arXiv:2003.06488v3 [cs.LO] 19 Aug 2020

Patch Graph Rewriting (Extended Version)*

Roy Overbeek✉ and Jörg Endrullis

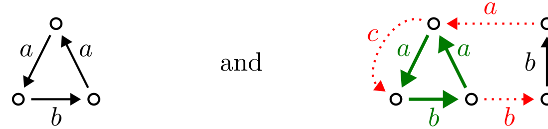
Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
{r.overbeek, j.endrullis}@vu.nl

Abstract. The basic principle of graph rewriting is the stepwise replacement of subgraphs inside a host graph. A challenge in such replacement steps is the treatment of the *patch graph*, consisting of those edges of the host graph that touch the subgraph, but are not part of it.
We introduce *patch graph rewriting*, a visual graph rewriting language with precise formal semantics. The language has rich expressive power in two ways. First, rewrite rules can flexibly constrain the permitted shapes of patches touching matching subgraphs. Second, rules can freely transform patches. We highlight the framework's distinguishing features by comparing it against existing approaches.

Keywords: Graph rewriting · Embedding · Visual language

1 Introduction

When matching a graph pattern P inside a host graph G, G can be partitioned into (i) a *match* M, a subgraph of G isomorphic to the pattern P; (ii) a *context* C, the largest subgraph of G disjoint from M; and (iii) a *patch* J, the graph consisting of the edges that are neither in M nor in C. So the patch consists of edges that are either (a) between M and C, in either direction, or (b) between vertices of M not captured by the pattern P. For example, if P and G are respectively



then the thick green subgraph is the (only) match M of P in G. The black subgraph of G is the context C, and the dotted red subgraph is the patch J. Metaphorically, patch J patches match M and context C together.

In graph rewriting, subgraphs of some host graph are stepwise replaced by other subgraphs. A requirement for such replacements is that they are properly re-embedded in the host graph. We contend that the patch is the most

* The present version extends the submission accepted to ICGT20 with an appendix. The final authenticated publication is available online at https://doi.org/10.1007/978-3-030-51372-6_8.

distinctive and interesting aspect of graph rewriting. This is because its shape is generally unpredictable, making it challenging to specify what constitutes a proper re-embedding of a subgraph replacement. This contrasts strongly with the situation for string and term rewriting, in which the embeddings of substrings and subterms are highly regular.

Most existing approaches to graph rewriting are rather uniform and coarse-grained in their treatment of the patch. For instance, suppose that we wish to delete the match M from G. What should happen to the edges of patch J, which would be left "dangling" by such a removal? The popular double-pushout (DPO) [9] approach to graph rewriting conservatively dictates that the application is not allowed in the first place: nodes connected to the patch *must* be preserved by the rewrite step, and the patch shall remain connected as before. The single-pushout (SPO) [18] variant, by contrast, permissively answers that such a deletion is always possible. As a side-effect, however, any resulting dangling patch edges are discarded.

In this paper, we introduce the *patch graph rewriting* (*PGR*) language. It has the following features:

- *Pluriform, fine-grained control over patches.* Rules themselves encode which kinds of patches are allowed around matches, as well as how they should be transformed for the re-embedding, using a *unified* notation. Thus, these policies are distinctly not decided on the level of the framework.
- *An intuitive visual language.* Despite their expressive power and formal semantics, patch rewrite rules admit a visual representation that we believe to be highly intuitive.
- *Lightweight formal semantics.* The formal details of PGR are based on elementary set and graph theory, and therefore accessible to a wide audience. In particular, an understanding of category theory is not required to understand these details, unlike for many dominant approaches in graph rewriting.

The remainder of our paper is structured as follows. To fix ideas and emphasize the visual language of PGR, we first provide an intuitive exposition in Section 2, and then follow with a formal introduction in Section 3. We show the usefulness of PGR by modeling wait-for graphs and deadlock detection in Section 4. We compare PGR to other approaches in Section 5. In Section 6, finally, we mention some future research directions for PGR.

2 Intuitive Semantics

We start with an intuitive introduction of PGR, to be made formally precise in Section 3. The graph G in Figure 1 will serve as our leading example.

We begin by considering the rewrite rule in Figure 2. Figure 3 contains a depiction of G in which the match, context and patch are highlighted: the thick green subgraph is the match for the left-hand side of the rule, the solid black subgraph is the context for this rule application, and the red dotted edges form the patch. In PGR, the rewrite rule in Figure 2 cannot yet be applied in G. This

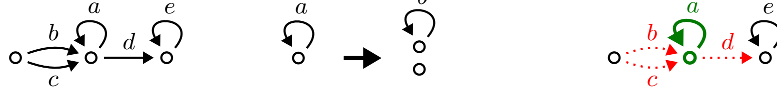


Fig. 1. Graph G. **Fig. 2.** A simple rule. **Fig. 3.** Match, context and patch.

is because without further annotations, the rule may only be applied if the patch is empty, that is, if the node with the a-loop has no additional edges. In effect, this rule only allows replacing an isolated node with an a-loop by two isolated nodes, one of which has a b-loop.

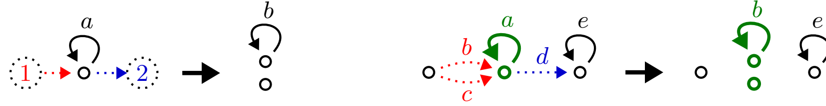


Fig. 4. An annotated rule. **Fig. 5.** Applying the rule on the left.

The rule in Figure 2 can be generalized to allow for patch edges from and to the context by annotating the left-hand side of the rule as shown in Figure 4. We call such annotations *patch type edges*. They can be thought of as placeholders for sets of patch edges:

(i) The dotted arrow with source (1) is a placeholder for an arbitrary number of edges from the context to the node with the a-loop.
(ii) Likewise, the outgoing dotted arrow with target (2) is a placeholder for an arbitrary number of edges going into the context.

The rule is now applicable to all nodes that have an a-loop and no other loop, allowing the node to be connected to the context through an arbitrary number of edges. In particular, then, the rule is applicable to the match highlighted in Figure 3, and it gives rise to the step shown in Figure 5.

Although we see how patch type edge annotations on the left-hand side can be used to *constrain* the set of permitted patches around a match, it does not tell us what to do with patch edges if a match is found. To indicate such transformations, the solution is simply to reuse the patch type edges in the right-hand side of the rule. The rule shown in Figure 4 does not reuse any of the patch type edges, explaining why the corresponding patch edges are deleted in Figure 5.

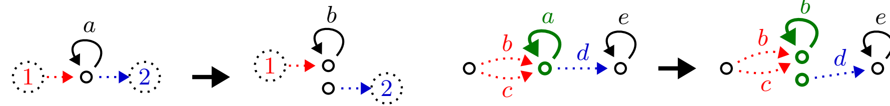


Fig. 6. Redirecting patch edges. **Fig. 7.** Applying the rule on the left.

One way to preserve the incoming edges bound to (1) and the outgoing edges bound to (2) is shown in Figure 6. As the visual representation suggests, the incoming edges bound to (1) get redirected to target the upper node of

the right-hand side, and the sources of the outgoing edges bound to 2 are redirected to the lower node. The respective sources and targets of the edges are defined to remain unchanged. Applying the rule in G results in the rewrite step depicted in Figure 7.

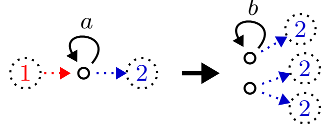


Fig. 8. Duplicating patch edges.

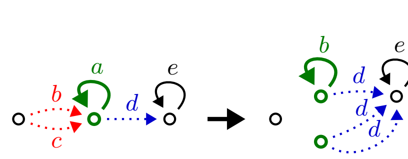


Fig. 9. Applying the rule on the left.

Using this visual language, it is easy to duplicate, remove, and redirect edges in the patch. The rule displayed in Figure 8 removes the incoming patch edges bound to 1, and duplicates the patch edges bound to 2: one copy for the upper node of the right-hand side, and two copies for the lower node. The resulting rewrite step is shown in Figure 9.

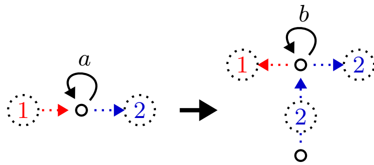


Fig. 10. Complex transformation.

