Reducibility between classes of port graph grammar

Charles Stewart
Dept. of Computer Science
Boston University
cas@linearity.org

24th March 2001

Abstract

This paper introduces a general notion of static-port graph grammar (SPGG) that encompasses existing formalisms that have been independently proposed, such as Linear Graph Grammars [Baw93], Interaction Nets [Laf90] and Partial Sharing Graphs [Lam90]. These formalisms have been shown to provide a computational framework for asynchronous computation that respects a very rigorous notion of local interaction, and have proven a suitable basis for the internal representation of program and data in the compilation of high-level programming languages for distributed execution.

The general class of SPGGs provide a more general computational framework, still possessing asynchronous concurrency but which has a reduction mechanism that is complex and non-local. In this paper, besides introducing the port graph grammars formally and describing important subclasses, we propose a theory of translations suitable for use in reasoning about compiler correctness that fully preserves concurrency, and we give results showing how general SPGGs may be translated to the purely local, basic SPGGs, which are a slight generalisation of the above existing formalisms.

Contents

1 Introduction

A kind of graph transformation system has been independently proposed on several occasions with minor variations to solve different problems:

- Alan Bawden proposed a framework in 1986, initially by the name 'Connection graphs', as a programming language for distributed computation based on his experience with the massively parallel Connection Machine at MIT [Baw86,Hil85]. His PhD thesis develops the idea further (now calling them 'Linear graph grammars', influenced by the link with linear logic described below), and is a key formalism used in the design of a compiler, where the linear graph grammars serve as both an intermediate representation used in compiler optimisation (in the same kind of way as the basic block representation is used in the Dragon book [ASU85]), and in the run-time environment as a basis for dynamic load-balancing [Baw93,BM].

- Yves Lafont formulated a more restrictive variant, his 'Interaction nets', as a computational framework in a close correspondence with multiplicative linear logic, which is intended to explore a sense in which linear logic serves as a kind of process calculus [Laf90]. This correspondence has been developed further: a joint paper by Georges Gonthier, Martin Abadi and Jean-Jacques Lèvy shows how multiplicative-exponential linear logic may be encoded into John Lamping's partial sharing graphs[1] [GAL92b], whilst Satoshi Matsuoka's additive interaction nets possesses a correspondence with multiplicative-additive linear logic [Mat00]. Interaction nets have become quite widely known, and are of interest to researchers interested in the compilation of pure functional languages (Ian Mackie's PhD thesis describes the implementation of a simple graph rewriting language using interaction nets) as well as within the research program associated with linear logic.

- John Lamping devised a specific graph grammar that permits the encoding of the lambda calculus, with explicit annotations for sharing of subterms. Reduction in this calculus is optimal in the sense of Lèvy: all beta redexes are shared when they are performed if they descend from a common beta redex, an important property that has proved surprisingly difficult to capture in a rewrite system. Another joint paper by Gonthier, Abadi and Lèvy has exposed links between this algorithm and linear logic, a relationship that reveals deep connections between Lèvy labelling and Girard's Geometry of Interaction [GAL92a]. Andrea Asperti and Stefano Guerrini have implemented a simple compiler, the Böhm machine, for a pure functional language whose implemented reduction strategy is Lèvy-optimal; the implementation is described in their book [AG98].

This paper proposes a graph grammar framework, the *static-port graph grammars* (SPGGs) that generalises the above specific formalisms. The motivating application of this paper is to provide a theoretical basis for a compiler with similar aims to that described in Alan

[1] Only inexactly; Ian Mackie's PhD thesis describes an exact encoding for this part of linear logic [Mac94].

Bawden's PhD thesis: namely to compile a typical high-level programming language into an executable capable of being run on a cluster of machines in a manner that permits on-the-fly load-balancing. The linear graph grammar formalism possesses a decisive advantage over SPGGs when it comes to use in the run-time advantage, namely the computational triviality of detecting pattern matches[2], but it is argued here that the more general SPGG formalism is a useful intermediate form for the compiler, since it allows more transparent encoding of some programming language constructs, and more opportunities for optimisation (this case is made in section two).

To be able to use SPGGs at one stage in the compilation process, but to output code based upon a more primitive stage, one must be able to transform an SPGG representation of the program into the simpler representation. Let us examine a hierarchy of graph grammar formalisms, organised from most expressive to least:

1. *Undirected hyperedge replacement grammars* are a widely used and investigated formalism that allow relatively direct generalisation of term rewrite systems[3] [Roz97]. The question of whether they can be reduced to SPGGs is an unsolved problem discussed in the conclusions.

2. *Static-port graph grammars* are introduced in this paper, and the main result of this paper is to show that they can be reduced to basic SPGGs. SPGGs are essentially a kind of constrained hyperedge replacement grammar, according to the following correspondence: the vertices of SPGGs correspond to hyperedges, the edges of the SPGGs correspond to nodes, the ports of the vertices correspond to the tentacles of the hyperedges, free ports implicitly correspond to vertices with zero or one incident hyperedges, and the constraints on the generated graphs are (i) that at most two tentacles may be incident on any node of the grammar, (ii) the patterns of SPGG rewrite rules must be connected and contain at least one vertex, and (iii) the free ports in the replacement graph of each rewrite rule must be the same as those in the pattern graph (this last invariant is responsible for the name 'static-port' in the name of the formalism).

3. *Elementary SPGGs* are SPGGs all of whose patterns have at most two vertices. This is the strongest sort of grammar that might plausibly be the basis for program representation of the kind of distributed run-time environment proposed by Bawden's PhD thesis; plausibly the patterns of elementary SPGGs also can be detected in constant time.

4. *Basic SPGGs* are elementary SPGGs all of whose patterns correspond to one of three restrictive forms, namely binary interactions (consisting of two vertices joined by an edge), explode symbol rewrites (consisting of a single vertex with no internal edges) and loop absorption (a single symbol with a number of looping edges).

[2]These can be implemented in effectively constant-time using hash-table lookup of pairs of vertex types

[3]This is not exactly true, since some term rewrite systems allow the structure of a subterm occurrence to influence if it is a rewrite or not. In the absence of such constraints, hyperedge replacement grammars generalise term rewrite systems.

5. *Linear graph grammars* are effectively elementary SPGGs all of whose patterns are binary interactions. It is possible to reduce grammars with explode symbol rewrites to ones without, but, for reasons discussed in chapter 6 of Alan Bawden's PhD thesis it is impossible to reduce grammars with loop absorption symbols to ones without, due to the impossibility of loop detection by the rewrites of such a grammar.

6. *Interaction nets* are linear graph grammars that respect a number of additional requirements, most distinctively that each vertex interacts in all patterns on the same port. This criteria ensures the rewrite grammar is confluent (and hence that linear graph grammars are not reducible to interaction nets).

In summary, the principal contributions made in this paper are:

- The formalisation of a notion of translation between rewrite systems, which gives a precise sense in which the intended reducibility result may be said to maintain compiler correctness and conserve concurrency (ie. all opportunities for concurrency that are present in the source rewrite system are present in the target system). This is given in section two, together with a discussion of compiler architecture and goals.

- The introduction of static-port graph grammars in section three as a general framework in which all the above mentioned grammars arise as special subclasses by the method of filtered grammars. The basic SPGGs are described as the target of the translation.

- The general reducibility of arbitrary static-port graph grammars to the basic static-port graph grammars is shown by constructing of a basic SPGG from any given source SPGG and providing a translation. The correctness of the translation is shown by proving that all reachable graphs of the target grammar satisfy a structural property. Section four gives an informal overview of the translation scheme by going through an example, as well as discussing difficulties that face construction of such a translation. Section five constructs the target grammar from a given source grammar, and section six defines the encoding and proves it correct.

2 Correctness of concurrent compilation

The general form of the compiler we are interested would follow a typical form: the compiler accepts the users code in the form of a file written in a high-level programming language, and parses it to obtain a syntax tree for the program, after performing some preprocessing on the file. The syntax tree is transformed into an internal representation, based upon the semantics of the programming language. A series of optimisation steps are performed by applying analyses upon the internal representation to reveal ways in which the code can be transformed to a more efficient form without endangering its correctness. When these optimisations are exhausted, the target code is generated according to the internal representation, in the form of input to a 'C' compiler.

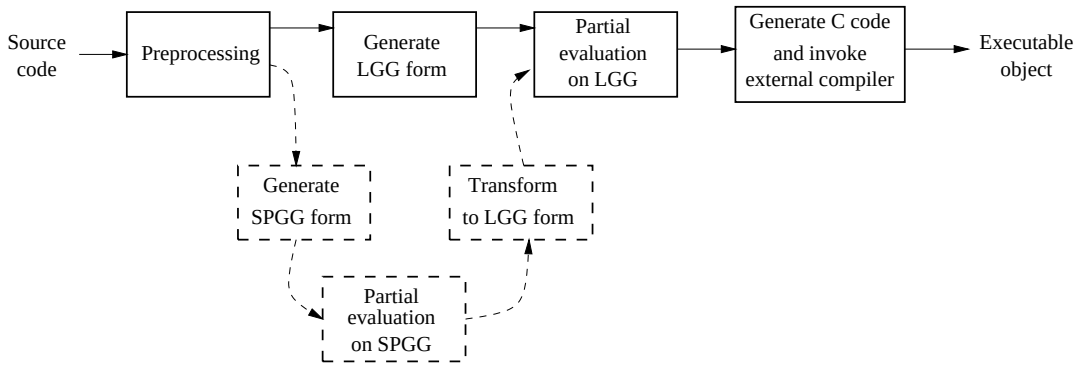


Figure 1: Structure of compiler

We shall suppose we have an existing compiler which uses linear graph grammars for its internal representation, and that we wish to change the encoding of the language so that the language is coded directly into a SPGG representation, and then has some optimisation steps applied to it, before converting to the linear graph grammar form to perform the rest of the optimisation steps. How might such an extension of the compiler work? In figure 1 we show the stages of a very simple compiler, where the straight path from source code to executable along the top of the figure indicates the existing, compilation strategy which makes use of the linear graph grammar intermediate form, and the alternate path indicated beneath shows the new structure that makes use of SPGGs.

Let us look at the parts of the compiler, which we call *stages*. In the first stage the compiler performs elementary preprocessing on the source file provided by the user, such as macro expansion. The next stage transforms the program into its intermediate representation, consisting of a linear graph grammar and a working graph. The third stage applies a simple optimisation strategy to the grammar and working graph output from stage two, namely a form of partial evaluation, where if a redex occurs in the replacement graph of a rewrite, then that redex is automatically expanded, and also the stage applies selective expansion of redexes in the working graph. In the final stage a code generator is then applied to the result of this phase to generate C code, which is passed to a 'C' compiler to return the final executable or linkable object file.

The modified compiler makes use of a different transformation of the source into intermediate representation, compiling the source language into the more general SPGG representation (likewise consisting of a static-port graph grammar and working graph). A similar optimising stage is performed, before transforming the general SPGG code into a basic SPGG. Then the remaining stages of the original compiler are performed (omitting, of course, the original code generation stage) – note that the reducibility result proven in this paper does not permit reducing the SPGG grammar to a linear graph grammar, so we must generalise these later stages from LGGs to basic SPGGs; in practice this appears not to be difficult to achieve. Finally the C code is generated and compiled.

What do we gain from the modified compiler? There are three principal advantages. Firstly, when compiling the constructs of the programming languages, it may be more

natural to express a construct using the more complex patterns that are permitted in an SPGG than with the binary interactions that we are forced to use in a linear graph grammar. Second, it may be much easier to see how to conserve the potential for concurrency in the source when using SPGGs. Lastly, potential optimisations may be explicit in the SPGG form that are hard to spot in the linear graph grammar form.

All three of these can be illustrated with an example. Consider the following piece of pseudo-ML code to code a non-deterministic merge function, where the `alt` construct behaves like a concurrent pattern matching statement (ie. we can despatch on any guard if the pattern matches the expression after the `alt` literal; the choice of guard is unspecified if more than one pattern could apply):

```
let rec ndmerge arg
 = alt arg with
 | (a::as, bs) -> a::(ndmerge (as, bs))
 | (as, b::bs) -> b::(ndmerge (as, bs))
 | ([],[]) -> []
```

This code is then ambiguous in the result of `ndmerge (1::fs, 2::gs)`.

Using standard techniques, we can transform such pattern matching expressions into complexes of `if-then-else` constructs, which can easily be compiled into a linear graph grammar. To do so necessitates a choice as to which part of the argument of `ndmerge` is inspected first, potentially losing the opportunity for concurrency and optimisations.

To see this, suppose the left-hand part of the pair is tested first. Then in an instance such as `g(ndmerge (f(a), b::cs))`, the evaluation of `ndmerge` is forced to wait for the function `f` to return its result, whilst if we had concurrent semantics we could choose the alternative evaluation of `ndmerge`, and the function `g` could begin to process its argument (if it has a lazy semantics in its argument) even before `f` has completed. Similar reasoning shows how optimisations can be missed.

Providing a concurrent encoding for pattern matching into linear graph grammars is fraught with complications, but with SPGGs we can naturally express the `alt` construct by coding the separate cases as patterns of three rewrite rules, as shown in 2. Reduction in this graph grammar is concurrent in just the sense we are after, and the method of partial evaluation we outlined is quite able to deal with such cases as `g(ndmerge (f(a), b::cs))`.

Having described the kind of compiler framework we are interested in, it remains to characterise the criteria that the transformation from arbitrary SPGGs to basic SPGGs must satisfy. In the remainder of this section, we shall give a formal criteria for a translation (described generally in terms of rewrite systems) to be correct, and for it to preserve concurrency (it would be worthless to go the effort of providing representations of the programming language that maximally express the opportunities for concurrency if we then lose all of this information in one of the stages of the compiler).

