to the host graph derivations it represents. For its proof see [11].

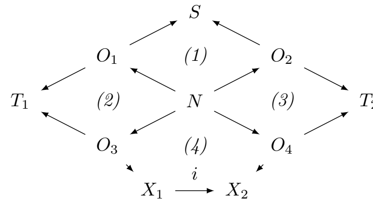
Lemma 1 (Soundness of symbolic rewriting). *For each symbolic derivation* $S \overset{r,g,\sigma}{\Rrightarrow} T$ *and each host graph* $G = S^\lambda$ *and assignment* λ*, there exists a direct derivation* $G \overset{r,g,\alpha}{\Rightarrow} H$ *where* $H = T^\lambda$ *and* $\alpha = \lambda \circ \sigma$*.*

4.2 Joinability

Confluence analysis is based on the joinability of critical pairs. Informally, a symbolic critical pair $T_1 \Lleftarrow S \Rrightarrow T_2$ is *joinable* if there exist symbolic derivations from T_1 and T_2 to a common graph. However, it is known that joinability of all critical pairs is not sufficient to prove local confluence [17]. Instead, one needs to consider a slightly stronger notion called *strong* joinability that requires a set of *persistent* nodes in a critical pair to be preserved by the joining derivations. The set of persistent items of a critical pair consists of all nodes in S that are preserved by both steps, and are defined in terms of the pullback N of the intermediate graphs O_1 and O_2 of the critical pair[2].

Definition 3 (Strong joinability). *A symbolic critical pair* $T_1 \Lleftarrow S \Rrightarrow T_2$ *is strongly joinable if we have the following:*

1. *joinability: there exist symbolic derivations* $T_1 \Rrightarrow^* X_1 \cong_E X_2 \Lleftarrow^* T_2$ *where* $i : X_1 \to X_2$ *is an E-isomorphism.*
2. *strictness: let* N *be the pullback object of* $O_1 \to S \leftarrow O_2$ *(1). Then there exist morphisms* $N \to O_3$ *and* $N \to O_4$ *such that the squares (2), (3) and (4) commute:*



The strictness condition can be restated in terms of the track morphisms of the joining derivations, as in [17]: the track morphisms $track_{S \Rrightarrow T_1 \Rrightarrow^* X_1}$ and $track_{S \Rrightarrow T_1 \Rrightarrow^* X_2}$ are defined and commute on the persistent items of the critical pair, i.e. $i(track_{S \Rrightarrow T_1 \Rrightarrow^* X_1}(x)) = track_{S \Rrightarrow T_2 \Rrightarrow^* X_2}(x)$ for each $x \in N$. See the first author's thesis for a proof of equivalence [8]. The graphs O_3 and O_4 in the above definition are the derived spans of the joining derivations as the joining derivations are of arbitrary length, e.g. see [11] and [3].

Lemma 2 (Joinability preservation). *If a symbolic critical pair* $T_1 \Lleftarrow S \Rrightarrow T_2$ *is strongly joinable, then each of its instances according to some assignment* λ *(*$T_1^\lambda \Leftarrow S^\lambda \Rightarrow T_2^\lambda$*) is also strongly joinable.*

Here we consider the critical pair instances to be critical pairs over the rule instances in their own right. For proof(s), see [11].

[2] For the construction of pullbacks over partially labelled graphs see [7, Section 4]

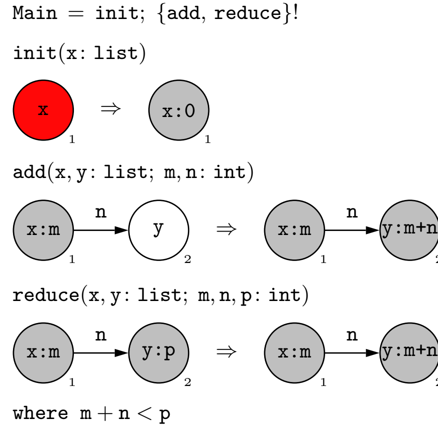


Fig. 3: Shortest Distances program

5 Case Study: Shortest Distances

The shortest distances problem is about calculating the paths between a given node (the *source* node) and all other nodes in a graph such that the sum of the edge weights on each path is minimized. The Bellman–Ford algorithm [2] is an algorithm that solves that problem. It is based on *relaxation* in which the current distance to a node is gradually replaced by more accurate values until eventually reaching the optimal solution. An assumption made is that there is no *negative cycle* (a cycle whose edge weights sum to a negative value) that is reachable from the source, in which case there is no shortest path.