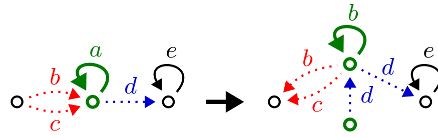


Fig. 11. Applying the rule on the left.

Patch graph rewriting also allows for some more exotic transformations, such as inverting patch edges or pulling edges from the context into the pattern. The rule in Figure 10 reverses the direction of 1 and pulls 2 into the pattern, giving rise to the step in Figure 11.

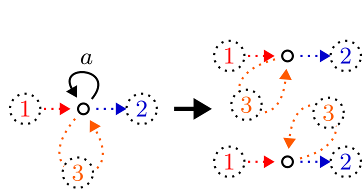


Fig. 12. Node duplication.

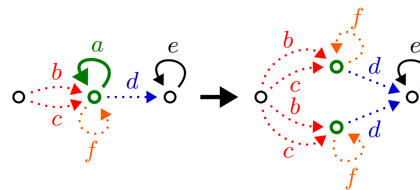


Fig. 13. Applying the rule on the left.

All of the above rules are only applicable to nodes that have an a-loop and no other loop. If we want the rule to be applicable to nodes that have additional loops, this can be expressed as in Figure 12. This rule is now applicable to any node with an a-loop. It makes a copy of the node, as well as all edges incident to it (except for the a-loop, which is removed). If we slightly modify G to include an f-loop on the middle node, the rule gives rise to the rewrite step in Figure 13.

In this brief visual introduction, we have focused on the transformation of the patch. The left-hand sides of the rules has each time been a single

node with an a-loop. Its generalization to other patterns is largely obvious, but some edge cases arise. For instance, what could be the semantics of the rule which redirects patch edges between nodes of the pattern into the context? We now turn to the formal semantics of path rewriting, which makes all preceding transformations precise and excludes such edge cases.

3 Formal Semantics

Notation 1 (Preliminaries). For functions $f : A_f \to B_f$ and $g : A_g \to B_g$ with disjoint domains (but possibly overlapping codomains), we write $f \cup g$ for the function $(f \cup g) : (A_f \cup A_g) \to (B_f \cup B_g)$ given by the union of f and g's underlying graphs. If typing permits, we generalize functions f to tuples (x, y) and sets S in the obvious way, i.e., $f((x, y)) = (f(x), f(y))$ and $f(S) = \{ f(x) \mid x \in S \}$.

We define directed, edge-labeled multigraphs in the standard way.

Definition 2 (Graph). *A* graph $G = (V, E, s, t, \ell)$ *with edge labels from* L *consists of a finite set of* vertices *(or* nodes*)* V*, a finite set of edges* E*, a* source map $s : E \to V$*, a* target map $t : E \to V$*, and a* labeling $\ell : E \to L$*. For* $e \in E$*, we say that* $s(e)$*,* $t(e)$ *and* $\ell(e)$ *are the* source*,* target *and* label *of* e*, respectively.*

For convenience, we will write $x \xrightarrow{\alpha} y \in E$ *to denote an edge* $e \in E$ *such that* $s(e) = x$*,* $t(e) = y$ *and* $\ell(e) = \alpha$*.*

We depict graphs as usual. An explanation may be found in Appendix A. A discussion on how to encode vertex labels as edge labels is found in Appendix B.

Definition 3 (Basic Graph Notions). *We define the following basic graph notions:*

(i) An unlabeled graph $G = (V, E, s, t)$ *is a graph* (V, E, s, t, ℓ) *over a singleton label set. In this case we suppress the edge labels.*
(ii) A graph is simple *if for all* $e, e' \in E$*,* $s(e) = s(e')$*,* $t(e) = t(e')$ *and* $\ell(e) = \ell(e')$ *together imply* $e = e'$*.*
(iii) We say that graphs G *and* H *are* disjoint *if* $V_G \cap V_H = \varnothing = E_G \cap E_H$*.*
(iv) For disjoint edge sets $E \cap E' = \varnothing$*, we define the* graph union *as follows:*

$$(V, E, s, t, \ell) \cup (V', E', s', t', \ell') \quad = \quad (V \cup V', E \cup E', s \cup s', t \cup t', \ell \cup \ell') \,.$$

To rename vertices and edges of a graph, we introduce "graph renamings". A renaming is a graph isomorphism, where the domain of the renaming is allowed to be a superset of the set of vertices/edges of the graph. In this way, the same renaming can be applied to various graphs with different vertex and edge sets.

Definition 4 (Graph Renaming). *A* graph renaming ϕ *for a graph* G *consists of two bijective functions* $\phi_V : V_1 \to V_2$ *and* $\phi_E : E_1 \to E_2$ *such that* $V_G \subseteq V_1$ *and* $E_G \subseteq E_1$*.*

The ϕ-renaming of G, denoted $\phi(G)$, is the graph (V, E, s, t, ℓ) defined by

$$V = \phi_V(V_G) \qquad s(\phi_E(e)) = \phi_V(s_G(e)) \qquad \ell(\phi_E(e)) = \ell_G(e)$$
$$E = \phi_E(E_G) \qquad t(\phi_E(e)) = \phi_V(t_G(e))$$

for every $e \in E_G$.

Definition 5 (Graph Isomorphism). *Graphs G and H are* isomorphic*, denoted $G \approx H$, if there is a graph renaming ϕ such that $H = \phi(G)$.*

Let L be a finite nonempty set of labels. In the sequel, we tacitly assume that all graphs have labels from L.

As motivated by the preceding sections, we allow to compose a context graph C and a match graph M by a "patch" J that may add edges between the nodes of C and M, as well as between the nodes of M.

Definition 6 (Patch). *Let C and M be disjoint graphs. A* patch *for C and M is a graph J such that $E_J \cap (E_C \cup E_M) = \varnothing$ and $V_J = s(E_J) \cup t(E_J)$, and*

$$(s(e), t(e)) \in (V_C \times V_M) \cup (V_M \times V_C) \cup (V_M \times V_M)$$

for every edge $e \in E_J$. In this relation mediated by J, we call C the context graph *and M the* match graph*.*

Definition 7 (Patch Composition). *Let J be a patch for a context graph C and a match graph M. The* patch composition *of C and M through* patch *J, denoted by $C \cdot_J M$, is the graph union $C \cup J \cup M$.*

Example 8. Consider the following graphs C, M and G, respectively:

The composition of C and M through patch $J = \{\, 2 \xrightarrow{a} 3,\ 6 \xrightarrow{b} 2,\ 4 \xrightarrow{b} 5,\ 4 \xrightarrow{b} 6 \,\}$ is G, in which C functions as the context graph and M functions as the match graph (w.r.t. J).

Before we consider the formal definition of rewriting, let us discuss the basic principle and motivate some of the design choices. As a first approximation, a graph rewrite rule $L \to R$ is a pair of graphs that behave like patterns. Since the edge and vertex identities in such rules are arbitrary (not to be confused with the edge labeling), we close the rule under isomorphism. The rule should also be applicable in contexts in which a patch connects a context and the pattern of the rule. The rule $L \to R$ thus give rise to rewrite steps of the form $C \cdot_J L' \to C \cdot_{J'} R'$ for graphs C, valid patches J, J' and graphs $L' \approx L$ and $R' \approx R$.

Additionally, we would like to exert control over the shape of patches in two ways. A graph rewriting rule should enable one to (a) constrain the choices for the patch J, and (b) define the patch J' in terms of J. For these purposes, we introduce the concepts of a patch type and a scheme.