We formalise rewrite systems abstractly as follows, together with some useful definitions:

DEFINITION 1

1. A *rewrite system* G consists of a set of *terms* $\mathcal{G}(G)$, and a set of rules $\mathcal{R}(G)$ where each

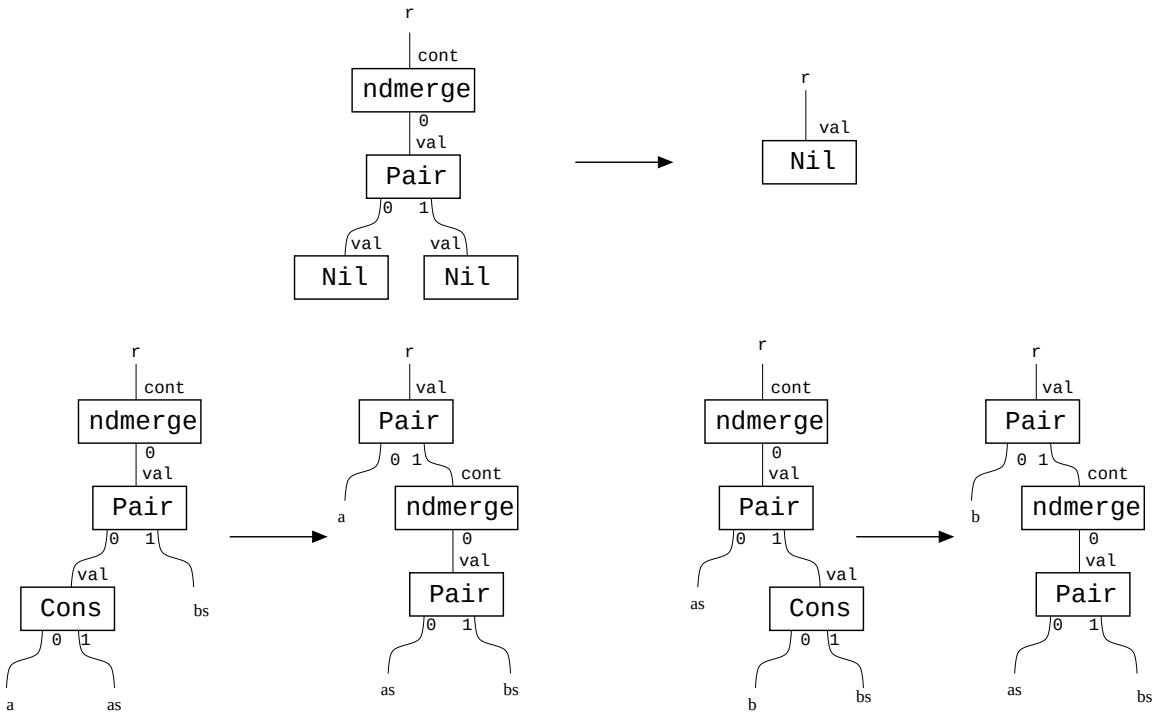


Figure 2: Coding `ndmerge` into SPGG rules

rule specifies (i) a *pattern*, (ii) a *match function* from terms in $\mathcal{G}(G)$ to the set of matches for the given pattern in the term, and (iii) a *replacement function* which takes a pair of a term and a match for the given pattern to the replacement term.

2. Given a rule r, a *rule instance* of r consists of a triple $\langle t, m, u\rangle$, where t is a term, m is a match for the pattern of r in t, and u is the replacement for m in t.

3. The one-step rewrite relation $\rightarrow^1_r$, where r is a rule of G, is specified by $t \rightarrow^1_r u$ if and only if there is some match m such that $\langle t, m, u\rangle$ is a rule instance of r. If R is a subset of the rules of G, then $t \rightarrow^1_R u$ when there is $r \in R$ such that $t \rightarrow^1_r u$, and we write $t \rightarrow^1_G u$ if $t \rightarrow^1_r u$ for any rule r of the rewrite system G.

4. The reduction relation $\rightarrow^*_R$ is the reflexive, transitive closure of $\rightarrow^1_R$.

5. A *rewrite chain* is a sequence of rule instances $\langle\langle t_1, m_1, u_1\rangle, \ldots, \langle t_n, m_n, u_n\rangle\rangle$ where for each $i < n$, $t_{i+1} = u_i$.

6. A term is R-normal if there is no term t for which $s \rightarrow^1_R t$.

7. Two rule sets R and S satisfy the *joint diamond property* if whenever $s \rightarrow^1_R t_R$ and $s \rightarrow^1_S t_S$, then there is a term u such that $t_R \rightarrow^*_S u$ and $t_R \rightarrow^*_S u$. A rule set satisfies the diamond property if it satisfies the joint diamond property with itself.

DEFINITION 2 A *translation* of the rewrite system G into the grammar H is given by:

1. An embedding map $F : \mathcal{G}(\Sigma_G) \to \mathcal{G}(\Sigma_H)$.
2. An interpretation map on graphs $R : \mathcal{G}(\Sigma_H) \to \mathcal{G}(\Sigma_G)$.
3. A partial interpretation map on rules $R_1^{\to} : \mathcal{R}(H) \rightharpoonup \mathcal{R}(G)$.

which satisfies the following conditions:

1. (Left inverse) $\forall g \in \mathcal{G}(\Sigma_G).g = R(F(g))$.
2. (Neutrality of bookkeeping rewrites) $\forall r \in \mathcal{R}(H)$, with $r \notin \mathsf{dom}(R_1^{\to})$ then for each rule instance $\langle t, m, u\rangle$ of r, $R(t) = R(u)$.
3. (Correspondence of working rewrites) $\forall r \in \mathcal{R}(H)$, with $r' = R_1^{\to}(r)$, for each rule instance $\langle t, m, u\rangle$ of r, there is a match m' such that $\langle R(t), m', R(u)\rangle$ is a rule instance of r' in G.

DEFINITION 3 We define a map from rewrite chains of H to rewrite chains of G as follows;

$$R^{\to}(\langle r\rangle) = \begin{cases} \langle\rangle & \text{if } r \notin \mathsf{dom}(R_1^{\to}(r)) \\ \langle R_1^{\to}(r)\rangle & \text{otherwise} \end{cases}$$

$$R^*(\langle r_1, \ldots, r_n\rangle) = R^*(r_1) \hat{} \ldots \hat{} R^*(r_n)$$

Together with the interpretation map R, this gives us an interpretation functor from the rewrite system H to the rewrite system G. This notion of translation allows that a single rewrite in G may correspond in the translation to a chain of rewrites in H: this allows relatively complex rewrites in G to be broken up into a series of simpler rewrites in H, provided we nominate one of these translation rewrites as performing the work of the original rewrite.

The intention is that we give a translation from the rewrite system G to the rewrite system H. Two important ways in which it might fail are:

- Firstly, it might be the case that there is a graph $h \in \mathcal{G}(\Sigma_H)$ which admits no reductions, but $R(h)$ is reducible in rewrite system G. In this case, we say that the translation may *deadlock*.
- Secondly, it might be the case that the graph h may perform endless reductions not in the domain of ϕ (these we call 'book-keeping reductions'). In this case, we say that the translation may *livelock*.

If the translation satisfies both deadlock and livelock freeness it may be said to be *correct*: the definition of translation ensures that any reductions performed in the target grammar are justified by the rewrite semantics of the source grammar, whilst deadlocak and livelock freeness ensures that there are not either dead-ends or infinite detours of meaningless computation in the target grammar. Correctness ensures that the transformation is

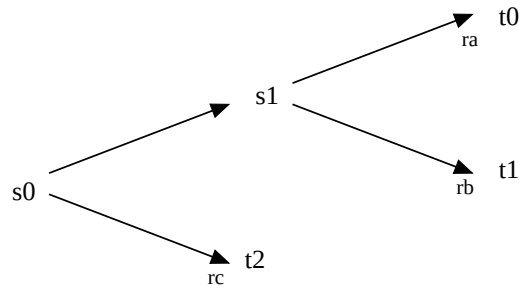


Figure 3: Blocked rewrites

adequate from the point of view of the semantics of the compiler, but it does not ensure that we conserve the potential for concurrency present in the source system, an crucial property for the kind of compiler we wish to construct, since without it there my be no opportunity for parallelism in our final code.

Let us look at how the translation may fail to conserve concurrency. It may be the case that he translation restricts the possible set of rewrites. For example, if the source graph has two overlapping patterns available from the original graph s0, say PA and PB, a translation may always ensure that the rewrite with pattern PB never takes place whilst there is a PA pattern available in the graph. This would mean that, for some graphs, possible rewrite chains in the source grammar would not correspond to any rewrites in the target grammar, or in other words, the translation loses potential concurrency.

DEFINITION 4 The *encoded rewrites available* at a graph h of the target grammar H are the set of rewrites r of $R(h)$ in the source grammar G for which there is a rewrite chain $\vec{s}$ from h for which $\phi^*(s) = \langle r \rangle$. The rewrites of $R(h)$ which aren't the image of some such rewrite chain are called the *encoded rewrites blocked* of h.

It would be a mistake, however, to suppose that one should insist that no graph ever has blocked rewrites: in figure 1, we see that from the first graph, s0, all possible final graphs, t0-t2, are available, but to obtain either t0 or t1 we must perform a bookkeeping rewrite to an intermediate graph s1, so blocking the possible final graph t2[4]. So we see that a more general criterion is needed:

DEFINITION 5 An translation of rewrite system G into rewrite system H is *complete* if for any rewrite chain $g_0 \rightarrow^*_{\vec{r}} g_1$ in G and graph h_0 such that $h_0 = F(g_0)$, there is a rewrite chain $h_0 \rightarrow^*_{\vec{s}} h_1$ in H, where $\vec{r} = R^{\rightarrow}(\vec{s})$.

Note that the completeness criteria does not entail either livelock freeness (since the condition ensures only that some terminating book-keeping chains exist), nor deadlock freeness (we ensure the graphs in the codomain of $F(-)$ are deadlock free, but other graphs could deadlock). It does ensure that all possible reduction chains in the source system have

[4]As an aside, we might note that the interpretation function $R^{\rightarrow}$ can be viewed as a sort of trace semantics for reductions in H, where rewrites not in the domain of $R^{\rightarrow}$ are rather like τ interactions in Robin Milner's CCS [Mil89]. Ideas from trace semantics may be useful in further developing the formalisms introduced here.

corresponding realisations in the target system, so in this sense a complete translation is concurrency conserving.

DEFINITION 6 A *functorial translation* is given by a 4-tuple $\langle F, F^{\rightarrow}, R, R^{\rightarrow} \rangle$, where $\langle F, R, R^{\rightarrow} \rangle$ is a translation, and $F^{\rightarrow}$ a map from rewrite chains of G to rewrite chains of H that satisfies:

1. If $g_0 \rightarrow^*_{\vec{s}} g_1$ then $F(g_0) \rightarrow^*_{F^{\rightarrow}(\vec{s})} F(g_1)$.
2. $F^{\rightarrow}(\langle\rangle) = \langle\rangle$ and $F^{\rightarrow}(s_0 \hat{} s_1) = F^{\rightarrow}(s_0) \hat{} F^{\rightarrow}(s_1)$, for all rewrite chains $\vec{s_0}, \vec{s_1}$ of G with the last graph of the chain $\vec{s_0}$ matching the first graph of the chain $\vec{s_1}$.
3. $G^{\rightarrow}(F^{\rightarrow}(\vec{s})) = \vec{s}$, for all rewrite chains $\vec{s}$ of G.

It is immediately apparent that all functorial translations are complete; but it is perhaps surprising that not all complete translations can be extended to functorial translations.

DEFINITION 7 $\phi : G \rightarrow H$ is a *faithful translation* if it is livelock free, deadlock free and complete.

We say that a class of rewrite systems is *reducible* to another class if for each system G in the first class, there is a system H in the second class together with a faithful translation $\phi : G \rightarrow H$.

3 Static-port Graph Grammars

DEFINITION 8

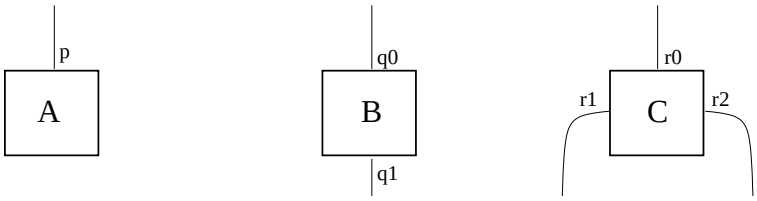
1. A *p-graph signature* Σ consists of a finite set of vertex types, where each vertex type consists of a distinct vertex symbol and a non-empty, finite set of distinct port identifiers.
2. A *p-graph* over a p-graph signature Σ consists of:
   (a) A finite list of *vertices*, where each vertex is associated with a vertex type from Σ. The set of indexes of the vertex list of the p-graph g is denoted VX_g. Two functions are defined with domain VX_g: ports maps VX_g onto the set of ports associated with the given vertex type, and sym maps VX_g onto the symbol.
   (b) A finite set of *free ports*, denoted FP_g. The ports of the graph are the union of the free ports and the ports associated with the vertices (denoted by a pair of the vertex index and the port identifier):
   $$FP_g \cup \{\langle v, pt\rangle \mid v \in \mathsf{VX}_g \wedge pt \in \mathsf{ports}(v)\}.$$
   (c) The *facing port function*, which is an endofunction face_g whose domain/range is the ports of the graph, and which satisfies:
      i. (Idempotency) $\forall pt \in \mathsf{VX}_g.\mathsf{face}_g(\mathsf{face}_g(pt)) = pt$.
      ii. (Irreflexivity) $\nexists pt \in \mathsf{VX}_g.\mathsf{face}_g(pt) = pt$.

3. The set of p-graphs defined over a graph signature Σ is written $\mathcal{G}(\Sigma)$. We say that a graph is *saturated* if it contains no free ports.

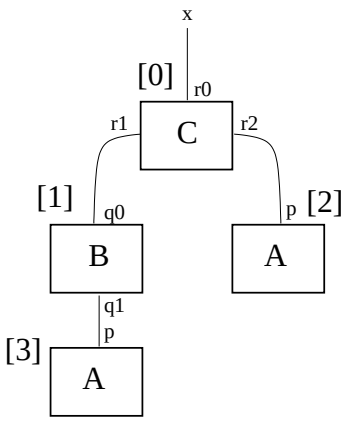
REMARK 9 The facing port function determines the edges of the p-graph by specifying how edges connect pairs of ports. We make this precise by saying that an (undirected) *edge* is an unordered pair $\{pt, qt\}$ with the property that $\mathsf{face}(pt) = qt$. This formulation, together with the conditions on face, ensures that membership of an edge is an equivalence relation on the ports of the graph. Conversely, specifying the edge set is equivalent to giving the facing port function.

EXAMPLE 10

1. A p-graph signature of three vertex types may be given graphically:



2. A p-graph over the above signature is:



The labels "[0]", "[1]" and "[2]" indicate the vertex identifiers of the vertices they are next to. The label at the top of the graph gives the free port identifier of the single free port of the graph. Thus the edge set for this graph is

$$\{\{x, \langle 0, r0\rangle\}, \{\langle 1, q0\rangle, \langle 0, r1\rangle\}, \{\langle 2, p\rangle, \langle 0, r2\rangle\}, \{\langle 1, q1\rangle, \langle 3, p\rangle\}\}$$

DEFINITION 11

1. We say that an edge is *internal* if neither of its ports are free ports, and that a port is internal if it occurs in an internal edge. A port is *external* if it faces a free port, but is not itself a free port. Thus each port of the graph is either free, internal or external.