GP 2 implementation. A GP 2 program that implements the above algorithm is shown in Figure 3. Distances from the source node are recorded by appending the distance value to each node's label. Nodes *marks* are used: the source node is red, visited nodes are gray, and unvisited nodes are unmarked. Given an input graph G with a unique source node and no negative cycle, the program initializes the distance of the source node to 0. The `add` rule explores the unvisited neighbours of any visited nodes, assigns them a tentative distance and marks them as visited to avoid non-termination. The `reduce` rule finds occurrences of visited nodes whose current distance is higher than alternative distances, i.e. only when the application condition ($\mathtt{m} + \mathtt{n} < \mathtt{p}$) is satisfied by the schema instantiation. The program terminates when neither `add` or `reduce` rules can be further applied.

However, since rule application is non-deterministic, different graphs may result from a program execution. The above algorithm is correct only if the loop {`add,reduce`}`!` is confluent. In the absence of a full program verification, a programmer may want to check that this loop indeed returns unique results.

Critical Pairs. There are 7 critical pairs in total for the above program: two between `add` with itself (SD1/2), one between `add` and `reduce` (SD3), and four between `reduce` with itself (SD4-7). Figure 4 gives the first three critical pairs,

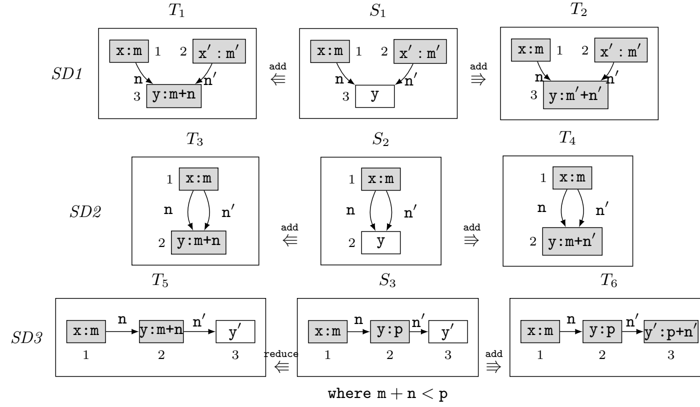


Fig. 4: Shortest Distances critical pairs involving `add`.

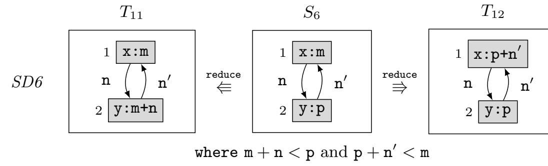


Fig. 5: A `reduce` critical pair requiring a semantic joinability argument.

whereas the `reduce` critical pairs are very similar to those and are omitted for space reasons. The only interesting `reduce` critical pair involves a 2-cycle where either node gets its distance updated by `reduce` and is given in Figure 5. All of the conflicts are due to relabelling of a common node. Note that due to the semantics of GP 2 marks (marked cannot match unmarked), other conflicts are not possible. Variables have been renamed where necessary. The persistent items of all critical pairs are the graph items of S since the rules do not delete any graph items, and the common node of each critical pair that gets relabelled (e.g. node 3 in SD1) does not have a label in the pullback graph N of Definition 3.

The critical pairs SD1/2 are between the rule `add` with itself where an unvisited node can get initialized with different distance values, either from 2 neighbouring nodes or from the same node but different (parallel) edges. In SD3 the distance of a node in a path is used in different ways: either to initialize the distance of a neighbouring node (via `add`), or to have its own distance updated (via `reduce`). Note that the application condition is recorded as part of the critical pair. The critical pairs SD4/5 represent a conflict of `reduce` with itself where a node may get different updated distance values depending on which path is chosen, similar to SD1/2. SD6 involves a 2-cycle where either node gets its distance updated by `reduce`. SD7 involves a sequence of three nodes, similar to SD3.