Definition 9 (Patch Type). *A* patch type *T for a graph G is an unlabeled patch for G and the trivial graph with node set $\{\square\}$. Here, the trivial graph functions as the context graph.*

Let J be a patch for a context graph C and match graph M, and T a patch type for M. A patch edge $(j_s \xrightarrow{\alpha} j_t) \in E_J$ (α any label) adheres to *a patch type edge $(t_s \to t_t) \in E_T$ if the following conditions hold:*

$$j_s \in V_C \Rightarrow t_s = \square \qquad j_s \in V_M \Rightarrow j_s = t_s$$
$$j_t \in V_C \Rightarrow t_t = \square \qquad j_t \in V_M \Rightarrow j_t = t_t$$

The patch J adheres to *patch type T if there exists an* adherence map *from J to T, i.e., a function $f : E_J \to E_T$ such that e adheres to $f(e)$ for every $e \in E_J$.*

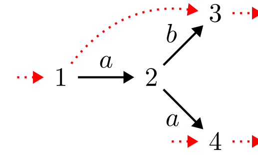
The restriction to unlabeled patch type edges is motivated purely by simplicity. We intend to relax the definition in future work.

Proposition 10 (Unique Adherences). *Let the patch type T be a simple graph. If a patch J adheres to T, then the witnessing adherence map is unique.*

Intuitively, we use patch types to annotate the patterns of a rewrite rule. The result we call a scheme.

Definition 11 (Scheme). *A* scheme *is a pair (P, T) consisting of a graph P, called a* pattern, *and a patch type T for P.*

Example 12 (Depicting Schemes). We extend the representation for graphs to schemes (P, T) as shown on the right. The pattern P consists of the solid labeled arrows, and the patch type T consists of the dotted arrows. For dotted arrows without a source (or target), the source (or target) is implicitly the context graph node $\square$. So T consists of the edges $\{\square \to 1,\ 3 \to \square,\ \square \to 4,\ 4 \to \square,\ 1 \to 3\}$.



We are now ready to define a graph rewrite rule as a relation between schemes (P_L, T_L) and (P_R, T_R). We equip the rewrite rule with a "trace function" τ that relates edges in T_R back to edges in T_L, allowing us to interpret T_R as a transformation of T_L, in which patch edges may be freely moved, deleted, duplicated and inverted. For this we require the following constraint: if a patch type edge $e \in E_{T_R}$ connects to the context, the corresponding edge $\tau(e) \in E_{T_L}$ must also connect to the context. Without this constraint, it would not be clear how to interpret e's connection to the context.

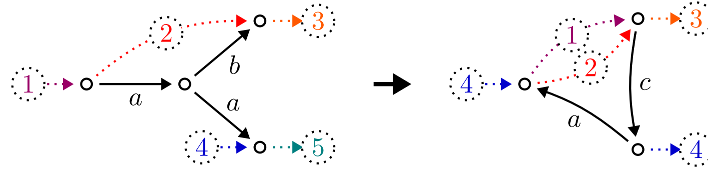
Definition 13 (Quasi Patch Graph Rewrite Rule). *A* quasi patch graph rewrite rule *$L \xrightarrow{\tau} R$ is a pair of schemes $L = (P_L, T_L)$ and $R = (P_R, T_R)$, equipped with a* trace function *$\tau : E_{T_R} \to E_{T_L}$ that satisfies $\square \in \{s(e), t(e)\} \implies \square \in \{s(\tau(e)), t(\tau(e))\}$ for all $e \in E_{T_R}$.*

We normally require the left patch type T_L to be simple, so that the edges of T_L-adherent patches J adhere to a single edge in T_L (Proposition 10). As we shall see, this allows us to define a graph rewrite relation in which matches of a rule produce a unique result (modulo $\approx$).

Definition 14 (Patch Graph Rewrite Rule). *A* patch graph rewrite rule *is a quasi patch graph rewrite rule $(P_L, T_L) \xrightarrow{\tau} (P_R, T_R)$ in which T_L is simple.*

Since we restrict attention to unlabeled patch type graphs in this paper, we will use the opportunity to visualize the trace function τ by means of labels on patch type edges.

Example 15 (Depicting Rules). A depiction of a valid rewrite rule is:



The trace function τ is visualized by means of labels on the type edges: τ maps type edges with label n on the right-hand side to the type edge with label n on the left-hand side. *Throughout the paper, colors are merely used as a supplementary visual aid.* (An application example will be given in Example 19.)

Definition 16 (Rule Isomorphism). *(Quasi) rewrite rules $L_1 \xrightarrow{\tau_1} R_1$ and $L_2 \xrightarrow{\tau_2} R_2$ are* isomorphic*, denoted $L_1 \xrightarrow{\tau_1} R_1 \approx L_2 \xrightarrow{\tau_2} R_2$, if there exists a graph renaming ϕ such that $\phi_V(\Box) = \Box$, $\phi((L_1, R_1)) = (L_2, R_2)$, and $\phi_E \circ \tau_1 = \tau_2 \circ \phi_E$.*

Definition 17 (Patch Graph Rewrite System). *A* (quasi) patch graph rewrite system *(*PGR*) $\mathcal{R}$ is a set of (quasi) rewrite rules. For $\mathcal{R}$ we define the isomorphism closure class $\mathcal{R}^{\approx} = \{ y \mid x \in \mathcal{R},\ y \approx x \}$.*

For a patch J, patch type T and adherence map $h : E_J \to E_T$, we define

$$ctx(e, h) = \begin{cases} \{ s(e) \} & \text{if } s(h(e)) = \Box \\ \{ t(e) \} & \text{if } t(h(e)) = \Box \\ \varnothing & \text{otherwise} \end{cases}$$

for every $e \in E_J$. So $ctx(e, h)$ contains the context node involved in the edge e, or is $\varnothing$ if the edge does not involve the context.

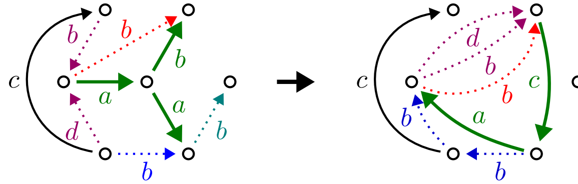
Definition 18 (Patch Graph Rewriting). *A (quasi) patch graph rewrite system $\mathcal{R}$ induces a rewrite relation $\to_{\mathcal{R}}$ on the set of graphs as follows:*
$C \cdot_J P_L \ \to_{\mathcal{R}} \ C \cdot_{J'} P_R$ *if*

(i) $(P_L, T_L) \xrightarrow{\tau} (P_R, T_R) \in \mathcal{R}^{\approx}$,
(ii) $h_L : E_J \to E_{T_L}$ *is an adherence map from patch J to patch type T_L,*

(iii) $h_R : E_{J'} \to E_{T_R}$ *is an adherence map from patch* J' *to patch type* T_R*, and*
(iv) *for every* $t \in E_{T_R}$ *there exists a bijection* $\sigma : h_R^{-1}(t) \to h_L^{-1}(\tau(t))$ *such that* $\ell_R(e) = \ell_L(\sigma(e))$ *and* $ctx(e, h_R) \subseteq ctx(\sigma(e), h_L)$ *for every* $e \in h_R^{-1}(t)$*.*

For such a rewrite step, we say that the graph $C \cdot_J P_L$ *contains the* redex P_L*.*

Example 19 (Application Example). The rule given in Example 15 gives rise to the following rewrite step:



In the graph on the left we have highlighted the match (thick green) and the patch (dotted). We have additionally indicated the adherence map of the patch edges by reusing the colors of the rule definition.

We refer to Section 2 for many examples of simple rewrite rules and rewrite steps, and to Appendix C for rewrite rules demonstrating standard graph operations such as merging, copying and splitting nodes. A graph rewrite system modeling wait-for graphs will be given in Section 4.

Remark 20 (Finding Redexes). Checking for the presence of a redex is simple. A graph G contains a redex with respect to rule $(P_L, T_L) \xrightarrow{\tau} (P_R, T_R) \in \mathcal{R}$ if and only if

1. there exists a subgraph M of G isomorphic to P_L, and
2. every edge $e \notin E_M$ incident to a $v \in V_M$ adheres to an edge in T_L.

Definition 18 can be understood in more operational terms as follows.