2. A graph is *discrete* if it has no internal edges.

3. A *path* across a graph g consists of a sequence of ports $\langle pt_1, \ldots, pt_n \rangle$ ($n \geq 1$) such that for $1 \leq i < n$, the ports $\mathsf{face}_g(pt_i)$ and pt_{i+1} are ports of the same vertex (neither may be free ports). For this path pt_1 is the starting port, and $\mathsf{face}_g(pt_n)$ is the final port: if these are not free ports, then the associated vertices may be said to be the start and end points of the path.

4. The *terminals* of a path are the start and the final terminal, where the start terminal is the starting port if it is a free port, or the vertex of which it is the port otherwise; similarly for the final terminal. A *circuit* is a path whose terminals are the same vertex.

5. A graph is *connected* when both:

   (a) Either each pair of vertices are terminals of some path, or there is just one vertex.

   (b) Each free port faces an external port.

If a graph is not connected, then its *components* are its maximal connected subgraphs.

To define rewrites of the graph we need to specify the homomorphisms between two graphs. We shall need two kinds of homomorphism, a weak form (partial homomorphism) and a strong form (full homomorphism). Let g, h be two p-graphs sharing a common signature:

1. A *full homomorphism from g to h* consists of an injective mapping ϕ from the ports of g to the ports of h which satisfies the following properties:

   (a) (Preserves vertices)

   $$\begin{aligned}\forall pt_0, pt_1.(\exists V \in \mathsf{VX}_g.pt_0, pt_1 \in \mathsf{ports}(V)) \\ \supset (\exists W \in \mathsf{VX}_h.\phi(\langle V, pt_0\rangle), \phi(\langle V, pt_1\rangle) \in \{\langle W, qt\rangle \mid qt \in \mathsf{ports}(W)\})\end{aligned}$$

   (b) (Preserves edges) $\forall pt, qt.pt = \mathsf{face}_g(qt) \supset \phi(pt) = \mathsf{face}_h(\phi(qt))$.

   (c) (Preserves vertex type and port label) $\forall v, pt, w, qt.\langle v, pt\rangle = \phi(\langle w, qt\rangle) \supset \mathsf{sym}_g(v) = \mathsf{sym}_h(w) \wedge pt = qt$

2. A *partial homomorphism from g to h* is defined in case g is connected, and consists of either:

   (a) An injective mapping ϕ from the internal ports of g to the internal ports of h which satisfies the same three properties above, in case g has internal ports.

   (b) A vertex from VX_h if g has no internal ports.

3. An *equivalence between g and h* is a consists of a full homomorphisms: $\phi : g \to h$ which is surjective on the ports of h. Two graphs are equivalent if they have an equivalence. Two full homomorphisms are equivalent if one is the composition of the other with an equivalence.

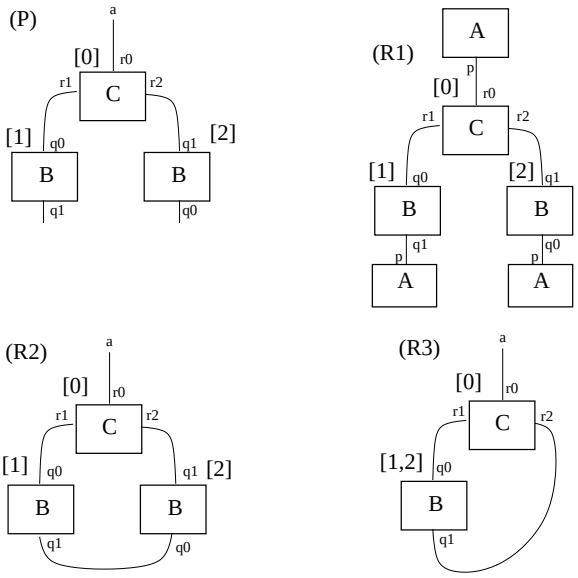


Figure 4: Full and partial homomorphisms

Figure 4 specifies three homomorphisms as follows: there is a full homomorphism from P to R1 by mapping each port of a vertex with a given vertex identifier in P to the same port of the similarly labelled vertex of R1 (the labels next to the graphs in R1 to R3 are not vertex identifiers, but just specify the action of the homomorphism from P). Similarly there is a full homomorphism from P to R2. The homomorphism from P to R3 is partial but not full: while the mapping on internal edges is injective, both of the external ports $\langle 1, q0\rangle$ and $\langle 2, q0\rangle$ of P map to the same port of R2. For a mapping that fails to be even a partial homomorphism, look at the reverse map from R2 to P: the map of the edge from "[1]" to "[2]" in R2 doesn't map onto an edge in P, so fails to satisfy the edge preservation condition.

Observe that due to the vertex preservation property, there is always a unique mapping on vertices induced by the partial and full homomorphisms associated with the mapping on ports.

PROPOSITION 12

1. The mapping on vertices induced by a full homomorphism is injective.
2. If a partial homomorphism induces an injective mapping on vertices, then it uniquely determines a full homomorphism.
3. An equivalence is a relabelling function on vertex identifiers and free ports.

PROOF Part 1. To each vertex in the domain of the full homomorphism, choose an arbitrary port, and let the action of the induced function on the vertex be the vertex associated with the homomorphism of this chosen port. Since all of the ports of each vertex are in the domain of the full homomorphism, this function is injective and independent of the choice of ports.

Part 2. Observe that the induced function on vertices by the partial homomorphism may only fail to be injective if there are two vertices in the source of the same vertex type, but whose set of internal ports are disjoint. If these vertex ports are always disjoint, we may extend the partial homomorphisms to a full homomorphism by mapping each external port of a vertex in the source graph to the facing port of the corresponding vertex in the target graph.

Part 3. Observe firstly that the induced function on vertices must be bijective, and so the homomorphism must map free ports onto free ports. Since it is injective and surjective on ports, it must be bijective on free ports, and because of the edge preservation property, the facing port function for the target must be specified by the facing port function for the source graph. By the port label preservation property, these two bijections exhaust the action of the homomorphism, so it is just a relabelling function on vertex identifiers and free ports. □

REMARK 13 Note that the partial and full homomorphisms from a graph g into another graph are completely specified by their action on one port (possibly free) of each component of g. This is easily seen by considering the action of the homomorphisms on the paths in g.

DEFINITION 14 A *join of g and h along $\vec{p}$ and $\vec{q}$* (where $\vec{p}$ is a sequence of free ports of g, $\vec{q}$ is a sequence of free ports of h, and $\vec{p}$ and $\vec{q}$ have equal length) is a graph g^* which satisfies:

1. There are full homomorphisms $\phi : g \to g^*$ and $\psi : h \to g^*$.
2. All edges of g^* are in the range of either ϕ or ψ (ie. the maps cover g^*).
3. $\forall i. \phi(p_i) = \psi(\mathsf{face}_h(q_i))$.

The ports $\vec{p}$ and $\vec{q}$ are called the *gluing ports* of the join.

PROPOSITION 15 The join is uniquely defined for any two connected graphs, up to equivalence.

PROOF Let g_0^* and g_1^* be two joins of g and h along $\vec{p}$ and $\vec{q}$. We can construct an equivalence between the g_0^* and g_1^* by pointwise identification of the ranges of the full homomorphisms ϕ_0 used to construct g_0^* and ϕ_1 used to construct g_1^*, and similarly with ψ_0 and ψ_1; since the maps cover their respective joins, the constructed function is bijective on ports. Since the full homomorphisms preserve vertices, vertex type, port label and edges, so does the bijection, and hence it is an equivalence. □

DEFINITION 16

1. The *matches* for g in g^* are the set of graphs and pairs of gluing ports $\{\langle h, \vec{u}, \vec{v}\rangle\}$ such that g^* is a join of g and h along $\vec{u}, \vec{v}$.
2. If p, r are graphs with the same free ports (ie. their free ports are the same in number and have the same identifiers), then for each match $\langle h, \vec{u}, \vec{v}\rangle$ in the graph g, there is a *replacement* of r for p in g given by the join of r and h along $\vec{u}$ and $\vec{v}$.

3. We say that two matches $\{\langle h_1, \vec{u}_1, \vec{v}_1\rangle\}$ and $\{\langle h_2, \vec{u}_2, \vec{v}_2\rangle\}$ are *equivalent* if $\vec{u}_1$ and $\vec{u}_2$ are sequences of the same length n and there is a permutation on n-sequences ϕ such that h_1 and $h_2[\vec{u}_1 := \phi(\vec{u}_2)]$ are equal up to (i) vertex relabelling and relabelling of non-gluing free ports, and (ii) $\vec{v}_1 = \phi(\vec{v}_2)$.

PROPOSITION 17 There is a bijection between the set of full homomorphisms of p into g (modulo equivalence) and the set of matches for p in g (modulo equivalence), in case p is connected and has free ports.

PROOF We shall specify a map from full homomorphisms of p into g to matches for p in g, and show firstly that each match is equivalent to the image of some homomorphism and secondly that the map takes pairs of homomorphisms onto equivalent matches if and only if the matches are equivalent.

Let $\phi : p \to g$ be the full homomorphism, and define $\vec{u}$ to be a sequence of free ports of p each of whose elements do not map onto free ports of g under ϕ. Let $\vec{v}$ and be a sequence of fresh identifiers of the same, and construct a graph h as follows: its vertex identifiers are the same as those of g not in the range of the vertex function induced by ϕ (inheriting the same vertex symbol and port set); its free ports are the free ports of g not in the range of ϕ together with $\vec{v}$, and its facing port function is the same as g's for internal ports of h, and for free ports it is defined: $\mathsf{face}(v_i)$ is the port that is the image of u_i, and the facing port of the other free ports is the same as g's. By inspection we see that the joins of p and h along $\vec{u}, \vec{v}$ are equivalent to g.

To each match we can associate a homomorphism by taking the appropriate one from the two full homomorphisms that justifies the join construction. The image of this homomorphism is easily seen to be equivalent to the match (the permutation being determined by the choice of the sequence $\vec{u}$).

Lastly, the distinct matches created by two equivalent homomorphisms differ only in their vertex identifiers, the labels of their ports and the sets of identifiers $\vec{u}, \vec{v}$. Since the gluing ports $\vec{u}$ are specified by the free ports of the graph p, and the order of the ports $\vec{v}$ two equivalent homomorphisms will map onto equivalent matches, and vica versa. □

As a result of the above proposition, to specify a rewrite on a graph g, we need to specify the pattern p (which must be connected and have free ports), a full homomorphism $\phi : p \to g$ and a replacement graph r whose free ports are in a specified bijection ψ with the ports of p.

DEFINITION 18

1. An *SPGG rule* over a signature Σ is a pair $\langle p, r\rangle$, where p and r are graphs over Σ with the same free ports (ie. with the same free port identifiers), and p is connected and contains at least one vertex. A *rule set* is a set of SPGG rules over a common signature.

2. The matches of an SPGG rule $\langle p, r\rangle$ are as specified above. The replacement function given a term t and a match m for p in t is the replacement of r for p in t along the ports specified according to the above lemma.

Thus each SPGG determines a rewrite system.

DEFINITION 19

1. A *free static-port graph grammar* consists of a pair $\langle \Sigma, \mathcal{R} \rangle$ where:
   (a) Σ is a graph signature.
   (b) $\mathcal{R}$ is the rule set, a set of rewrite rules where each rule is specified by a triple $\langle p, r \rangle$, where p and r are graphs over Σ with the same free ports, and p is connected and contains at least one vertex. The graphs p_i are called the *patterns* of the SPGG.
2. A *filtered static-port graph grammar* consists of a triple $\langle \Sigma, \mathcal{R}, \phi \rangle$ where Σ and $\mathcal{R}$ are as before, and ϕ, the filter, is a predicate on graphs of the grammar, such that the rewrites specified by $\mathcal{R}$ are closed over $\{g \in \mathcal{G}(\Sigma) \mid \phi(g)$.

Clearly, any free SPGG G may be considered to be a filtered SPGG, where the filter always holds.

EXAMPLE 20

1. A *static-port tree grammar* may specified for any filtered static-port graph grammar G. It is the filtered SPGG sharing the same signature, whose rule set is the restriction of the rule set for G to rules whose rewrites do not contain circuits, and whose filter is the conjunction of the filter for G with the constraint that there are no circuits.
2. A graph grammar is *elementary* if all of its patterns contain at most two vertices. It is *complex* otherwise. A special case of the elementary graph grammars are the *basic* SPGGs; all the patterns of such a grammar take one of the following forms:
   (a) A *binary pattern* consists of two vertices joined by a single internal edge. A rewrite with a binary pattern is called an *interaction*.
   (b) A *unary pattern* contains exactly one vertex and has no internal edges. A rewrite with a binary pattern is called a *symbol expansion*.
   (c) A *looping pattern* contains exactly one vertex and one or more looping internal edges. A rewrite with a looping pattern is called a *loop absorption*.
3. A *typing* for a graph grammar is a symmetric binary relation on port types. A static-port graph satisfies a typing if the pairs of port types in each edge satisfy the typing relation. A *polar typing* is a typing where the port types can be partitioned into positive and negative port types, and all pairs of port types satisfying the typing consists of a positive and a negative port type. A filtered static-port graph grammar G respects a typing if all $g \in \mathcal{G}(G)$ satisfy the typing.
4. The *active ports* of a vertex V in a graph grammar G are the ports of V that are internal ports of some pattern of G. Lafont's Interaction Nets are examples of filtered, basic graph grammars that respect a polar typing, all of whose patterns are binary, and all of whose vertices have a single active port[5].

[5]We neglect the type-completeness, simplicity and semi-simplicity requirements, which can be formulated as certain kinds of filter.

4 Reducing general SPGGs to basic SPGGs

The problem we wish to solve is: can we provide a scheme of faithful translations that shows that the general class of free SPGGs is reducible to the class of basic SPGGs?

We shall solve this problem for a given G by constructing a quite complex grammar H and a relatively simple translation function. Since the constructed grammar is complex, we shall devote this section to describing the general features of the constructed grammar and giving informal reasons why simpler constructions do not suffice, and look at an example, before going into the formal construction.

The grammar H consists of four kinds of vertex: there are *working vertices*, which correspond to the vertices in G; *pattern vertices*, which contain information on prospective matches for patterns of rules in G; *message passing vertices*, which mediate certain interactions that cannot be resolved in a single basic rewrite, *agglutinated vertices*, which stand in for complexes of pattern vertices and working vertices, more precisely they stand in for arbitrary tree subgraphs[6], and *fire pattern* vertices, which represent patterns that are ready to perform work rewrites. Both working vertices and pattern vertices may contain additional information (which we call *attributes*) in the form of finite bit sets; since the set of possible values for the attributes are finite this is just a convenient shorthand for a finite grammar with a larger number of vertices and rules.