Joinability Analysis. Due to space limitations, here we only give a top-level explanation of why each of the critical pairs are strongly joinable. See [11] for the full details. The result of the analysis is that all critical pairs are strongly

```
(define-fun T1_T2() Bool
    (forall ((m1 Int) (m2 Int)
             (n1 Int) (n2 Int))
            (= (+ m1 n1) (+ m2 n2)) ))
(assert T1_T2)
```

(a) Label equivalence example for SD1.

```
(define-fun T77_T888 () Bool
    (forall ((m Int) (p Int)
             (n1 Int) (n2 Int))
        (=> (< (+ m n1) p)
            (< (+ m n1 n2) (+ p n2)) )))
(assert T77_T888)
```

(b) Implication checking for SD3.

Fig. 6: Z3 code for label equivalence analysis of shortest distances.

joinable except the 2-cycle critical pair SD6 whose label condition is unsatisfiable assuming non-negative cycles and the semantic argument that both schemata do not modify edge labels. (Without using this information, the critical pair is not joinable.) Hence the loop {`add, reduce`}`!` is confluent.

An interesting practical aspect of joinability is that it involves, in most cases, checking label equivalences for validity. (We check for validity rather than satisfiability since we need that all instances of a strongly joinable critical pair to be strongly joinable rather than at least one.) For this purpose, we use the SMT solver Z3 [15]. It provides support for (linear) integer arithmetic, arrays, bit vectors, quantifiers, implications, etc.

For the critical pair SD1, the result graphs T_1 and T_2 are isomorphic only if the label equivalence $m + n = m' + n'$ is valid, which it is not (encoded as a `forall` expression in Figure 6a where variables have been renamed). The analysis proceeds by applying `reduce` to both T_1 and T_2, and the semantics of the `reduce` condition (containing comparison of integer expressions) guarantees a strong isomorphism between the results. Note that `reduce` is necessary for the joining derivations, meaning the rule `add` is not confluent on its own. The analysis of SD2 proceeds in a similar way as SD1 with the same conclusion. For SD3, one needs to check *implications* between conditions to ensure strong joinability between a pair of derivable graphs. An implication that shows up during the analysis is shown in Figure 6b which Z3 reports to be valid. Therefore the critical pair is strongly joinable. The analysis for the critical pairs SD4/5 is the same as for SD1/2.

The critical pair SD6 is different than the rest - its label condition is satisfiable only when the sum of the edge labels is negative ($\mathtt{n} + \mathtt{n}' < 0$), which is not possible under the assumption of no negative cycles and the observation that no rules modify edge labels. Without this semantic information, it is possible to instantiate the critical pair to a concrete graph with non-isomorphic normal forms, and thus obtain an example of non-confluence.

6 Local Confluence

In this section we present the Local Confluence Theorem which establishes the local confluence of $\mathcal{R}$ if all symbolic critical pairs are strongly joinable. It was first shown in [17] for the (hyper)graph case and later extended to (weak) adhesive categories in [5]. We also discuss our method for confluence checking based on symbolic critical pairs.

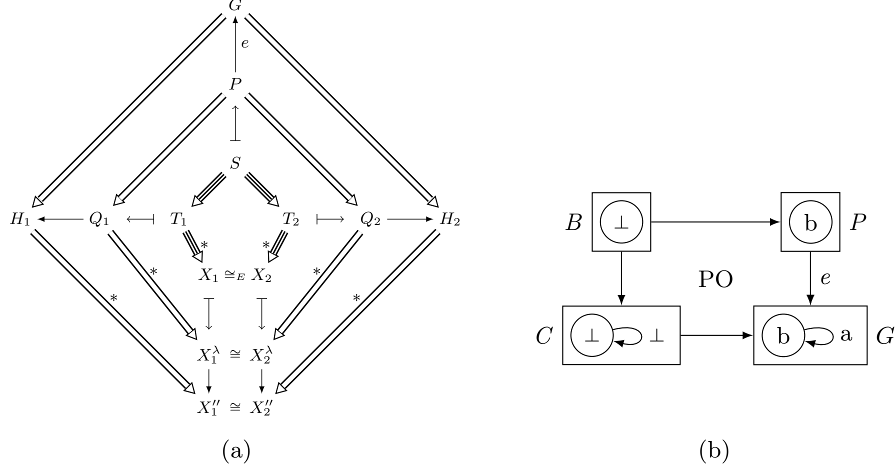


Fig. 7: Local Confluence Diagram (a) and Initial pushout in $\mathcal{G}_\perp$ (b).