Lemma 21 (Constructing J'). *If conditions (i) and (ii) of Definition 18 are satisfied (fixing some adherence map* h_L*), the patch* J' *and adherence map* h_R *that satisfy conditions (iii) and (iv) are uniquely determined up to isomorphism. The patch* J' *can be constructed using the following procedure.*

For every type edge $t = (t_s \to t_t) \in E_{T_R}$*, consider every patch edge* $j = (j_s \xrightarrow{\alpha} j_t) \in E_J$ *for which* $h_L(j) = \tau(t) = (t_s^\tau \to t_t^\tau) \in E_{T_L}$*. There are five exclusive cases:*

1. *If* $\square \notin \{t_s, t_t\}$*, add a new edge* $t_s \xrightarrow{\alpha} t_t$ *to* J'*.*
2. *If* $t_s = t_s^\tau = \square$*, add a new edge* $j_s \xrightarrow{\alpha} t_t$ *to* J'*.*
3. *If* $t_t = t_t^\tau = \square$*, add a new edge* $t_s \xrightarrow{\alpha} j_t$ *to* J'*.*
4. *If* $t_s = t_t^\tau = \square$*, add a new edge* $j_t \xrightarrow{\alpha} t_t$ *to* J'*.*
5. *If* $t_t = t_s^\tau = \square$*, add a new edge* $t_s \xrightarrow{\alpha} j_s$ *to* J'*.*

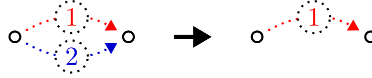
Here, the "new" edge j' is an edge not in C, P_R or the intermediate construction of J'. The adherence map h_R is defined such that $h_R(j') = t$ for each of the considered j' and t.

Non-quasi rules have the following desirable property.

Proposition 22 (Rule Determinism). *Let $G = C \cdot_J P_L$. If a rule $(P_L, T_L) \xrightarrow{\tau} (P_R, T_R) \in \mathcal{R}^{\approx}$ derives both $G \rightarrow_{\mathcal{R}} C \cdot_{J'} P_R = G'$ and $G \rightarrow_{\mathcal{R}} C \cdot_{J''} P_R = G''$, then $G' \approx G''$.*

Proof. This is a direct consequence of Proposition 10 and Lemma 21. $\square$

In contrast to (non-quasi) graph rewrite rules, quasi rules are not generally deterministic. For instance, consider the quasi rewrite rule



which can match any graph G consisting of two nodes x and y and n edges from x to y. For each $e \in E_G$, the left adherence map h_L can either map e to the patch type edge labeled with 1, or to the type edge labeled with 2. Thus, 2^n choices for h_L are possible, and each choice causes a different subset of J to be deleted in a single rewrite step.

Notation 23 (Shorthand Notation). Given a pattern P, we often want to allow for any patch edges between the nodes of a subset $N \subseteq V_P$ as well as the context node $\Box$. In the notation we have discussed so far, we would then need to draw the complete patch type graph induced by $N \cup \{\Box\}$ (minus the loop on $\Box$), which consists of $(|N| + 1)^2 - 1$ patch type edges.

To avoid spaghetti-like figures, we extend the visual presentation of schemes by allowing each node to be annotated with a set of names (written without set brackets). We say that a node *has name* x if x is in the set of names of this node. So a node can have 0 or more names. The name annotations are then shorthand for the following patch type edges:

(i) For every node n and name x of n, the node n has the two patch type edges $(\Box, x) \dashrightarrow n \dashrightarrow (x, \Box)$ from and to the context.

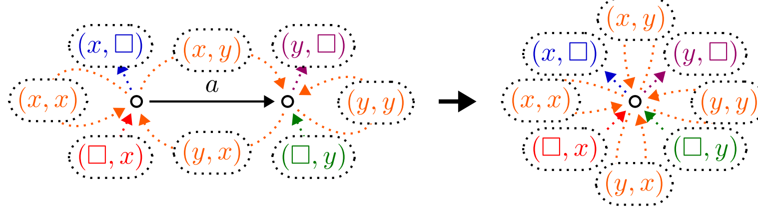
(ii) For every pair of nodes n, m and every name x of n and y of m, there is implicitly the patch type edge $n \cdots (x, y) \dashrightarrow m$ from from n to m. Here n and m can be the same node, and x can be equal to y.

Observe that rules are non-quasi iff every node in the left-hand side has at most one name. We therefore require that distinct nodes do not share names.

As an example, a rule for merging two nodes can be written as

$$\underset{x}{\circ} \xrightarrow{a} \underset{y}{\circ} \quad \rightarrow \quad \underset{x,y}{\circ} \tag{1}$$

which is shorthand for



For a more elaborate shorthand notation, see Appendix D.

4 Modeling Wait-For Graphs and Deadlock Detection

We now give a more extensive and more realistic modeling example that showcases the expressive power of PGR.

A *wait-for graph* [14] is a hypergraph in which nodes represent processes, and hyperedges represent requests for resources. A hyperedge has a single source p, representing the process requesting the resources, and $M > 0$ targets distinct from p, representing the processes from which a resource is requested. The process p requires $0 < N \leq M$ of these resources. Thus, for a fixed M, there are multiple types of hyperedges, each representing a particular N-out-of-M request. Processes can have at most one outgoing N-out-of-M request.

The following distributed system behavior is associated with wait-for graphs. A process without an outgoing request is said to be *unblocked*. An unblocked process can *grant* an incoming request, deleting the edge, or create a new N-out-of-M request. A process becomes unblocked when its pending N-out-of-M request is *resolved*, i.e., when N targeted processes have granted the request.

In order to better illustrate some of PGR's transformational power, we introduce one additional, noncanonical behavior. We consider a process p *overloaded* when it has $n > 2$ incoming requests. When p is overloaded, a clone of p, $c(p)$, may be created which takes over $n - 2$ of p's incoming requests. Because we assume that any outgoing request must be resolved before any incoming requests can be resolved, $c(p)$ replicates p's outgoing request, if p has one.

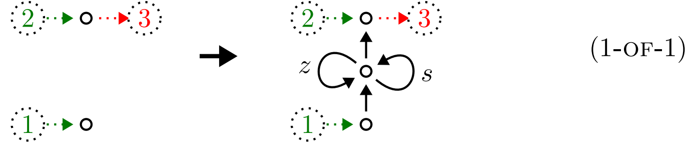
We first define a graph grammar that defines the class of valid wait-for graphs. Then, we will show how to augment the rule set in order to model the distributed system behavior. Finally, we explain how deadlocks can be detected. Throughout, we encode hypergraphs as multigraphs. Note that in this encoding, vertices representing processes are always free of loops, while vertices representing hyperedges always have loops. Hence, the given rules can appropriately discriminate between the two types of vertices.

4.1 Wait-For Graph Grammar

The starting graph of the grammar is the empty graph, denoted by $\varnothing$. Rule

$$\varnothing \rightarrow \circ \qquad \text{(CREATE)}$$

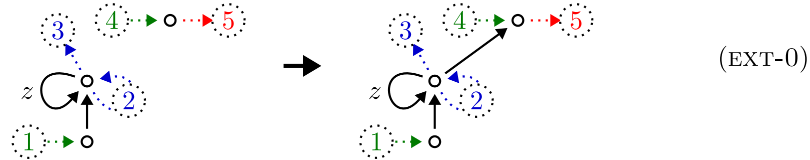
models process creation, and rule



(1-OF-1)

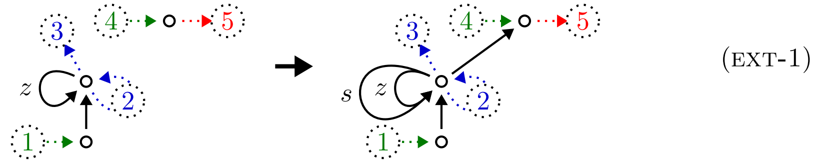
allows constructing a valid 1-out-of-1 request between nodes. Labels z and s should be interpreted as 0 and the successor function, respectively, so that n s-loops encode that n requests are yet to be granted.

In the grammar, any N-out-of-M request can be extended to a valid N-out-of-$(M+1)$ request using rule



(EXT-0)

and to a valid $(N+1)$-out-of-$(M+1)$ request using rule



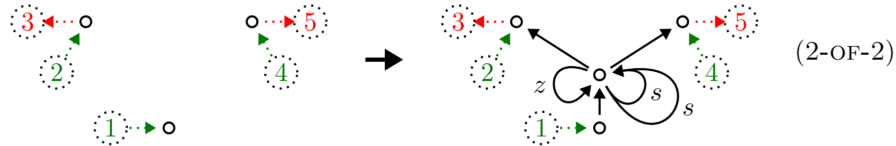
(EXT-1)

These four rules suffice for generating any valid wait-for graph.