The agglutinated vertices are the key to breaking up rewrites whose patterns involve more than two vertices into a series of basic rewrites. Consider the rewrite labelled PA from the grammar in figure 5.

This can be broken into two rewrites in the target grammar, by introducing a bookkeeping interaction (an *agglutinative rewrite*) between the left 'B' and 'C' vertices to form an agglutinated vertex, which can then perform a working rewrite with the remaining 'B' vertex to produce the replacement graph as shown in figure 6.

The convention we shall follow is that agglutinations are displayed as the subgraph they stand in for, whose pattern vertices are displayed with dashed rather than solid lines. The internal edges of the represented subgraph are not edges of the whole graph, the ports of the agglutinated vertex are the external ports of the represented subgraph and whose port identifiers of the agglutination are the ports of the subgraph (ie. the port label is the tuple of the vertex identifier of the concerned vertex and the port identifier), and the symbol of the agglutination is the represented subgraph.

By admitting agglutinations (ie. the vertices and the rewrites) of all graphs that are connected subgraphs of patterns of the general grammar, all of the patterns of the general grammar can be reached by some sequence of intermediate agglutinative steps. Adding in the working rewrites, one obtains a simple translation which is livelock free and complete.

Unfortunately the translation deadlocks: in general a vertex may belong to two different pattern matches in a way that means that to agglutinate towards one pattern means blocking the other pattern (in the sense that it cannot be reached anymore by a series of agglutinative rewrites). Since whichever choice is made may turn out to have been the

[6]We shall not need more complex agglutinations in this paper, but it makes sense to allow agglutinations to stand in for allow arbitrary connected subgraphs.

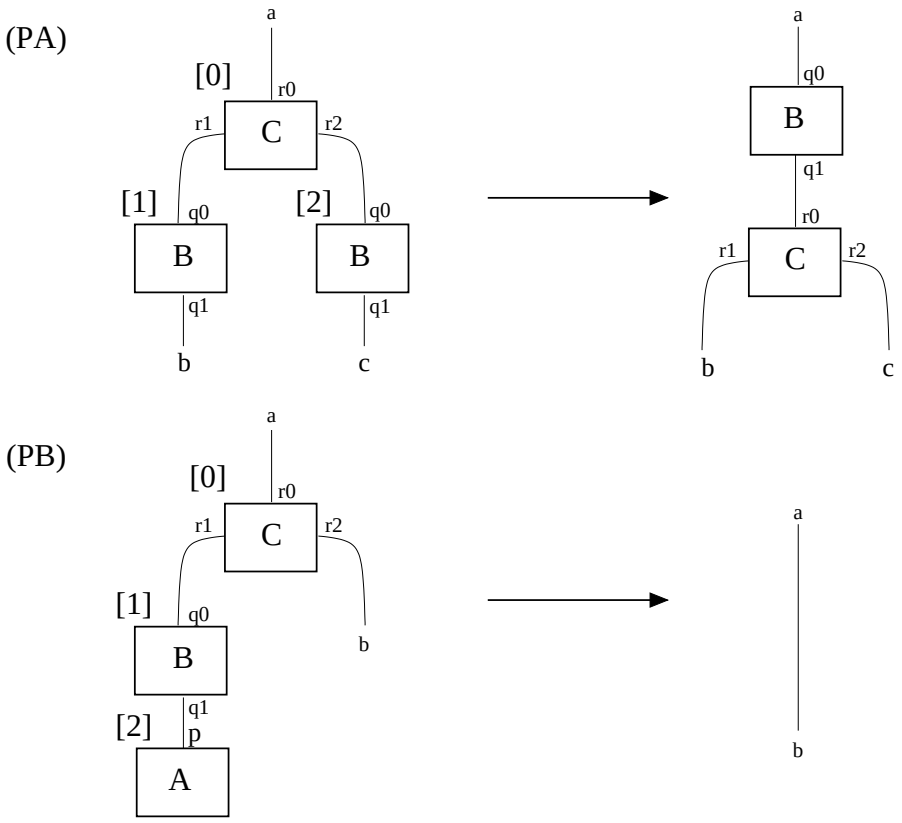


Figure 5: A two rule grammar

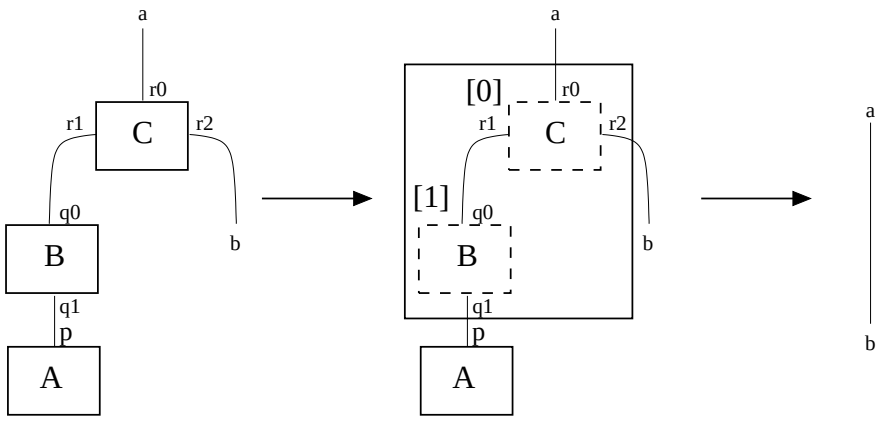


Figure 6: Translating with an agglutinated vertex

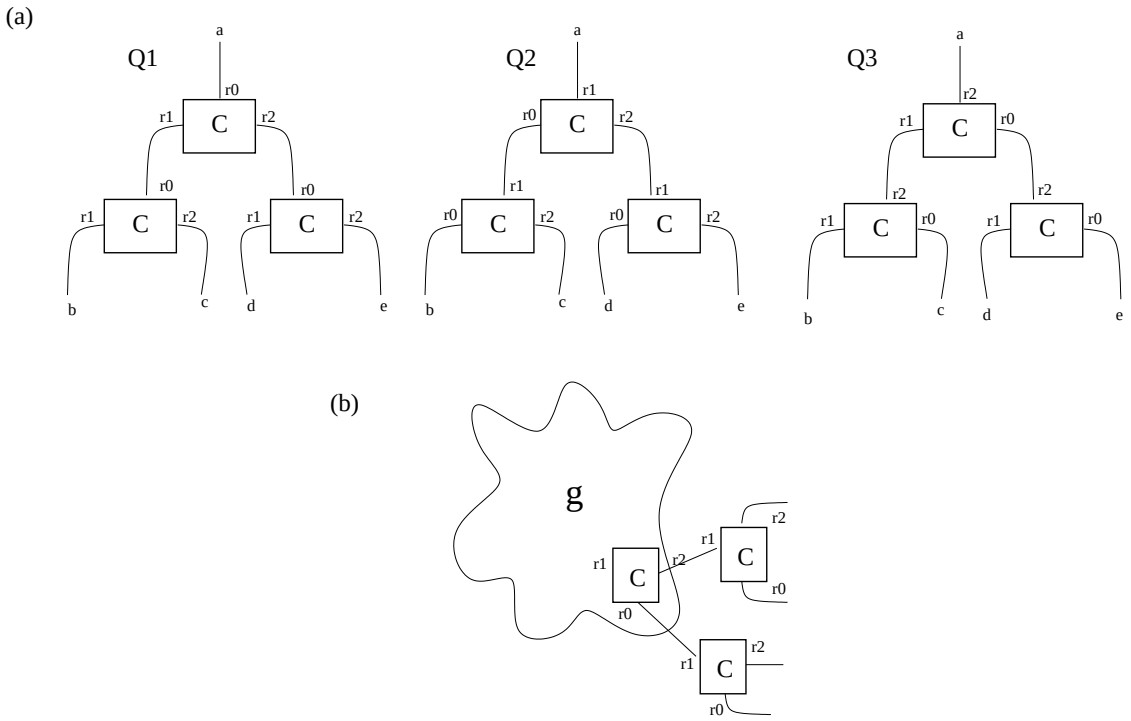


Figure 7: Agglutination counterexample

wrong choice (since we only have the locally available information to use in making the choice), the blocked pattern may be the only one that applies.

For example, consider any grammar whose patterns contain the three patterns from part (a) of figure 7 (these patterns differ only in their port identifiers). By the above method, we can agglutinate to each pattern by adding the agglutinations and rewrites to represent the two-vertex subpatterns of the three patterns (three subpatterns in all). But these subpatterns may overlap with another pattern, so that forming the overlapping agglutinations will make it impossible to form the agglutination needed to intermediate the working rewrite.

One can solve any given working graph by providing appropriate larger agglutinations, however it is impossible to solve all graphs by adding these larger and larger agglutinative vertices, since for the above grammar no finite set of agglutinative vertices suffices. To see this consider the agglutination representing the largest tree-form graph k consisting solely of 'C' vertices: take an arbitrary extremal vertex of the graph; then one may create an overlapping pattern by adjoining two new 'C' vertices (in a way determined by the port identifier of the pattern that is an internal port of the maximal graph), as shown in part (b) of figure 7. Since the agglutination is reachable, the graph consisting of the agglutination of the pattern with the two leftover vertices of the pattern is a possible reachable graph: to match the pattern one must have an additional intermediate agglutination, contradicting the premise that the current agglutinative vertex was the largest tree-form agglutination

consisting entirely of 'C' vertices.

The approach we shall take to avoiding this form of deadlock is to introduce a new form of intermediate rewrite, whose purpose is to gather together sufficient information about the neighbouring structure of the graph to tell what patterns occur in the graph: as a result we need not perform any speculative agglutinations, and so we do not need to worry about this blocking problem. This information is a kind of metadata: the existence of particular pieces of data is meant to stand for certain properties of the graph being satisfied, and the construction of the encoding ensures that the information so represented is accurate.

The method we adopt to representing this information is to give explicit additional graph structure, in the form of additional pattern vertices, each of which stores information relevant to the possible realisation of a pattern of the general grammar. The pattern vertices connect to additional ports, called *pattern ports* of the working vertices: each pattern port of a given working vertex is connected to the corresponding *match port* of the pattern vertex for the patterns it might occur in; edges consisting of a match port and a pattern port are called *match–pattern edges*. The attribute for pattern vertices is an edge set, representing the set of edge matches of the pattern so far been observed. Patterns may not yet have observed their full complement of vertices: in this case the unmatched match ports are connected to a dummy vertex, called a *bung vertex* (a kind of message vertex).

Then information about the state of the graph is passed from working vertices to pattern vertices by means of a series of *edge raising* interactions. A pair of working vertices interact on their shared edge, generating an edge raising vertex which is directed towards all the pairs of pattern in which the edge may occur. These edge raising vertices act to mediate the adding of an edge to these patterns; potentially merging disjoint patterns. The working vertices must remember which edges have interacted to create the edge raising messages so to avoid livelock (endlessly generating the same edge raising vertices). This is achieved by adding a port set attribute to each working vertex.

The reader may wonder at this point: why the complexity of an additional class of pattern vertices? It would be possible to represent the information contained there by means of attributes representing local graph structure in the working vertices, so avoiding the need for the message passing interactions to communicate this information indirectly. The principal reason is that the pattern vertices make life easier when it comes to updating information in the wake of a work rewrite. Whereas when information is stored in special pattern vertices, false information can be updated in a single interaction, with non-local information in the working vertices, it will take several steps to eliminate false information. Designing systems along these lines is difficult, since there is very little one can take for granted about the accuracy of information in the working vertices: an especial danger is livelock, since whilst one part of the graph engages in interactions to eliminate some false information, out-of-date information elsewhere is busy propagating itself again.

The raising of edge information to the patterns is the most complex part, but there is still more work to be done. Now we have information on the patterns that exist, we must form the agglutinations of working nodes, which we do by agglutinating the working vertices to the pattern vertices, a process we call *locking the pattern*. Since several patterns may be overlapping, we must decide which pattern has precedence. In general this belongs to

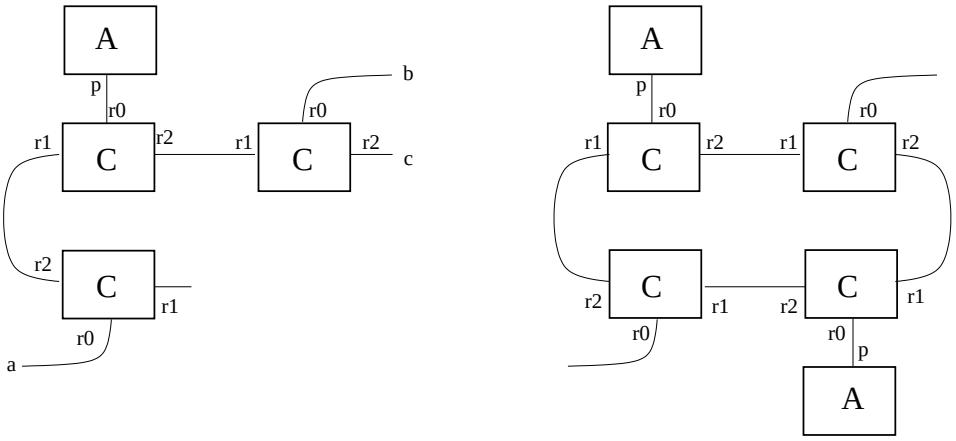


Figure 8: Symmetrically self-overlapping pattern

the tricky class of consensus problems (it isn't enough to specify a precedence ordering on patterns, since a pattern may overlap itself in a perfectly symmetric manner, as in figure 8); we solve this by means of a terminating back-off protocol.

Finally, once the pattern is locked, we need to apply the work rewrite, and spread updates of the information to all pattern vertices that were overlapping the locked pattern.

So let us see this in action: we start with the two rule grammar from figure 5. We will need to choose a precedence between patterns, specify that $PA < PB$ The target grammar will have the following vertices:

- We will have six working vertices, three just as the three vertices of the source grammar (ie. the plain working vertices) and three with additional pattern ports. The new working vertex 'A' will have just one pattern port, corresponding to the occurrence of A in rule PB; 'B' will have three pattern ports, two for the left and right occurrences in PA and the third matching the occurrence in PB; and 'C' will have two pattern ports, one for the occurrence in each pattern. The working ports are the ports of these vertices that are not pattern ports.

- There will be two pattern vertices, corresponding to the two patterns PA and PB; each will have three match ports, since both have three vertices in them.