Theorem 2 (Local Confluence Theorem). *A set $\mathcal{R}$ of left-linear rule schemata is locally confluent if all of its symbolic critical pairs are strongly joinable.*

The full proof closely follows the Local Confluence Theorem proof of [3, Theorem 6.28], which requires several properties of $\mathcal{M}$ and $\mathcal{N}$ established in [7]. Due to space limitations, here we give only an outline containing the important steps.

Proof outline. For a given pair of direct derivations $H_1 \overset{r_1,m_1,\alpha}{\Leftarrow} G \overset{r_2,m_2,\alpha}{\Rightarrow} H_2$, we have to show the existence of derivations $H_1 \Rightarrow^*_{\mathcal{R}} X_1'' \cong X_2'' \Leftarrow^*_{\mathcal{R}} H_2$ as the outer part of Figure 7a. If the given pair is independent, this follows from the Church-Rosser Theorem for rule schemata [10]. If the given pair is in conflict, Theorem 1 implies the existence of a symbolic critical pair $T_1 \overset{r_1,m_1',\sigma}{\Leftarrow} S \overset{r_2,m_2',\sigma}{\Rightarrow} T_2$ with extension diagrams as in the upper half of Figure 7a involving an instance of the critical pair, and the extension morphism $e : P \to G \in \mathcal{N}$. By assumption, this critical pair is strongly joinable. By Lemma 2, the critical pair instance $Q_1 \Leftarrow P \Rightarrow Q_2$ is also strongly joinable, leading to derivations $P \Rightarrow Q_1 \Rightarrow^* X_1^\lambda \cong X_2^\lambda \Leftarrow^* Q_2 \Leftarrow P$ where λ is the instantiation of the symbolic critical pair.

The next step is to show that the joining derivations $P \Rightarrow Q_1 \Rightarrow^* X_1^\lambda \cong X_2^\lambda \Leftarrow^* Q_2 \Leftarrow P$ can be extended via the morphism $e : P \to G \in \mathcal{N}$. This involves constructing the initial pushout of e and showing that the joining derivations preserve its boundary graph. Here the commutativity of the squares (2) and (3) in Definition 3 is used, together with the properties of pullbacks and initial pushouts. The final step involves showing that $X_1'' \cong X_2''$. This is due to the commutativity of (4) in Definition 3 and that pushouts are unique up to isomorphism. □

Remark. The full proof of the theorem requires the construction of the boundary/context graph of $e : P \to G \in \mathcal{N}$. This is always possible in our setting - use the same definition as in the unlabelled case (e.g. see [3, Example 6.2]) and omit

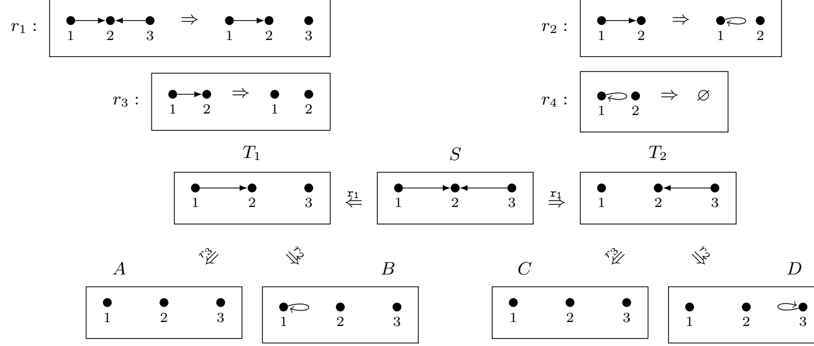


Fig. 8: Joinability analysis with persistent reducts.

labels as done in Figure 7b. Other necessary results include the Embedding and Extension Theorems, which are easily obtained by inspecting the proofs in [5] which already considers categories with a special set of vertical morphisms.

Confluence analysis. Next we give our decision procedure for confluence based on symbolic critical pairs. In the following, we consider only terminating sets of rules $\mathcal{R}$ since a non-terminating rule set may be locally confluent but not confluent. We begin by discussing persistent reducts.