4.2 System Modeling

To model a distributed system, we need rule CREATE for process creation, as well as its inverse, DESTROY, for process destruction. Note that DESTROY constrains the process selected for destruction to be isolated in our framework (i.e., it is not associated with any pending requests), as desired.

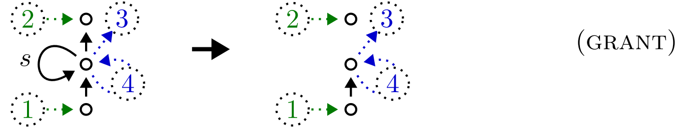
Any N-out-of-M request is understood to be an atomic action. So for, e.g., modeling 2-out-of-2 requests, we need the rule



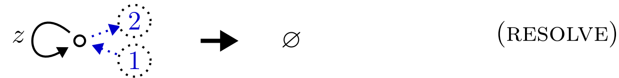
(2-OF-2)

Such rules can be easily simulated by a contiguous sequence of rewrite steps 1-OF-1 $\cdot$ EXT-0* $\cdot$ EXT-1*, in which the node making the request remains fixed. We omit the details.

A grant request may be modeled by

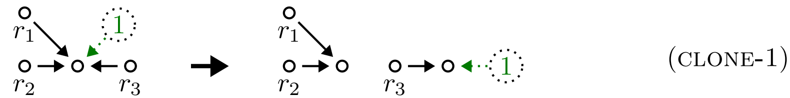


and a request resolution by

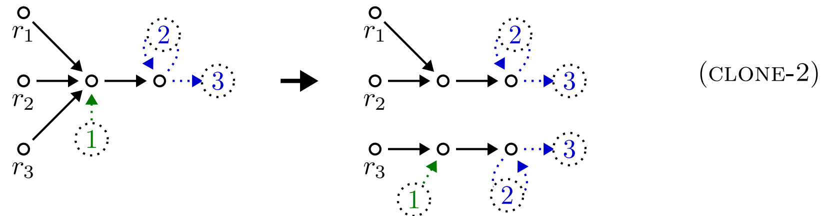


This leaves only the cloning behavior for an overloaded process p. This requires two rules: one for the case in which p is unblocked, and one for the case in which p is blocked. We use the shorthand notation introduced in Notation 23, so that named nodes r_i induce type edges among themselves and from and into the context.[1]

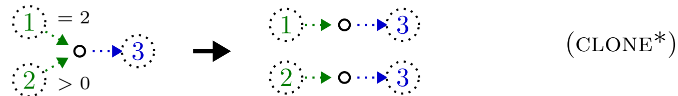
The case in which p is unblocked is modeled by rule



and the case in which p is blocked is modeled by rule



Cloning would be easier to express if PGR were to be extended with support for hyperedges and cardinality constrained type edges. We envision a rule like



to capture the same cases as rules CLONE-1 and CLONE-2 combined. We leave such an extension to future work. In particular, the precise semantics of hyperedge transformation would have to be determined.

[1] The type edges between distinctly named nodes $r_i \neq r_j$ are redundant in the considered scenario, since we know that these type edges will never have adherents.

4.3 Deadlock Detection

Deadlock detection on some valid wait-for graph G can be performed by restricting the rewrite system to rules GRANT, RESOLVE and DESTROY, yielding a terminating rewrite system. Then the network represented by G contains a deadlock if and only if the (unique) normal form of G is the empty graph $\varnothing$.

For a comparable modeling example, see Appendix E, in which the Dijkstra–Scholten termination detection algorithm is modeled.

5 Comparison

We compare PGR to several other rewriting frameworks. We have selected these frameworks because of their popularity and/or because they bear certain similarities to our approach.

Double-Pushout (DPO). Ehrig et al.'s double-pushout approach (DPO) [9] is one of the most prominent approaches in graph rewriting.

A rewrite rule in the DPO approach is of the form $L \hookleftarrow K \to R$, where L is the subgraph to be replaced by subgraph R. The graph K is an "interface", used to identify a part of L with a part of R, and it can be thought of as describing which part of L is preserved by the rule. The identification is formally established through the inclusion $L \hookleftarrow K$ and the graph morphism $K \to R$. The morphism $K \to R$ may be non-injective, allowing it to merge nodes that are in the interface.

A DPO rewrite rule $L \overset{\varphi}{\hookleftarrow} K \overset{\psi}{\to} R$ is applied inside a graph G as follows [7,8]. Let $m : L \to G$ be a graph morphism, which we may assume to be injective [16]. The graph $m(L) \approx L$ is said to be a *match* for L. The arising rewrite step replaces $m(L)$ of G by a fresh copy $c(R)$ of R, redirecting edges left dangling by the removal of a $v \in m(L)$ to node $c(\psi(\varphi^{-1}(m^{-1}(v))))$. For the redirection of edges to work, nodes that leave dangling ends need to be part of the interface, that is, in $m(\varphi(K))$. This is known as the "gluing condition".[2] If the gluing condition is not met, the rewrite step is not permitted.

Using Notation 23, it is easy to see that PGR at least as expressive as DPO with respect to the generated rewrite relation. A DPO rule $L \overset{\varphi}{\hookleftarrow} K \overset{\psi}{\to} R$ can be directly simulated by a PGR rule $L \to R$ in which the nodes are annotated with their (set of) names in the interface: $v \in V_L$ is annotated with the names $\varphi^{-1}(v)$, and $v \in V_R$ is annotated with the names $\psi^{-1}(v)$.

However, DPO is stronger in one respect: a DPO rewrite step preserves the subgraph specified in K, whereas a PGR rewrite can be thought of as destroying and replacing the left-hand side of the rule. The consequences for metaproperties relating to parallelism and concurrency will have to be investigated.

Generalized DPO. In some variants of DPO, the inclusion $L \hookleftarrow K$ is generalized to a (possibly non-injective) morphism $\varphi : K \to L$. Intuitively, this allows a

[2] By the injectivity assumption for m, we need not consider what is known as the "identification condition".

node v of L to be "split apart" in the interface K. Applying the DPO method to a host graph ensures that the patch graph edges incident to v will be incident to one of v's split copies. It does not dictate how these edges should be distributed. Thus, such rewrite steps are non-deterministic.

Generalized DPO rules $L \overset{\varphi}{\leftarrow} K \overset{\psi}{\rightarrow} R$ can be translated to PGR rules $L \to R$ in the same way as discussed for DPO. Since φ is no longer required to be injective, nodes $v \in V_L$ can be annotated with multiple names $\varphi^{-1}(v)$, thereby leading to (non-deterministic) quasi rules (Definition 13).See also Example 26.

Single-Pushout (SPO). The single-pushout (SPO) approach by Löwe [18] is the destructive sibling of DPO. It is operationally like DPO, but it drops the gluing condition. Any edges that would become dangling in the host graph are instead removed.

An SPO rule $L \overset{\varphi}{\hookleftarrow} K \overset{\psi}{\rightarrow} R$ can be simulated by a PGR rule $L \to R$ with annotations as discussed above for DPO, except that each node $v \in V_L$, for which $\varphi^{-1}(v)$ is empty, is now annotated by a fresh name. The rewrite step will then delete all patch edges connected to such a node.

DPO Rewriting in Context (DPO-C). The DPO Rewriting in Context (DPO-C) approach by Löwe [19, 20] addresses the issue of non-determinism in generalized variants of DPO, using ingoing and outgoing arrow annotations to dictate how these edges should be distributed over split copies. The visual representation of DPO-C therefore bears some similarity to that of PGR. In addition, absence of arrow annotations also define negative application conditions like in PGR. However, the patch cannot be transformed as freely as in PGR. For instance, see rule (2) below.

AGREE. AGREE [3] and PBPO [4] by Corradini et al. extend DPO with the ability to erase and clone nodes, while also being able to specify how patch edges are distributed among the copies. For this purpose, a "filter" for the edges determines what kind of patch edges are to be dropped. This filter can distinguish different types of edges based on their source, target and label. Thereby AGREE and PBPO subsume mildly restricted versions of DPO, SPO, and other formalisms.