- There will be twelve agglutinative vertices, one for each non-empty connected subgraph of the patterns of G. These agglutinations will have working, match and pattern ports.

- There will be two 'fire rewrite' vertices, one for each distinct pattern of G.

- There will be an assortment of message-passing vertices, six basic, and one composite vertex for each pattern, making eight in all.

Let us examine the behaviour of the encoded grammar, by seeing it work on the working graph in figure 9. This graph contains two overlapping redexes, one matching the PA

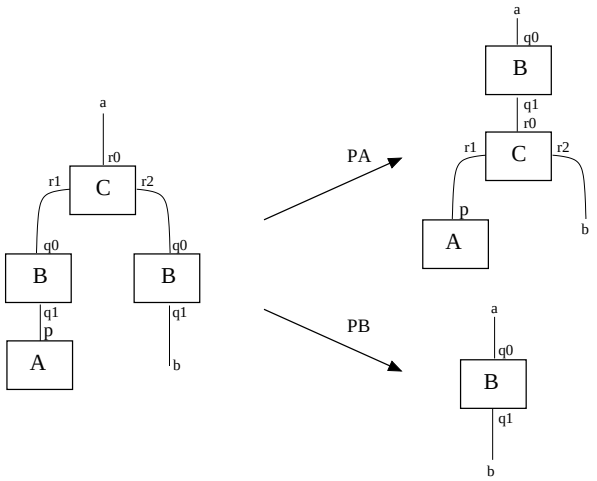


Figure 9: Example source graph

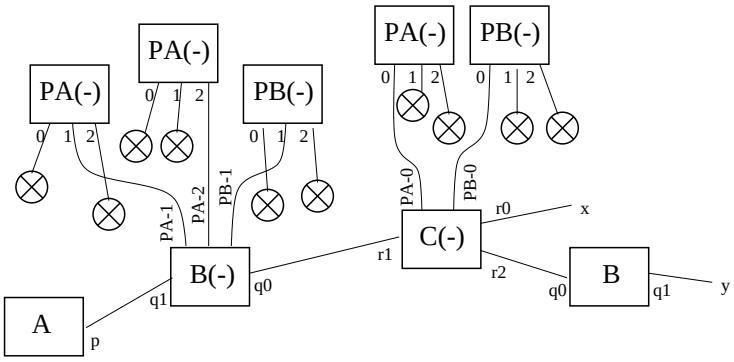


Figure 10: Creating pattern nodes

rewrite, and one matching the PB rewrite. The translation of this graph is just the identity (we have that $\Sigma_H \subseteq \Sigma_G$).

Each vertex is a single node redex: each node expands to a group of the matching pattern ports. After expanding a 'B' and a 'C' vertex, the working graph is as shown in figure 10, with one edge-raising rewrite available. Applying this rewrite creates two edge-raising messages, as shown in figure 11 (the edges $e1, e2$ are, by coincidence, the same edge $\{\langle 0, r1\rangle, \langle 1, q0\rangle\}$). After four rewrites (completing each edge-raising takes two rewrites: an intermediate rewrite is needed to interact with both involved patterns), the two PA pattern vertices and two PB vertices are merged to obtain figure 12.

One continues in this manner until all the possible edge-raising rewrites are performed, to obtain the graph in figure 13. At this point, two of the pattern vertices are complete and so can begin to lock their vertices; notice also that there are still three other pattern vertices that do not match any possible redexes in the working graph.

If the complete PA pattern locks its 'C' vertex and the non overlapping 'B' vertex, and

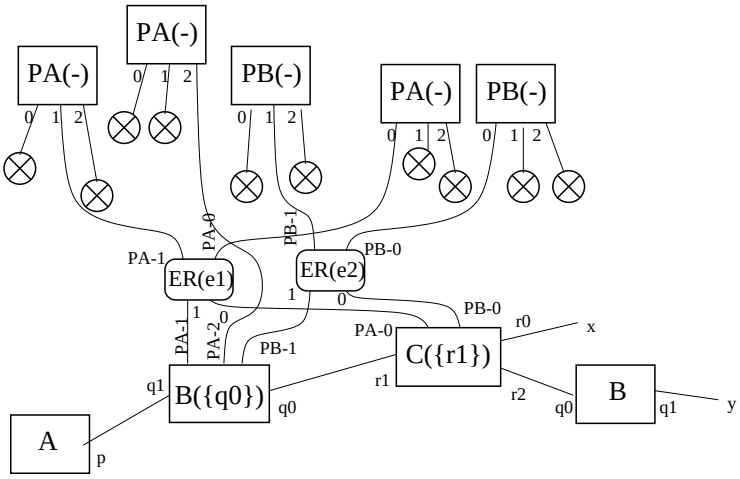


Figure 11: Raising an edge

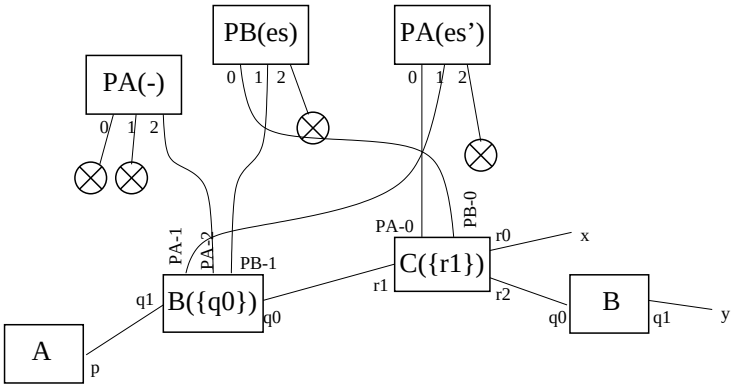


Figure 12: First edge raised

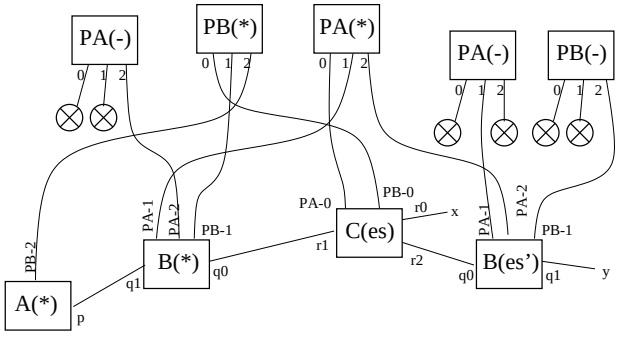


Figure 13: All edges raised

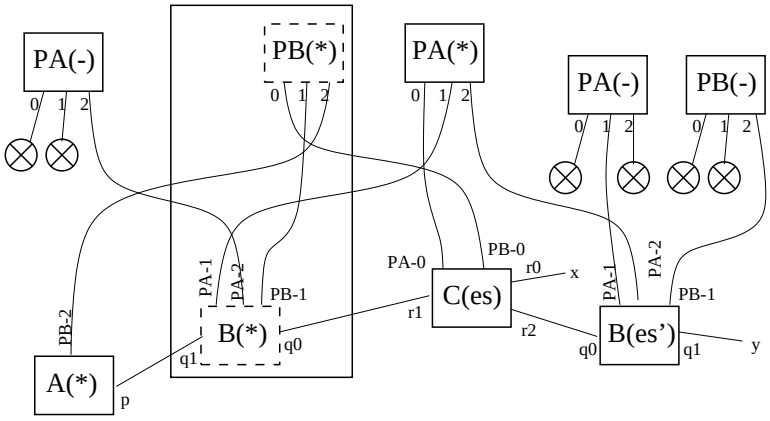


Figure 14: Pattern conflict

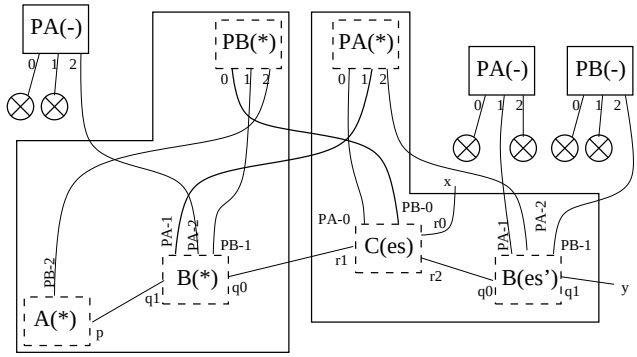


Figure 15: Stealing the lock

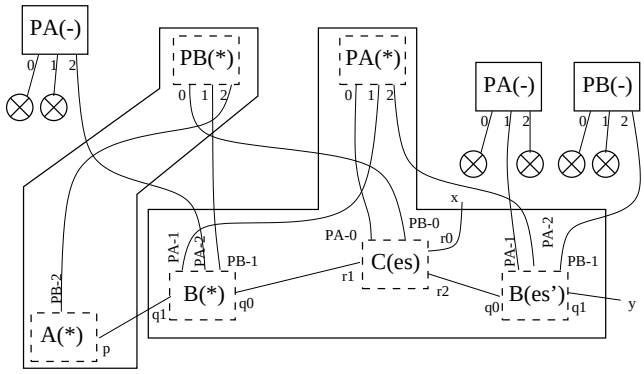


Figure 16: Ready to rewrite

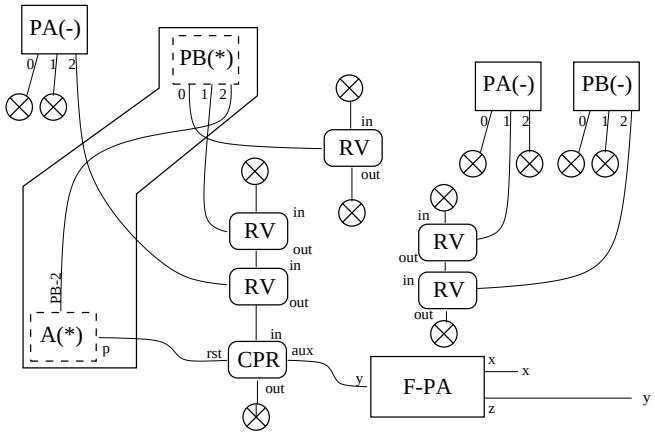


Figure 17: Propagate update messages

the PB pattern locks its 'B' and 'A' vertices, as shown in figure 14, then the two patterns find themselves in conflict: to proceed each must lock a vertex locked by the other. This is shown by the two edges going from the pattern vertex of an agglutination to a working vertex of the other. To resolve this, we make use of the pattern precedence we specified: since PA has higher precedence than PB, it can 'steal' one of the vertices which the other pattern vertex has locked, as shown in figure 15.

The pattern PB is then able to lock the final 'A' working vertex. Since it now has all of its associated working vertices locked, it is ready to fire the main rewrite. To do this correctly, it is necessary to spread update information to the other two pattern vertices that match the locked working vertices. The rewrite takes the graph in figure 16 to that in figure 15: special *port reset* messages are put on all the edges matching working vertices that had that port set, and *remove vertex* (RV) messages are put to disconnect the overlapping pattern vertices. These are joined in vertical *update chains* (one for each working vertex of the locked pattern), which prevent a possible race condition: were the PR messages associated with a locked working vertex to be discharged before the RV message, the working edge on

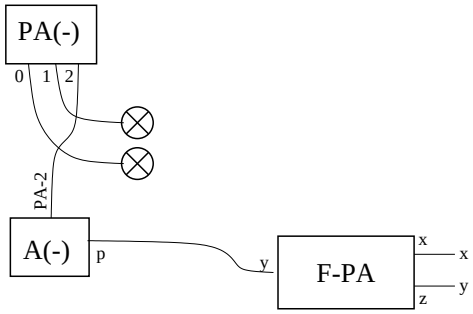


Figure 18: Update messages discharged

which the PR message stands could be raised in an A2 interaction, which could propagate up to the pattern match only to be annihilated by the now outdated RV message. To avoid this, all of the PR messages and RV messages associated with a common working vertex are chained together in such a way that none of the PR messages can fire until all of the RV messages are discharged, the so chained PR messages are called CPR messages (chained port reset messages).

The RV message interacts with the pattern it faces to break open locked patterns, and remove all edges of the edge set that borders the match port on which RV message is (it destroys the pattern if its edge set is empty), whilst the PR message resets the working port of the working vertex, so that it can re-raise this edge. After applying all the message rewrites, the working graph is as shown in figure 18: the vertex denoted F-PA can then be replaced by the appropriate replacement graph (ie. the 'B' and 'C' vertices from the replacement rule of PA from figure 5, the port labels of the F-PA vertex are the labels of the pattern of that rewrite rule)

The above, though correct, omits important details; most importantly it does not note that after having performed all of the edge-raising rewrites we do not know that the complete pattern vertices correspond to pattern matches in the underlying graph: since we have matched only internal edges, all we know is that there is a partial homomorphism from the pattern to the graph, whilst we need a full homomorphism to ensure the existence of a redex. Consequently it is possible that the class B rewrites will start locking to a pattern mismatch - we need rewrite rules to spot this (this case will occur when the same working vertex corresponds to two different vertices of the pattern match), and additional pattern vertices that record that the pattern has failed (we need two in the above case). We'll treat this properly when we introduce the formal definition of the grammar in the next section.

Another omission is that the graph chosen and its patterns are trees: this special case avoids the circumstance that we raise all the edges in a loop. If we suppose there to be an 'upwards' orientation on graphs, with working vertices at the bottom and upward paths passing through edge raising vertices from match ports to pattern ports, then the topmost vertices of the graphs are always pattern vertices. When the graphs and patterns contain circuits, however, it is possible for this property to be violated, with intermediate message vertices standing at the top of the graph.

5 Constructing the basic grammar

To define the grammar, we will need to compute the following functions and relations for the source grammar G:

1. $\mathsf{PM}(vt)$: the set of vertex matches in the set of patterns. These matches are represented as pairs, the first projection being the identifier of the rewrite, and the second being the vertex number in the pattern of the rewrite.
2. $\mathsf{PE}(pt_0, pt_1)$: the set of pairs of vertex matches in the patterns which specify internal edges of the pattern whose ends are of the given port type, and whose ends are different vertices.
3. $\mathsf{PE}_l(pt_0, pt_1)$: as above, but where the two ends of the edge are the same vertex in the pattern.
4. $<_{\text{pat}}$: a total order on patterns, chosen in any way.
5. $\mathsf{basis}(es)$: the set of vertex identifiers of the vertices at each end of the edge set es:
$$\mathsf{basis}(es) \mathrel{\hat{=}} \{U, V \mid \{\langle U, pt\rangle, \langle V, qt\rangle\} \in es\}$$
6. $\mathsf{PP}(es)$ (the *pattern partition function*): in case es is connected or empty, the function yields the singleton $\{es\}$, otherwise it yields the set of the connected subsets of es (so that $\bigcup(\mathsf{PP}(es)) = es$ and if $es_0, es_1 \in \mathsf{PP}(es)$ then either $es_0 = es_1$ or $es_0 \cap es_1 = \emptyset$, and $\emptyset \notin \mathsf{PP}(es)$).