In the context of a critical pair $T_1 \Leftarrow_{\mathcal{R}} S \Rightarrow_{\mathcal{R}} T_2$ with a set of persistent items N, a graph X derivable from T_1 or T_2 is a *persistent reduct* if the only rules applicable to X would delete the image of a common persistent item, i.e. an item in $track_{S \Rightarrow T_i \Rightarrow^* X}(N)$. Such graphs are useful when searching for joining derivations – one need not consider graphs derivable from such reducts because strong joinability requires the existence of all persistent items, expressed as the following proposition. For its proof see [11]

Proposition 1. *If a critical pair $T_1 \Leftarrow_{\mathcal{R}} S \Rightarrow_{\mathcal{R}} T_2$ is strongly joinable, then there exists a pair of E-isomorphic graphs X_1 and X_2 such that X_i is a persistent reduct derivable from T_i, $i = 1, 2$. Moreover, the isomorphism commutes on the persistent items of the critical pair in the sense of Definition 3.*

However, it is not enough to nondeterministically compute a pair of reducts and then compare them for strong joinability. Instead, one needs to consider *all* reducts. Consider the terminating set of rules in Figure 8. This system is non-confluent because of the derivations $A \Leftarrow^* S \Rightarrow^* D \overset{r_4}{\Rightarrow} \bullet$, which are two non-isomorphic normal forms. However, a confluence checker needs to search for strong joinability of critical pairs first. A strongly joinable critical pair $T_1 \Leftarrow_{\mathcal{R}} S \Rightarrow_{\mathcal{R}} T_2$ is given. All the nodes of S are persistent nodes, and there are no persistent edges. The graphs T_1 and T_2 have multiple persistent reducts - T_1 reduces to A and B while T_2 reduces to C and D. The isomorphism $A \cong C$ demonstrates strong joinability, $B \cong D$ but violates the strictness condition, $A \ncong D$ and $B \ncong C$, thus a confluence checker needs to compare all persistent reducts for isomorphism.

Input : A terminating set of left-linear rules $\mathcal{R}$ with a set of critical pairs CP

```
1  foreach cp = (T_1 ⇐_R S ⇒_R T_2) in CP do
2      for i = 1, 2 do
3          construct all derivations T_i ⇒*_R X_i where X_i is a persistent reduct
4          {let PR_i be the set of all persistent reducts X_i}
5      end
6      foreach pair of graphs (A, B) in PR_1 × PR_2 do
7          if there exists a strong isomorphism A → B then
8              mark cp as strongly joinable
9          end
10     end
11 end
12 if all critical pairs in CP are strongly joinable then
13     return "confluent"
14 else
15     return "unknown"
16 end
```

Algorithm 1: Confluence Analysis Algorithm

Confluence Algorithm. Given a set of symbolic critical pairs and a terminating rewrite relation $\mathcal{R}$, Algorithm 1 checks whether all symbolic critical pairs are strongly joinable (Definition 3) by computing persistent reducts and then checking for isomorphisms (that are compatible with the joining derivations according to Definition 3). If that is the case, then the symbolic critical pair is strongly joinable. It is sufficient to consider persistent reducts due to Proposition 1. If all critical pairs are found to be strongly joinable, the algorithm reports $\mathcal{R}$ to be confluent. Otherwise, it reports "unknown".

Isomorphism checking is an integral part of joinability analysis. Since at the host graph level every label is taken from the concrete GP 2 label algebra without variables, checking for isomorphism ($\cong$) is decidable. However, when analysing graphs at the symbolic level, the problem of E-isomorphism ($\cong_E$) involves deciding validity of equations in Peano arithmetic. To the best of our knowledge, the problem is open for pure equations (no negation). Nevertheless, decidable fragments exist such as Presburger Arithmetic, whose decision procedures can be used during the analysis of the shortest distances case study (shown in [11]).

Note that the confluence algorithm does not determine non-confluence. This is due to the limitations of symbolic rewriting: not every host graph derivation can be represented by a symbolic derivation. However, in certain cases it is possible to use a combination of unification and satisfiability checking to determine that two non-isomorphic persistent reducts represent non-isomorphic normal forms at the host graph level. In these cases the algorithm could report non-confluence, which is a topic of ongoing work.

Related work. It is important to stress the differences with the symbolic approach of [14] which also defines symbolic critical pairs. That paper is in the context of symbolic graph transformation [16] where whole classes of attributed graphs are transformed via symbolic rules (rules equipped with first-order logical for-

mulas). Symbolic critical pairs represent conflicts between such symbolic rules. However, no construction algorithm is given for these critical pairs. In fact, that paper treats attribute algebras as parametric, and thus a general construction algorithm is an undecidable problem. Joinability and local confluence are not considered. Symbolic rewriting is used to check critical pairs for strong confluence (joinability with 1/0-length derivations), which serves as an inspiration for validity checking in our case study.