PGR has some features that are not present in AGREE and PBPO. First, in PGR rule applicability can be restricted by conditions on the permitted shape of the patch. Second, PGR allows (almost) arbitrary redirecting, moving and copying of patch edges outside the scope of AGREE and PBPO. For instance,

$$1 \to \circ \to 2 \quad 6 \to \circ \to 7 \quad \Longrightarrow \quad 6 \to \circ \to 2 \quad 1 \to \circ \to 7 \qquad (2)$$

cannot be expressed in the latter frameworks. Also inverting the direction of patch edges, or moving patch edges between nodes of the pattern is not possible in AGREE and PBPO.

On the other hand, AGREE and PBPO capture some transformations that cannot be expressed in PGR. First, AGREE and PBPO can express some global transformations, unlike PGR. Second, the "patch edge filter" in AGREE and

PBPO can distinguish patch edges depending on their label and the "type" of the source/target in the context (here the type is given by some type graph). Both features are not present in PGR as presented in this paper. However, PGR can be extended with constraints on the patch type edges. (For a discussion on how to encode constraints in the present framework, see Appendix F.) We leave the investigation of a suitable constraint language to future work.

Nagl's Approach. Nagl [21] has defined a very powerful graph transformation approach. Rather than identifying "gluing points" for the replacement of L by R in a host graph G (as in the previous approaches), rules are equipped with expressions that describe how R is to be embedded into the remainder graph $G^- = G - L$. An expression can, e.g., specify that an edge must be created from $u \in G^-$ to $v \in R$ if there existed a path (of certain length and containing certain labels) from u to a $w \in L$. Thus, the embedding context may no longer even be local.

While not all of these transformations are supported by PGR, the expressions are arguably much less intuitive than our representation, in which both application conditions and transformations are visualized in a unified manner.

Habel et al.'s Approach. Habel et al. [15] have introduced graph grammars with rule-dependent application conditions that also admit a very intuitive visual representation. These conditions are more powerful than PGR's application conditions, since they can extend arbitrarily far into the context graph. However, transformations are not included in the approach, unlike in PGR, in which the notations for application conditions and transformations are unified.

Drags. To generalize term rewriting techniques to the domain of graphs, Dershowitz and Jouannaud [6] have recently defined the *drag* data structure and framework. A drag is a multigraph in which nodes are labeled by symbols that have an arity equal to the node's outdegree. Nodes labeled with nullary variable symbols are called *sprouts*, and resemble output ports. In addition, the drag comes equipped with a finite number of *roots*, which resemble input ports.

A composition operation $\otimes_\xi$ for drags, parameterized by a two-way *switchboard* ξ identifying sprouts with roots, gives rise to a rewrite relation $W \otimes_\xi L \to W \otimes_\xi R$. For this rewrite relation to be well-defined, it is required, among others, that L and R have the same number of roots and the same multisets of variables.

Since drag rewriting imposes arity restrictions on nodes, it is more restrictive than patch rewriting concerning the shapes of the graphs that can be rewritten. As drag rewrite steps are local, we believe that PGR can simulate them, but we leave this investigation to future work.

6 Conclusion

We have introduced *patch graph rewriting*, a framework for graph rewriting that enriches the rewrite rules with a simple, yet powerful language for constraining and transforming the local embedding—or *patch*.

For future work, we plan to investigate various meta-properties central in graph rewriting [8], in particular confluence [12, 17, 22], termination [2, 5], the concurrency theorem [10], decomposability and reversibility of rules. We intend to study these properties both globally, for all graphs, as well as locally [11, 13], for a given language of graphs [23]. Furthermore, we are interested in extending the framework with constraint labels on patch type edges, and in allowing label transformations. We believe this could be useful for modeling a larger class of distributed algorithms [14]. Another interesting direction of research is an equational perspective on patch rewriting, as similarly investigated by Ariola and Klop for term graph rewriting [1].

Acknowledgments This paper has benefited from discussions with Jan Willem Klop, Nachum Dershowitz, Femke van Raamsdonk, Roel de Vrijer, and Wan Fokkink. We thank Andrea Corradini and the anonymous reviewers for their useful suggestions. Both authors received funding from the Netherlands Organization for Scientific Research (NWO) under the Innovational Research Incentives Scheme Vidi (project. No. VI.Vidi.192.004).

References

1. Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Fundamenta Informaticae*, 26(3, 4):207–240, 1996. `doi:10.3233/FI-1996-263401`.
2. H. J. S. Bruggink, B. König, and H. Zantema. Termination analysis for graph transformation systems. In *Theor. Comput. Sci.*, pages 179–194. Springer, 2014. `doi:10.1007/978-3-662-44602-7_15`.
3. A. Corradini, D. Duval, R. Echahed, F. Prost, and L. Ribeiro. AGREE – algebraic graph rewriting with controlled embedding. In *Proc. Conf. on Graph Transformation, ICGT 2015*, volume 9151 of *LNCS*, pages 35–51. Springer, 2015. `doi:10.1007/978-3-319-21145-9_3`.
4. A. Corradini, D. Duval, R. Echahed, F. Prost, and L. Ribeiro. The PBPO graph transformation approach. *J. Log. Algebr. Meth. Program.*, 103:213–231, 2019. `doi:10.1016/j.jlamp.2018.12.003`.
5. N. Dershowitz and J.-P. Jouannaud. Graph path orderings. In *LPAR*, volume 57 of *EPiC Series in Computing*, pages 307–325. EasyChair, 2018. `doi:10.29007/6hkk`.
6. N. Dershowitz and J.-P. Jouannaud. Drags: A compositional algebraic framework for graph rewriting. *Theor. Comput. Sci.*, 777:204–231, 2019. URL: `https://doi.org/10.1016/j.tcs.2019.01.029`, `doi:10.1016/j.tcs.2019.01.029`.
7. H. Ehrig. Tutorial introduction to the algebraic approach of graph grammars. In *Proc. Workshop on Graph-Grammars and Their Application to Computer Science*, volume 291 of *LNCS*, pages 3–14. Springer, 1986. `doi:10.1007/3-540-18771-5_40`.
8. H. Ehrig, M. Korff, and M. Löwe. Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In *Proc. Workshop on Graph-Grammars and Their Application to Computer Science*, volume 532 of *LNCS*, pages 24–37. Springer, 1990. `doi:10.1007/BFb0017375`.

9. H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: An algebraic approach. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*, page 167180. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.11.
10. H. Ehrig and B. K. Rosen. Parallelism and concurrency of graph manipulations. *Theoretical Computer Science*, 11(3):247–275, 1980. doi:10.1016/0304-3975(80)90016-X.
11. J. Endrullis, R. C. de Vrijer, and J. Waldmann. Local termination: Theory and practice. *Logical Methods in Computer Science*, 6(3), 2010. doi:10.2168/LMCS-6(3:20)2010.
12. J. Endrullis, J. W. Klop, and R. Overbeek. Decreasing diagrams for confluence and commutation. *Logical Methods in Computer Sci.*, Volume 16, Issue 1, 2020. doi:10.23638/LMCS-16(1:23)2020.
13. J. Endrullis and H. Zantema. Proving non-termination by finite automata. In *Proc. Conf. on Rewriting Techniques and Applications (RTA 2015)*, volume 36 of *LIPIcs*, pages 160–176. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.RTA.2015.160.
14. W. Fokkink. *Distributed algorithms: an intuitive approach*. MIT Press, 2014.
15. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundam. Inform.*, 26(3/4):287–313, 1996. doi:10.3233/FI-1996-263404.
16. A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001. doi:10.1017/S0960129501003425.
17. R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In *Graph Transformation*, pages 161–176. Springer, 2002. doi:10.1007/3-540-45832-8_14.
18. M. Löwe. Algebraic approach to single-pushout graph transformation. *Theor. Comput. Sci.*, 109(1&2):181–224, 1993. doi:10.1016/0304-3975(93)90068-5.
19. M. Löwe. Double-pushout rewriting in context. In *Proc. Workshop on Software Technologies: Applications and Foundations*, volume 11176 of *LNCS*, pages 447–462. Springer, 2018. doi:10.1007/978-3-030-04771-9_32.
20. Michael Löwe. Double-pushout rewriting in context – rule composition and parallel independence. In Esther Guerra and Fernando Orejas, editors, *Graph Transformation – 12th International Conference, ICGT 2019*, volume 11629 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 2019. URL: https://doi.org/10.1007/978-3-030-23611-3_2, doi:10.1007/978-3-030-23611-3_2.
21. M. Nagl. Set theoretic approaches to graph grammars. In *Proc. Workshop on Graph-Grammars and Their Application to Computer Science*, volume 291 of *LNCS*, pages 41–54. Springer, 1986. doi:10.1007/3-540-18771-5_43.
22. D. Plump. 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*, pages 280–308. Springer, 2005. doi:10.1007/11601548_16.
23. A. Rensink. Canonical graph shapes. In *European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 401–415. Springer, 2004. doi:10.1007/978-3-540-24725-8_28.