EXAMPLE 21 For the grammar given in figure 5 we have:

$$\mathsf{PM}(B) = \{PA - 1, PA - 2, PB - 1\}$$
$$\mathsf{PE}(\langle C, c1\rangle, \langle B, b0\rangle) = \{\langle PA - 0, PA - 1\rangle, \langle PB - 0, PB - 1\rangle\}$$

We then construct the set of vertices and rewrites of the encoded grammar: these are divided into five classes:

1. The class 'A' rewrites create the pattern vertices and carry out edge raising.
2. The class 'B' rewrites lock working vertices to complete patterns, and settle conflicts between overlapping patterns. They also recognise those complete patterns that are mismatches, and create 'failed' pattern vertices.
3. The class 'C' rewrites propagate update information from fully locked rewrites, and prepares the fully locked pattern for the 'F' rewrite.
4. The class 'F' rewrites are the work rewrites of the grammar, ie. they introduce the replacement graphs. The 'F' stands for 'fire rewrite'.
5. The class 'M' rewrites resolve the message-passing vertices that may be created by class 'A' and class 'C' rewrites.

Finally, we introduce the translation homomorphism and prove that it has the desired properties of livelock and deadlock freeness, and preserves concurrency. This will be in the next section.

The diagrams used in this section stand in place of giving explicit set comprehensions to define the rules. Whilst this increases the comprehensibility of the definitions, we must be careful to avoid ambiguity. The following conventions are used in the diagrams; where there still exists possibility of ambiguity, we make the intention precise in the accompanying text.

- Vertex identifiers are left unspecified. Each vertex appearing has a unique vertex identifier, as do vertices implicitly specified by ellipsis or by graph clouds (indicated by g,h). Port identifiers are normally left unspecified, except where this would cause ambiguity. We shall informally refer to vertices by their symbols.

- The letters 'A','B' stand for the symbols of working vertices. If the letter appears by itself, it stands for a plain working vertex, if it has an attribute following the letter in brackets, it is a working vertex with pattern ports. The attribute is the port set, with '-' indicating the empty port set. Ports on the top of the vertex normally indicate pattern ports, whilst those to the side normally indicate working ports.

- The letters 'P','Q','R' stand for the symbols of the pattern vertices, and is followed by the edge set attribute in brackets. '-' indicates the empty edge set, '*' the complete edge set (ie. with all internal edges raised), and '#' indicates that the pattern is a failed pattern vertex (necessarily with complete edge set).

- The letters 'a','b', ..., 'h' indicate the free ports of the graph. If they are followed by an ellipsis '...' then they represent a vector of free ports. A special convention is followed in rules A2 and A3, there we indicate free ports implicitly with a label of the form PU1..n, which describes the label of the facing pattern port, that is, the ith free port of the vector faces the pattern port Pi-Ui, so we may specify the pattern Pi and the corresponding match port Ui.

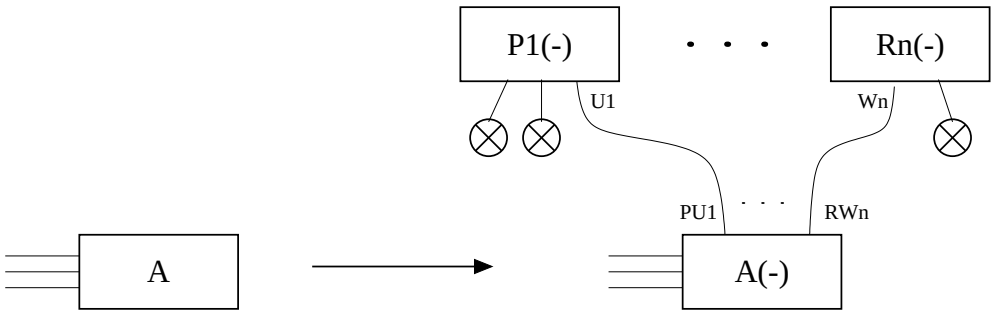
- Vertices and graphs appearing inside agglutination vertices are indicated by dotted line boundaries, these are not vertices of the pattern or replacement graph, but instead indicate vertices of the graph that the agglutination vertex stands for. Edges that occur entirely within the agglutination are edges of the represented subgraph; whereas if the edge crosses the boundary of the agglutination vertex, it is part of the pattern or replacement graph. We follow the convention described on page 17 for the symbol and port identifier of agglutinative vertices.

Class A rewrites

The class A rewrites act just on patterns involving just working vertices, both the plain working vertices, and the working vertices with match ports. The working vertices with match ports possess an attribute, which ranges over the subsets of the port types of the working vertex. There are class A rewrites of three forms, $A1 - A3$.

For the sake of introducing the vertex types and rewrites in a graphical form (rather than as set comprehensions), and without loss of generality, we will introduce some conventional simplifications. The working ports of working vertices are drawn to the left and right of the vertex, and a bundle of three unlabelled working vertices indicates 'all the other working ports'; if it is not clear which ports on the pattern side of a rewrite match up with those on the right hand side, a port vector (eg. $\vec{p}$) will be displayed. Pattern ports are drawn coming from the top of vertices, and match ports from beneath.

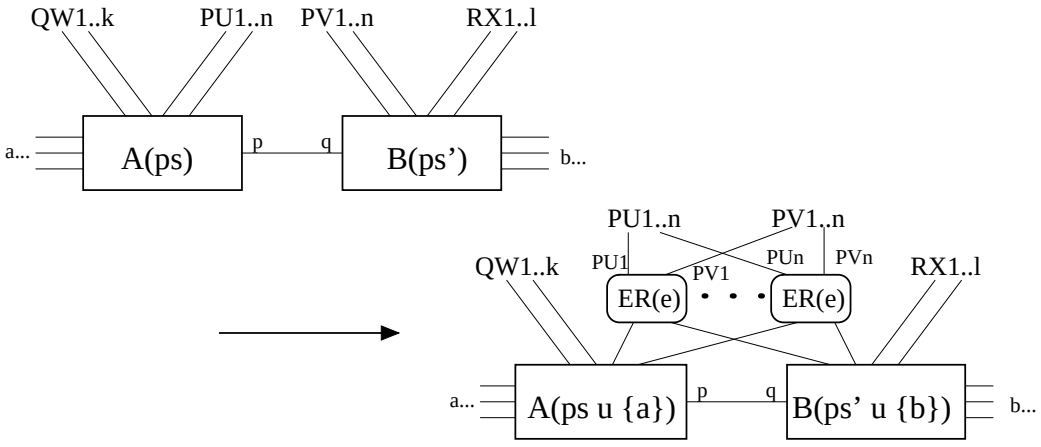
A1. For each $A \in \Sigma_G$, rewrite the plain working vertex to a graph consisting of the working vertex with pattern ports and the pattern vertices, given $\mathsf{PM}(A) = \{P - i, Q - j, R - k\}$, one of whose match ports are connected to the working vertex, ll the others are connected to bung vertices:



A2. For each pair of port types $\langle A, p\rangle$, $\langle B, q\rangle$, we have a conditional rewrite which fires in case $p \notin ps \wedge q \notin ps'$, and creates edge raising vertices according to a partition of the patterns that include the edge ($\vec{P}$) and those that don't ($\vec{Q}, \vec{R}$), specified as follows:

$$\mathsf{PE}(\langle A, a\rangle, \langle B, b\rangle) = \{\langle P0 - i_0, P0 - j_0\rangle, \ldots, \langle Pm - i_m, Pm - j_m\rangle\}$$
$$\mathsf{PM}(A) = \{P0 - i_0, \ldots\} \cup \{Q0 - k_0, \ldots\}$$
$$\mathsf{PM}(B) = \{P0 - j_0, \ldots\} \cup \{R0 - l_0, \ldots\}$$

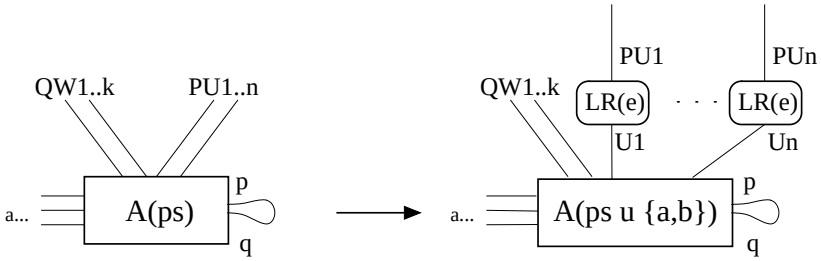
(where we let $e \mathrel{\hat{=}} \{\langle A, a\rangle, \langle B, b\rangle\}$)



A3. For each pair of distinct port types sharing the same vertex type $\langle A, p\rangle$, $\langle A, q\rangle$, we specify the special edge raising rewrite that applies to loops (ie. edges from a node to itself), and is conditional, firing in case $p, q \notin ps$. Similarly to above, we partition the pattern ports into those that include the loop ($\vec{P}$) and those that don't ($\vec{Q}$), as given:

$$\mathsf{PE}_l(\langle A, a\rangle, \langle A, b\rangle) = \{P0 - i_0 \ldots, Pm - i_m\}$$
$$\mathsf{PM}(A) = \{P0 - i_0, \ldots\} \cup \{Q0 - k_0, \ldots\}$$

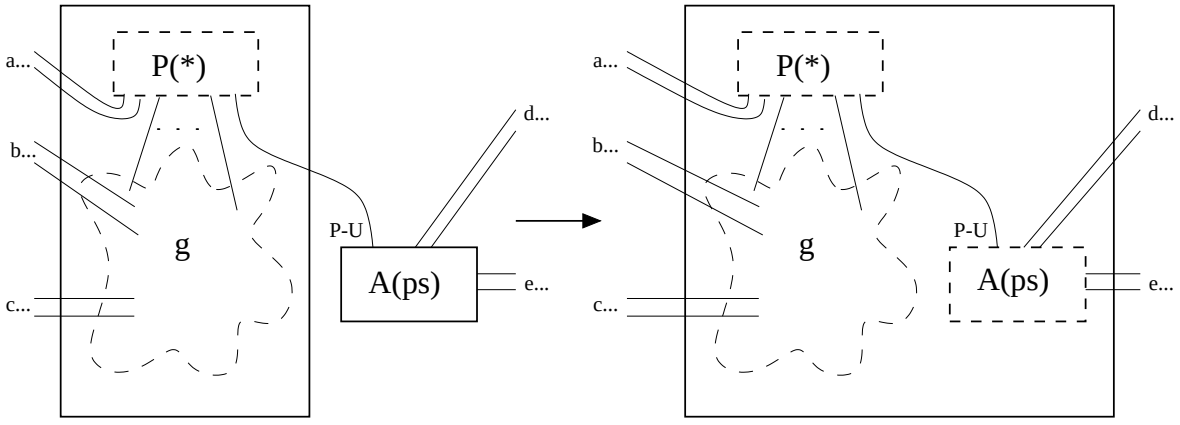
(where we let $e \hat{=} \{\langle A, a\rangle, \langle B, b\rangle\}$)



Class B rewrites

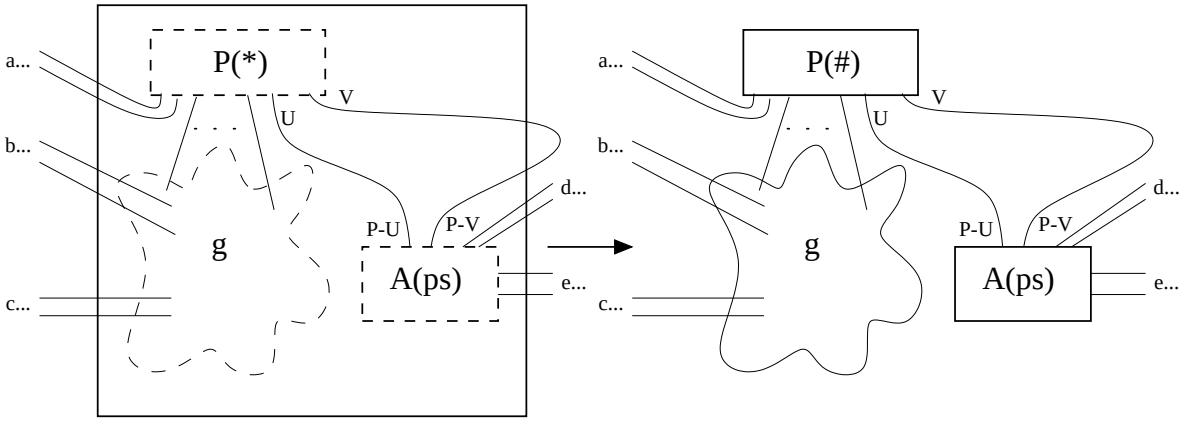
The class B rewrites act on patterns involving working vertices with match ports, complete patterns (ie. patterns whose edge set includes all internal edges, which we indicate by a '*'), and agglutinations of complete patterns and locked working vertices. The attributes of these agglutinations includes the attribute of all of the locked working vertices. There are class B rewrites of three forms, $B1 - B3$. The graph clouds shown inside these agglutinations are all discrete.

B1. (Lock working vertex) There are rewrites of this form for all interactions of a complete, partially locked pattern (partially locked in this sense covers normal complete patterns with no locked working vertices) on its match port V and a working vertex on its complementary pattern port P-V.