The differences with critical pairs in the attributed setting of [3] are similar to the above. In this setting, graph attributes are represented via special *data* nodes and linked to ordinary graph items via attribution edges, giving rise to infinite graphs. The critical pair construction however is restricted to rules whose attributes are variables or variable-free. The algorithmic aspects of confluence analysis are not considered.

7 Conclusion

We have presented a method for statically verifying confluence (functional behaviour) of terminating sets of GP 2 rules, based on constructing the symbolic critical pairs of a rule set and checking that all pairs are strongly joinable with symbolic derivations. The correctness of this method is a consequence of the main technical result, namely that a set of left-linear attributed rules is locally confluent if all symbolic critical pairs are strongly joinable. We have also shown it is sufficient to focus on the persistent reducts when checking strong joinability. In a case study, we used our method to verify the confluence of a graph program that calculates shortest distances.

An interesting topic of future work is the extension of confluence analysis to handle GP 2 program constructs other than looping, e.g. conditional branching. Other topics are the practical aspects of joinability analysis, namely developing decision procedures for label equivalence (e.g. see [13]), and the theoretical treatment of conditional rule schemata.

References

1. Bak, C.: GP 2: Efficient Implementation of a Graph Programming Language. Ph.D. thesis, University of York (2016), `http://etheses.whiterose.ac.uk/id/eprint/12586`
2. Bang-Jensen, J., Gutin, G.: Digraphs: Theory, Algorithms and Applications. Springer, second edn. (2009), doi:10.1007/978-1-84800-998-1
3. 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
4. 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

5. Ehrig, H., Habel, A., Padberg, J., Prange, U.: Adhesive high-level replacement categories and systems. In: Proc. International Conference on Graph Transformation (ICGT 2004). LNCS, vol. 3256, pp. 144–160. Springer-Verlag (2004), doi:10.1007/978-3-540-30203-2_12
6. Habel, A., Plump, D.: Relabelling in graph transformation. In: Proc. International Conference on Graph Transformation (ICGT 2002). LNCS, vol. 2505, pp. 135–147. Springer-Verlag (2002), doi:10.1007/3-540-45832-8_12
7. 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-Verlag (2012), doi:10.1007/978-3-642-33654-6_15
8. Hristakiev, I.: Confluence Analysis for a Graph Programming Language. Ph.D. thesis, University of York (2017), to appear
9. 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), doi:10.14279/tuj.eceasst.71.1002
10. Hristakiev, I., Plump, D.: Attributed graph transformation via rule schemata: Church-Rosser theorem. In: Software Technologies: Applications and Foundations – STAF 2016 Collocated Workshops, Revised Selected Papers. LNCS, vol. 9946, pp. 145–160. Springer (2016), doi:10.1007/978-3-319-50230-4_11
11. Hristakiev, I., Plump, D.: Checking graph programs for confluence (long version) (2017), `https://www.cs.york.ac.uk/plasma/publications/pdf/HristakievPlump.GCM17.Long.pdf`
12. Hristakiev, I., Plump, D.: Towards critical pair analysis for the graph programming language GP 2. In: Recent Trends in Algebraic Development Techniques (WADT 2016), Revised Selected Papers. LNCS, vol. 10644. Springer (2017), to appear, long version available at: `https://www.cs.york.ac.uk/plasma/publications/pdf/HristakievPlump.WADT16.Long.pdf`
13. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View. Texts in Theoretical Computer Science, Springer, second edn. (2016), doi:10.1007/978-3-662-50497-0
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). Electronic Proceedings in Theoretical Computer Science, vol. 181, pp. 97–112 (2015), doi:10.4204/EPTCS.181.7
15. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008). LNCS, vol. 4963, pp. 337–340. Springer (2008), `https://github.com/Z3Prover/z3`, doi:10.1007/978-3-540-78800-3_24
16. Orejas, F., Lambers, L.: Lazy graph transformation. Fundamenta Informaticae 118(1-2), 65–96 (2012), doi:10.3233/FI-2012-706
17. 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
18. Plump, D.: The design of GP 2. In: Proc. International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011). Electronic Proceedings in Theoretical Computer Science, vol. 82, pp. 1–16 (2012), doi:10.4204/EPTCS.82.1