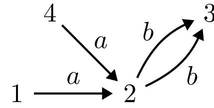
Appendix

A Depicting Graphs

Consider the graph $G = (V, E, s, t, \ell)$ defined by:

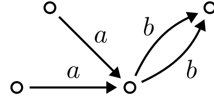
$$\begin{array}{llll} V = \{1,2,3,4\} & s(e_1) = 1 & t(e_1) = 2 & \ell(e_1) = a \\ E = \{e_1, e_2, e_3, e_4\} & s(e_2) = 2 & t(e_2) = 3 & \ell(e_2) = b \\ L = \{a, b\} & s(e_3) = 2 & t(e_3) = 3 & \ell(e_3) = b \\ & s(e_4) = 4 & t(e_4) = 2 & \ell(e_4) = a \end{array}$$

The graph G can be visualized as follows:



The edges are displayed as arrows from the source to the target of the edge, annotated by the label of the edge. In such a visualization names of the edges are typically suppressed; so the graph is defined only up to renaming of the edges.

We also suppress the vertex names if they are irrelevant. Then the graphs are defined up to isomorphism only. For instance,



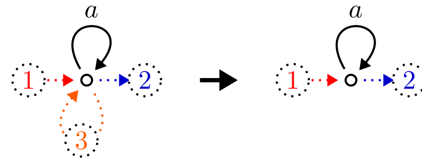
is a visualisation of G where edge and vertex names are suppressed. As a convention, nodes with distinct coordinates are always assumed to be distinct.

B Encoding Vertex Labels

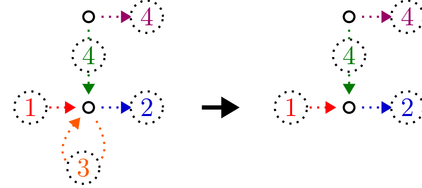
Vertex labels can be encoded in at least two different ways:

(i) Choosing a vertex label set L_V disjoint from edge label set L_E, and adding an edge $v \xrightarrow{\alpha} v$ when v has label α.
(ii) Using a distinguished node r that has precisely one edge $r \xrightarrow{\alpha} v$ to every other node v and α is v's vertex label. Then r is the only node with no incoming edges.

In PGR, approach (ii) has an advantage over (i) when one wants to match all loops of a node with an arbitrary label. Assume that we want to specify a rule that can drop all loops from an arbitrary node. Using approach (i) we need a rule of the form



for every node label a. Using approach (ii) a single rule suffices:

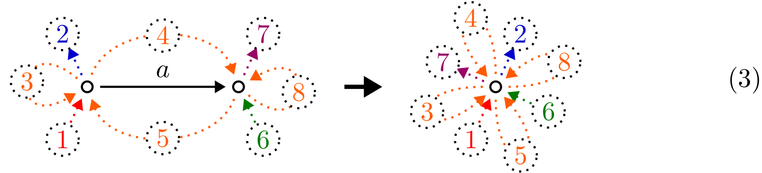


Note that the upper node cannot have incoming patch edges, so this is the node responsible for assigning labels to the other nodes. Thus (4) binds the edge that carries the node label of the lower node.

C Elementary Graph Operations

The following examples demonstrate that PGR easily supports a number of elementary graph operations.

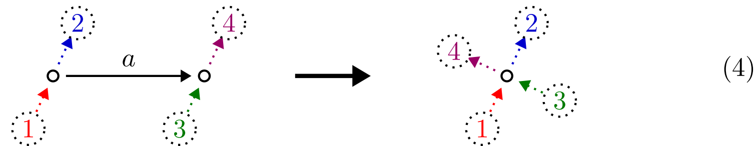
Example 24 (Merging Two Nodes). The following rewrite rule can be applied to any pair of nodes connected by an edge with label a:



(3)

When the rule is applied, the edge with label a is dropped, and the two nodes are merged into a single node. All incoming and outgoing edges are redirected accordingly.

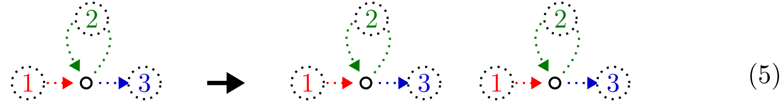
If we want to exclude edges between the nodes of the pattern other than the edge labeled with a, then the rule can be simplified as follows:



(4)

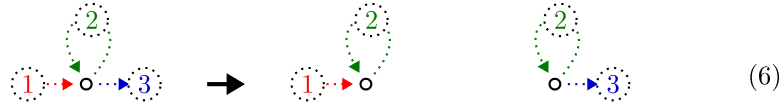
Now the patch can only contain edges between the context and the pattern of the rule.

Example 25 (Copying a Node). The following graph rewrite rule copies including all its edges (from the context, to the context and loops):



(5)

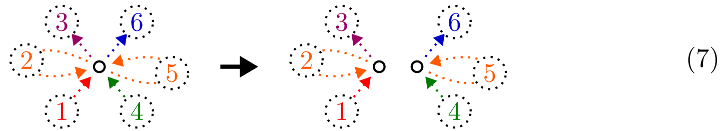
PGR allows a fine-grained control that enables one to do much more than simply copying a node. For instance, consider the rule



(6)

This rule makes a partial copy of a node. It copies the node including all its loops. However, the incoming edges (from the context) and outgoing edges (to the context) are not duplicated, but redistributed between the two copies. All incoming edges are assigned to the left copy, all outgoing edges are assigned to the right copy.

Example 26 (Non-deterministically Splitting a Node). In the preceding example we have seen how to copy including all its incoming and outgoing arrows. We now want to split a node into two nodes and non-deterministically distribute the edges between the two nodes. This can be achieved by the following quasi patch graph rewrite rule:



(7)

The patch type of the left-hand side is not a simple graph. Here

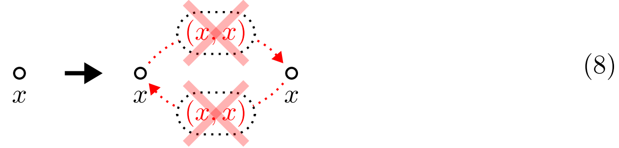
- 1 and 4 form a partitioning of the incoming edges (from the context),
- 3 and 6 form a partitioning of the outgoing edges (into the context).
- 2 and 5 form a partitioning of the loops on the node.

There is no fairness condition imposed here. For instance, the partitions 1, 2, 3 can be empty. Then all edges are assigned to the right node. A fair distribution of the edges could be achieved by extending the patch types with a richer constraint language.

D Extended Shorthand Notation

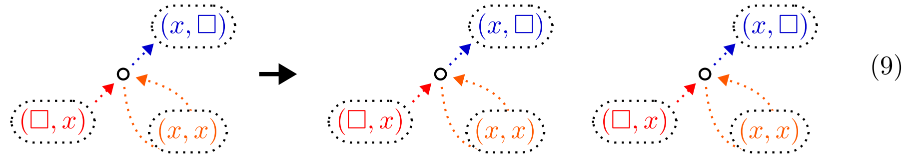
Extending Notation 23, we suggest the following notation to indicate that a certain implicit edge should not be present:

For instance, rule (5) for copying a node can be written as



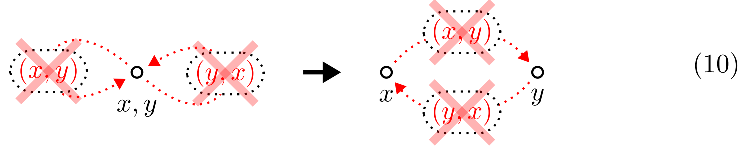
(8)

which is shorthand for



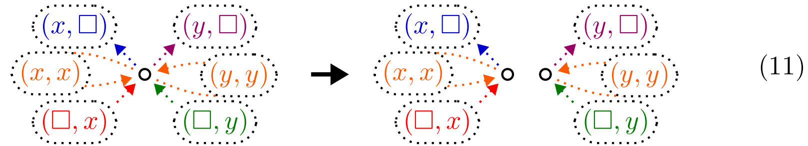
(9)

Rule (7) for non-deterministically splitting a node can now be written as



(10)

which is an abbreviation for



(11)

Compare the left-hand side of this rule with the right-hand side of the example given in rule (1).