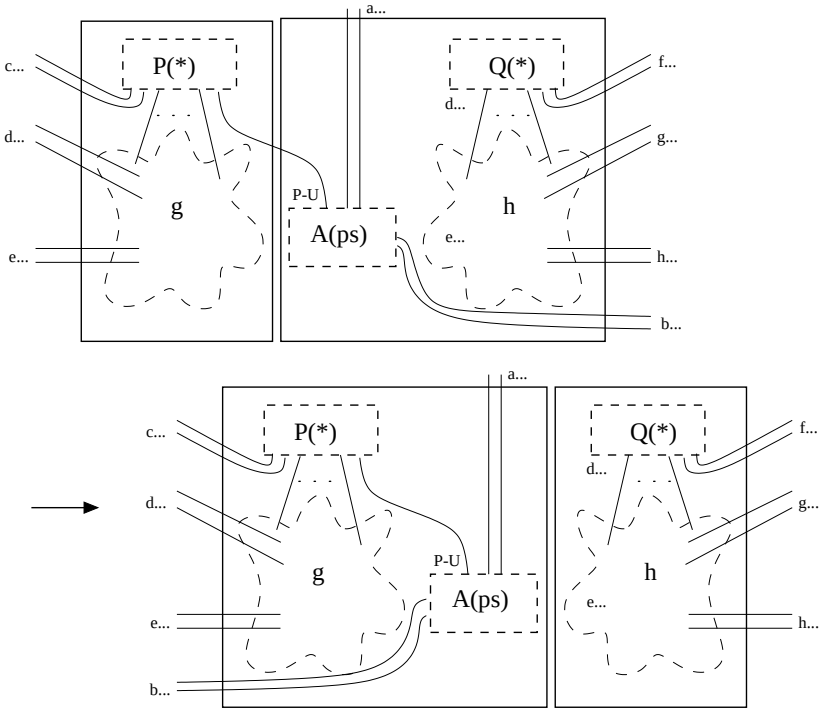


B2. (Spot failed pattern) For each complete pattern that some pair of working vertices of vertex type 'A', and which has locked the first of these to match port U, but has not locked

along the second match port V. The pattern, a looping pattern, indicates that this second match port V connects to the pattern port P-V of the already locked vertex, indicating that the two vertex occurrences are the same, and so the pattern has failed.



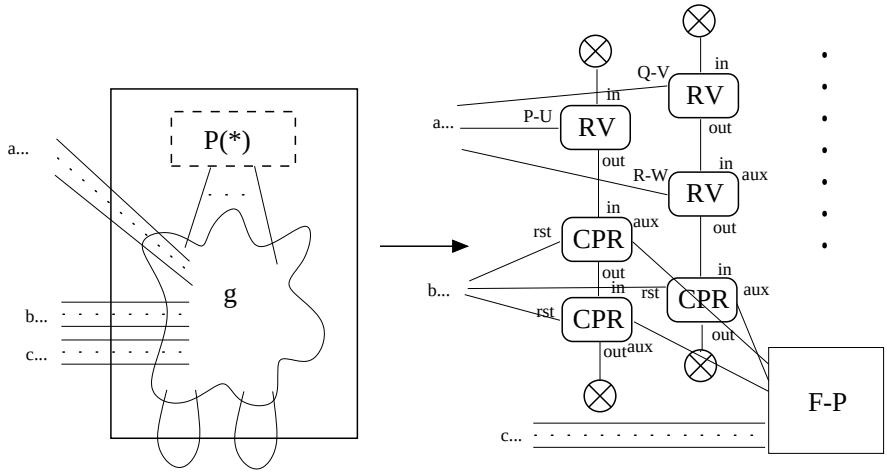
B3. (Steal lock) This pattern applies between two partially locked patterns $P1$ and $P2$ in case either (i) $P1 = P2$ and the pattern agglutination without the contested working vertex A has at least as many locked working vertices as the one with the contested vertex, or (ii) $P1 \neq P2$ and $P1 <_{\text{pat}} P2$.



Class C rewrites

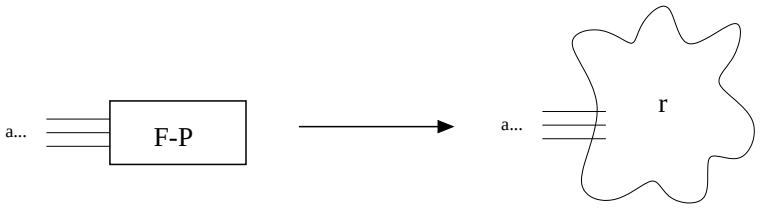
The class C rewrite acts on fully locked patterns, to create a 'fire pattern' vertex that stands for the pattern graph of a redex. The two rewrites can easily be transformed into a single rewrite by eliminating the intermediate symbol: breaking it into two is to avoid compounding the complexity in the first rewrite. The first class C rewrite introduces two kinds of messages, the match annihilation messages and the port reset messages.

C. (Propagate updates) One loop absorption rewrite for each fully locked pattern. The internal edges of the pattern are the loops of the locked pattern vertex that are absorbed. The right-hand side shows the fire pattern vertex, where the external ports facing the working vertices of the locked pattern have CPR messages attached in case the port was set in the corresponding locked working vertex, and the external ports facing the pattern ports are sent RV messages. The PR messages are organised into *update chains*, one for each working vertex of the locked pattern as described in section four. The working edges of the pattern g fall into three classes, either (i) they are loops of the agglutination (ie. both ends are the agglutinated vertex), in which case they are the loops absorbed by the pattern, indicated by the dangling loops at the bottom of the pattern, or (ii) they are non-looping edges attached to ports whose represented vertex has this port set in its port set (matching the free ports b...), or (iii) they are non-looping edges attached to ports whose represented vertex does not have this port set in its port set (matching the free ports c...).



Class F rewrites

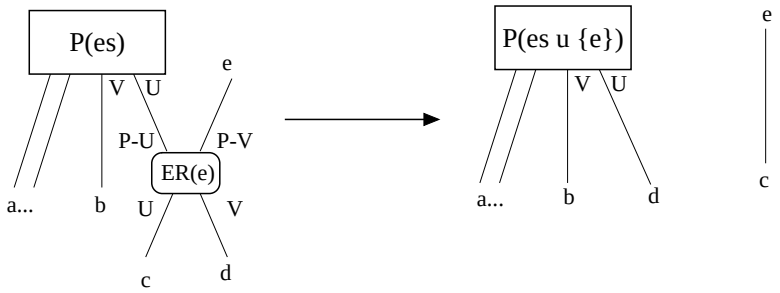
F. (Fire rewrite) For each rewrite rule of G, we have a rule that replaces the 'fire pattern' vertex associated with the pattern of the vertex with the replacement graph r.



Class M rewrites

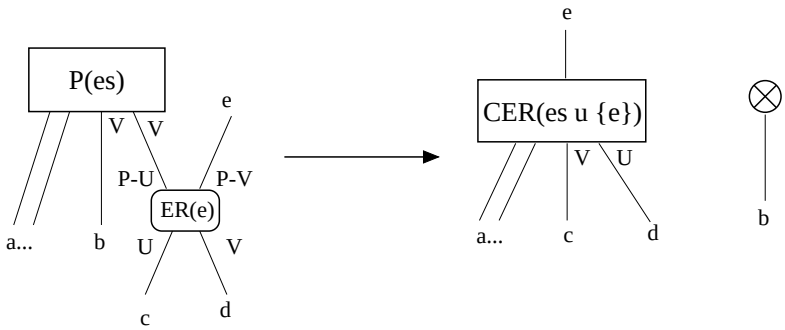
The last and most complex class of rewrites are the message passing rewrites, which resolve the messages created by the class A and class C rewrites.

M-ER1a. (Edge raising) When either pattern port of an edge raising message with edge attribute e faces the complementary match port of a pattern vertex with edge set attribute es, and $\mathsf{basis}(\{e\}) \not\subseteq \mathsf{basis}(es)$:

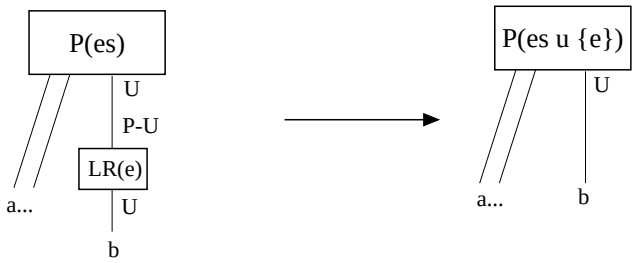


M-ER1b. (Conditional: Edge raising) When either pattern port of an edge raising message with edge attribute e faces the complementary match port of a pattern vertex with edge set attribute es, and the following conditions apply:

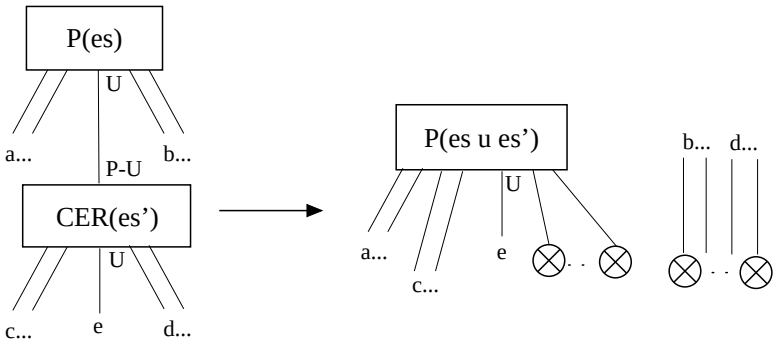
1. $\mathsf{basis}(\{e\}) \subseteq \mathsf{basis}(es)$.
2. $e \notin es$



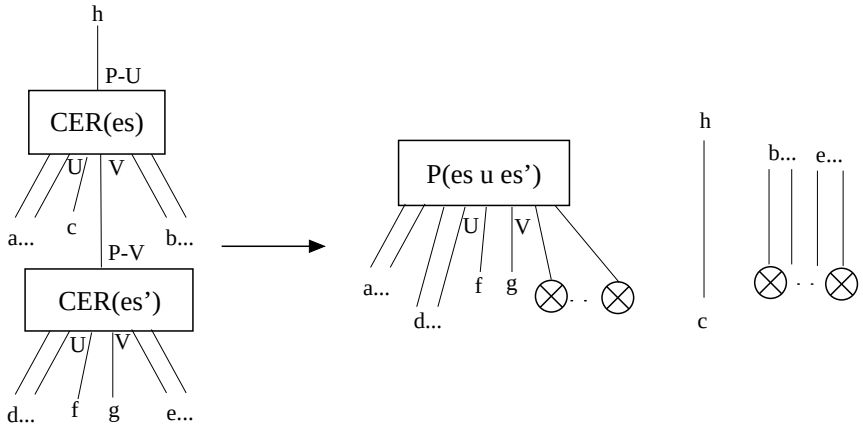
M-ER2. (Loop raising) If the pattern port of a loop raising message (LR message) faces the complementary match port of a pattern vertex, then the following rewrite rule applies:



M-ER3. (Composite edge raising) If the pattern port of a composite edge raising message (CER message) faces the complementary match port of a pattern vertex, and we have that $es \cap es' = \emptyset$ then the following rewrite rule applies:

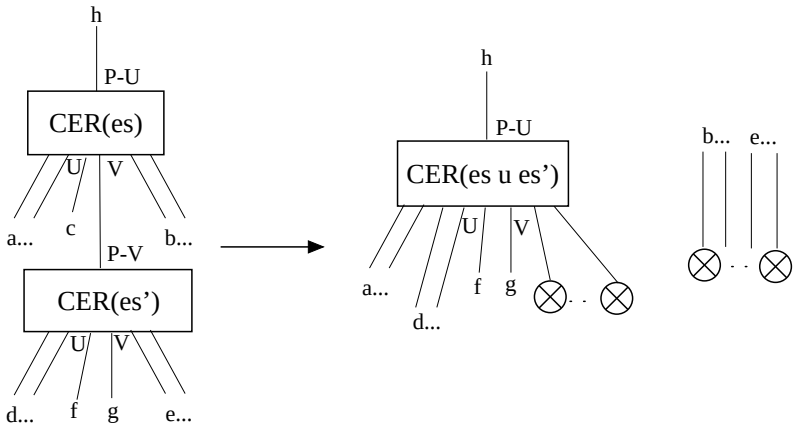


M-ER4a. (CER merging) If the pattern port $P - X$ of a CER message with edge set es faces the match port X of a CER message with edge set es' and pattern port $P - Y$ (we insist $X \neq Y$), and $Y \notin \mathsf{basis}(es)$, then the following rewrite rule applies:

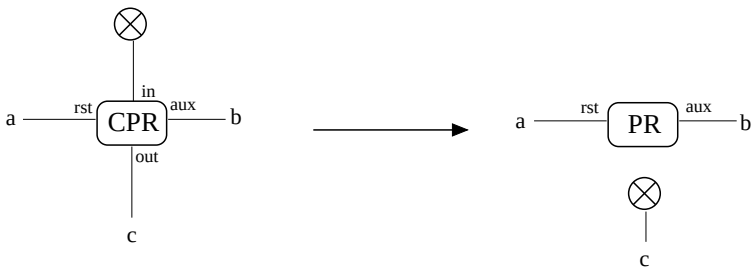


M-ER4b. If the pattern port $P - X$ of a CER message with edge set es faces the match port X of a CER message with edge set es' and pattern port $P - Y$ (we insist $X \neq Y$), but

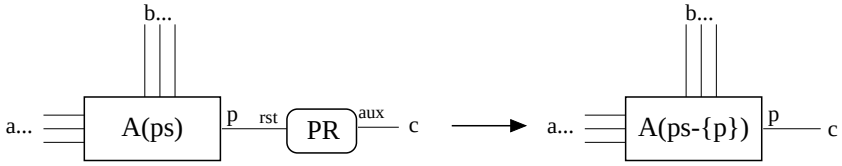
$Y \in \mathsf{basis}(es)$, then the alternate rewrite rule applies:



M-PR1. (Unchain port reset message) For each chained port reset message (CPR message) facing on its incoming chain port some bung node:



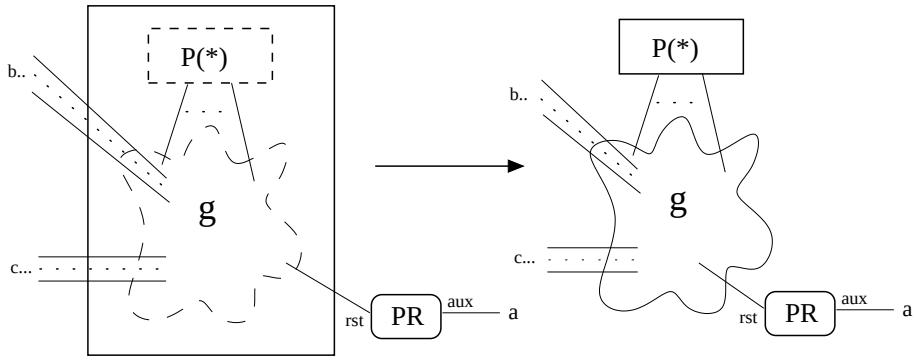
M-PR1. (Reset working port) For each unchained PR message (PR message) facing on its principal port any working port of a working vertex:



M-PR2. (Annihilate PR message) Two PR messages face on their principal ports.