E Termination Detection: Dijkstra–Scholten

The Dijkstra–Scholten algorithm [14] detects the termination of a centralized basic algorithm. It is assumed that the basic algorithm is executed on a loop-free undirected network, and that there is a distinguished initiator process. The Dijkstra–Scholten algorithm builds a tree alongside the execution of the basic algorithm, where processes that are active in the basic algorithm are part of the Dijkstra–Scholten tree. Each process maintains a counter, originally 0. A

counter represents a conservative estimate of how many of the process's children are active. When the initiator's counter is 0, it can quit the tree, by which it correctly announces that the algorithm has terminated (i.e., there are no more basic messages in transit, and all processes have quit the tree).

The algorithm can be described as follows:

- The initiator starts as the root of the Dijkstra–Scholten tree T.
- A process $p \in T$ can send a basic message to a neighbour process. When it does, it increments its own counter.
- When a $p \in T$ receives a basic message from a p', it sends p' a control message to indicate that it is already in the tree.
- When a $p \notin T$ receives a basic message from a p', it joins the tree, and stores p' as its parent.
- A non-initiator with a counter of 0 can quit the tree. When it does, it informs its parent that it is no longer its child via a control message.
- When a process receives a control message, it decrements its counter.
- An initiator with a counter of 0 can quit the tree, by which it announces that the basic algorithm has terminated.

In our graph representation modeling the state of a Dijkstra–Scholten tree, we use the following labels on directed edges (assume $u \neq v$):

- An edge $u \xrightarrow{e} v$ denotes a network connection between u and v.
- An edge $u \xrightarrow{b} v$ denotes a basic message in transit between u and v.
- An edge $u \xrightarrow{c} v$ denotes a control message in transit between u and v.
- An edge $u \xrightarrow{p} v$ denotes that v has stored u as its parent.
- A loop $u \xrightarrow{i} u$ denotes that u is the initiator.
- A loop $u \xrightarrow{t} u$ denotes that u is in the Dijkstra–Scholten tree.
- A loop $u \xrightarrow{s} u$ denotes an increment of the implicit counter at u, which should be interpreted to be 0 if there are no s-loops.

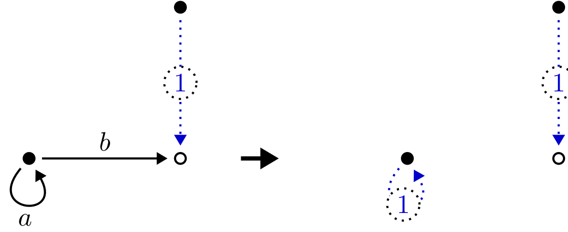
The starting undirected network is encoded as the smallest multigraph containing:

- $u \xrightarrow{e} v$ iff there is an undirected edge between u and v, and
- $u \xrightarrow{i} u$ and $u \xrightarrow{t} u$ iff u is the initiator of the basic algorithm.

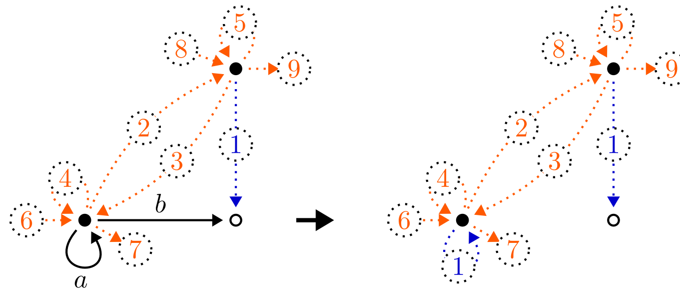
Notation 27 (Black Node Shorthand). As an abbreviation, we use black nodes in rules. For the black nodes and the context node we induce the largest patch type edge subgraph possible (i.e., the most permissive subgraph of patch type edges). The patch type edges are simply copied over to the right-hand side, where nodes that are in the same relative position on the left and on the right are identified.

The notation is best explained through an example.

Example 28. The rule



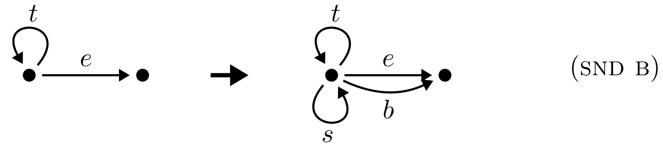
abbreviates the rule



Observe that Notation 27 provides a simpler alternative to Notation 23 for use cases where node deletion and merging do not take place. We believe it to be generally useful for modeling distributed algorithms, where it is often only the relations between processes that change.

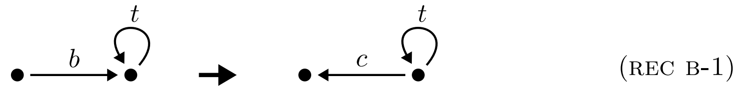
An execution of the Dijkstra–Scholten algorithm can now be modeled using the following rules:

– Sending a basic message:



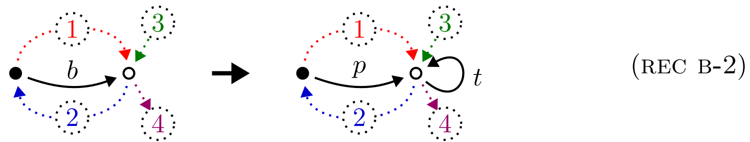
(SND B)

– Receiving a basic message while in the tree:



(REC B-1)

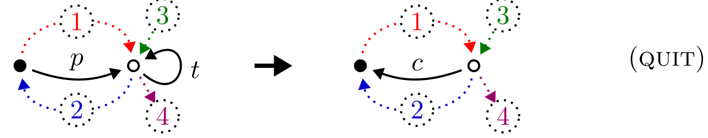
– Receiving a basic message while not in the tree:



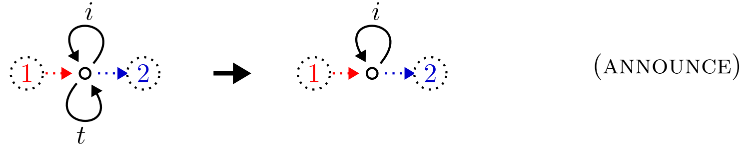
(REC B-2)

– Receiving a control message:

– A non-initiator quits:



– The initiator quits/announces:

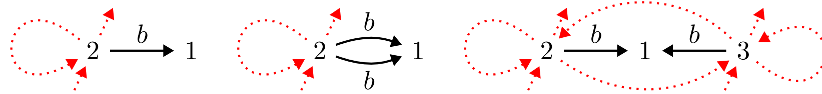


F Modeling Constraints

Instead of using unlabeled type edges, one could label the type edges with expressions from some suitable constraint language and strengthen the definition of adherence. Constraints that immediately suggest themselves include those that restrict the permitted number of adherent edges or their labels.

For many application scenarios, however, the number of edges is either unbounded, or there is only a small number of permitted choices. The latter case can in principle be handled with unlabeled type edges as follows. Consider the following left-hand side of some rule:

which allows 1 to have any number of incoming edges from nodes other than itself (and no outgoing edges are allowed). Assume instead that we want to express that node 1 has either one or two incoming edges labeled with b from nodes other than itself. Then we can replace the scheme with three schemes that together capture precisely this constraint:



Clearly, this transformation quickly leads to a prohibitive large number of rules. We have chosen to keep things simple as this suffices for many application scenarios.