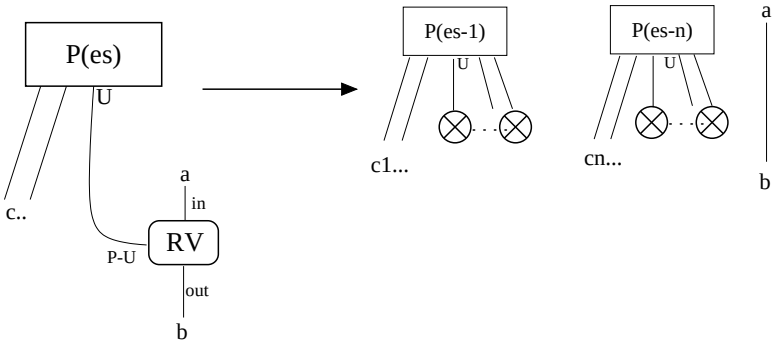
M-PR3. (Explode locked pattern I) For each pattern port of a PR message facing a locked working vertex the complementary match port of a pattern vertex with locked

working vertices.

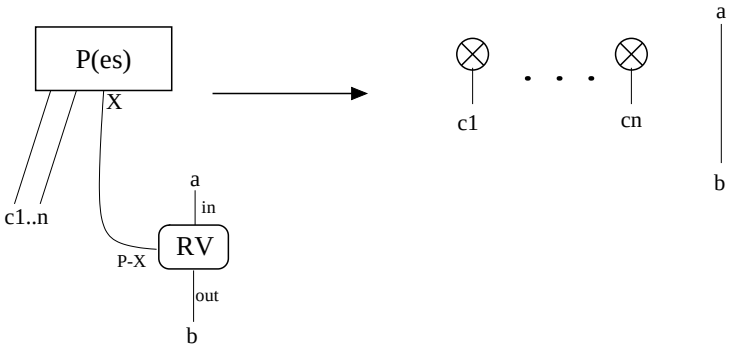


M-RV1a. (Remove vertex) For each pattern port of a remove vertex message (RV message) facing the complementary match port U of a pattern vertex with non-empty edge set attribute es, and we define:

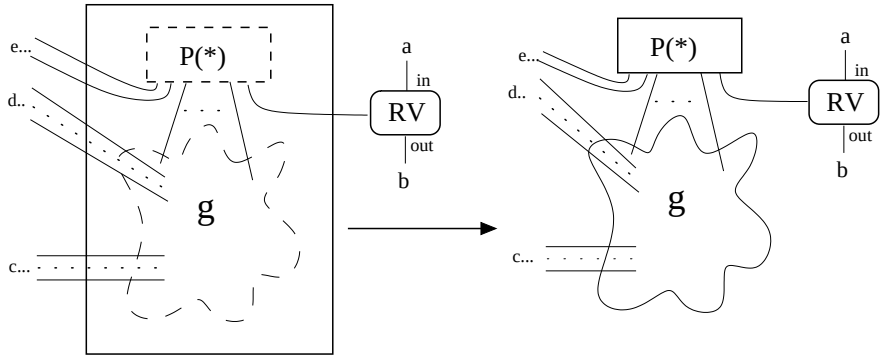
$$
\begin{aligned}
es' &\hat{=} \{e' \mid \nexists pt.\langle U, pt\rangle \in e'\} \\
\{es_1, \ldots, es_n\} &\hat{=} \mathsf{PP}(es') \\
\vec{V_i} &\hat{=} \mathsf{basis}(es_i) \text{ for } 1 \leq i \leq n \\
\vec{W} &\hat{=} \{W_j \mid W \in \mathsf{basis}(es) \wedge \nexists 1 \leq i \leq n. W_j \in \mathsf{basis}(es_i)\}
\end{aligned}
$$



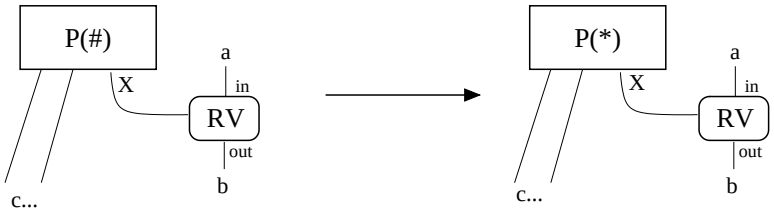
M-RV1b. As above, for each pattern port of an RV message facing the complementary match port V of a pattern vertex with empty edge set attribute es:



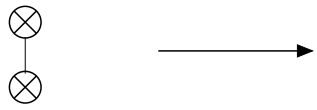
M-RV2. (Explode locked pattern) For each pattern port of an RV message facing the complementary match port of a pattern vertex with locked working vertices.



M-RV3. (Reset dead pattern) For each pattern port of an RV message facing the complementary match port of a dead pattern vertex.



M-GC. (Garbage collection) If two bung vertices face each other, then they annihilate (that is, they have the empty graph as their replacement graph).



6 Definition and properties of the translation

Recall from section three that to give an translation of SPGG G into SPGG H we must give an embedding map F and two interpretation maps $R_{\mathcal{G}}$ and $R_{\mathcal{R}}$. The translation is constructed as follows, where the *expanded form* of a graph is the graph obtained by replacing all of the pattern–vertex agglutinations with the graphs they stand in for, and all 'fire rewrite' symbols with the pattern of the associated rewrite rule.

1. The embedding map is just the identity (this is well defined because $\Sigma_G \subseteq \Sigma_H$).
2. The graph interpretation map R takes each graph of H to the graph obtained from its expanded form by applying all message passing rewrites until the graph has no messages left (possibly the elimination of rules is not deterministic or has messages in the normal form, in which case R is left undefined in this case), and then restricting it to its working vertices and working edges, throwing away all port set attributes and match ports and all other vertices and edges.

3. The partial rule interpretation map $R^{\rightarrow}$ is defined to have domain the F rewrites; for each rule r from the F rewrites it is defined to be the associated rewrite of G, selecting the obvious pattern match.

On the free grammar given by the rules before, this is not an translation, since working ports need not occur in working edges, and so the graph interpretation map is not total. So to obtain the translation, we must first construct a filter for H that ensures the conditions for the above construction to be an translation (over this filtered grammar we shall have that $\rightarrow_M^1$ is strongly-normalising and confluent). Once this is done, we shall prove livelock and deadlock freeness, and prove the translation is concurrency conserving by constructing a functorial translation.

We shall define a number of structural properties on graphs: we shall refer to vertices and edges occurring within agglutinations *transparently*, by which we mean we talk about graphs as if the agglutinations were exploded (this is the *B-expanded* form, where 'fire pattern' symbols are not expanded).

DEFINITION 22

1. A *match path* in a graph g is a path of g consisting entirely of identical match ports (ie. each match port has the same label). A match path is *maximal* if it is not a strict subsequence of any other match path.

2. A graph is *message free* if its only messages are bung vertices facing the match ports of pattern vertices.

3. A graph is *well-formed* if:

   (a) All of its internal edges are either (i) working edges, (ie. consisting of two working ports), or (ii) complementary match–pattern edges (ie. consisting of a pattern port $P - V$ and a match port V, where the two Vs are identical), (iii) Edges between the reset or auxiliary port of a PR message and the reset or auxiliary port of another PR message, or a working port, (iv) chain edges (ie. consisting of either the port of a bung vertex and the chain-in or chain-out port of a chained PR message, the chain port of a RV message and the chain-in port of a chained PR message, or the chain-in port and the chain-out port of two chained PR messages), or (v) edges between a bung and either a match port of a CER message, a pattern vertex, the chain-out port of a PR message or another bung.

   (b) All of its external edges are pairs of working ports and free ports.

   (c) No match paths are circuits.

4. A graph is *regular* if it is well-formed, has no messages and its expanded form satisfies the following four conditions:

   (a) (Port set condition) Each set working port faces a set working port.

   (b) (Edge raising condition) For any working vertex attributed by ps, and any pattern vertex $P0$ belonging to pattern P with edge set es which is connected to

the working vertex on its pattern port $P - V$, let ps_0 be the set of those ports in ps which are internal ports of P. Then

$$pt \in ps_0 \Leftrightarrow \exists e \in es. \langle V, pt \rangle \in e.$$

(c) (Basis condition) If the edge set es of each pattern P is empty, then the one of the match ports of the pattern is connected to a working vertex, and all the other match ports are connected to bungs. If es is non-empty, then for all $V \in \mathsf{basis}(es)$, V is connected to a working vertex, and for all $U \notin \mathsf{basis}(es)$, U is connected to a bung.

(d) (Pattern condition) If e is in the edge set of a pattern, and $e = \{\langle U, pt_0 \rangle, \langle V, pt_1 \rangle\}$, then the working vertices connected to the match port U is connected by an edge from its port pt_0 to the port pt_1 of the working vertex connected to the match port V of P.

5. A graph g is *admissible* if it is obtainable by some series of ABCFM rewrites from a regular graph. We define the predicate $\phi_H(h)$ to be true if h is an admissible graph in $\mathcal{G}(H)$.

PROPOSITION 23 $H \hat{=} \langle \Sigma_H, \mathcal{R}_H, \phi_H \rangle$ is a filtered SPGG.

PROOF Immediate consequence of the definitions of filtered SPGG and admissible. □

PROPOSITION 24

1. Match paths in well-formed graphs always belong to a unique maximal match path, which has terminals.
2. $\rightarrow^1_M$ is strongly normalising.
3. $\rightarrow^1_{ABCF}$ and $\rightarrow^1_M$ jointly satisfy the diamond property over well-formed graphs (ie. for any s, t_1, t_2, if $s \rightarrow^1_{ABCF} t_1$ and $s \rightarrow^1_M t_2$, then there is a graph u such that $t_1 \rightarrow^*_M u$ and $t_2 \rightarrow^*_{ABCF} u$).

PROOF Part 1. One may simply chase the match ports in either direction: by finiteness of the graphs this process must reach a terminal, loop or reach a free port of the graph; by well-formedness only the first can apply.

Part 2. Strong normalisation follows immediately from the observation that the measure on graphs that consists of the number of messages in the graph nodes plus twice the number of the simple edge raising messages, and the number of the intermediate edge raising messages is reducing under all message passing rewrites (we count the simple edge raising messages twice since M-ER1b rewrites to a graph with a CER message)

Part 3. We consider the possible critical pairs of ABCF redex and M redexes. ACF redexes only have working ports and pattern ports, and so can only form critical pairs with port reset messages: by inspection the only case where the conditions could hold are (i) A2 and M-PR2 and (ii) A3 and M-PR2. In each case the critical pair is easily resolved. The B redexes do not form critical pairs with edge raising messages, since the respective rewrite rules only involve incomplete pattern vertices (by the invariant). The B redexes all form critical pairs with PR and RV messages:

1. (B1, M-PR2) Applies when the PR message faces the working vertex. One obtains a common residual by either (i) simply applying the M-PR2 rewrite, or (ii) applying the B1 rule, which creates an M-PR3 redex, whose residual then admits a series of B1 redexes to obtain the common form.

2. (B1, M-PR3) Applies when the PR message faces one of the working ports of a locked working vertex. The common residual arises by either (i) by simply exploding the M-PR3 redex, or (ii) applying the B1 rule followed by the new M-PR3 rewrite.

3. (B2, M-PR4) Applies when the PR message faces any one of the working ports. The common residual arises either by (i) applying the M-PR4 rewrite, or (ii) applying the B2 followed by the M-PR2 rewrite.

4. (B3, M-PR3) Applies in one of three ways:

   (a) When the PR message faces the contested locked vertex of the left agglutination: the common residual arises either by (i) applying the B3 rewrite, or (ii) applying the M-PR2 rewrite, letting the contested vertex be acquired by the right pattern by the new B1 rewrite, and allow the left pattern to reacquire the $\vec{A}$ vertices by a series of B1 rewrites.

   (b) When the PR message faces another vertex of the left agglutination: the common residual arises either by (i) applying the M-PR3 rewrite, then applying the B1 rewrite between the formerly contested vertex and the right pattern (ii) applying the B3 rewrite, then applying the M-PR3 rewrite.

   (c) When the PR message faces any working port of the right agglutination, the common residual arises either by (i) applying the M-PR3 rewrite, or (ii) applying the B3 rewrite, followed by the M-PR3 rewrite and then applying the B1 rewrite between the formerly stolen vertex and the left pattern.

5. (B1, M-RV2) The locked pattern is exploded the same by the RV message either before or after applying the B1 rewrite. before or after the B1

6. (B2, M-RV3) Both rewrites have the same residual.

7. (B3, M-RV2) Applies in either of two ways:

   (a) When the RV message faces a match port of the left agglutination: similar to the second subcase of the (B3, M-PR2) critical pair.

   (b) When the RV message faces a match port of the right agglutination: similar to the third subcase of the (B3, M-PR2) critical pair.

Thus all the critical pairs are resolvable, and the joint diamond property is shown to be satisfied. □

DEFINITION 25

1. The *matching terminal* of a match port of a pattern in a well-formed graph is the vertex at the end of the match path beginning at that port. The matching terminal is always a working vertex, bung or remove vertex message.

2. A *link* of an internal edge $e \hat{=} \{\langle V_0, pt_0\rangle, \langle V_1, pt_1\rangle\}$ of a pattern consists of one of the following:

   (a) Two distinct match ports of the same pattern vertex or CER message with edge-set es both of whose matching terminals A_0, A_1 are working vertices which face on edge $\{\langle A_0, pt_0\rangle, \langle A_1, pt_1\rangle\}$, and where $e \in es$.

   (b) A match port of a pattern vertex of that pattern, where $V_0 = V_1$ and the matching terminal is a working vertex and has a loop between its ports pt_0, pt_1.

   (c) Two match ports of pattern vertices or CER messages (either the same or distinct), where the matching terminals are both working vertices, and the respective match paths pass through the pattern ports of an ER message raising e.

   (d) A match port, either V_0 or V_1, of a pattern vertex or CER message whose matching terminal is a working vertex, and where the match path passes through the pattern port of a CER message whose edge set contains e.

   (e) A match port of a pattern vertex or CER message of that pattern, where $V_0 = V_1$ and the matching terminal is a working vertex, and the match path has an LR message raising e.

3. A graph is *weakly regular* if it is well-formed and satisfies these five conditions:

   (a) (Weak port set condition) Each working port is either:

      i. Unset and faces an unset working port.
      ii. Set and faces a set working port.
      iii. Set and faces the reset port of a port reset (PR) message (either chained or unchained).
      iv. Unset and faces the auxiliary port of a PR message (either chained or unchained).
      v. Unset and is an external port of the graph.

   and the auxiliary port of any port reset message (either chained or unchained) faces a working port.

   (b) (Weak edge raising condition) For any working vertex A attributed by ps, and any pattern vertex or CER message $P0/M$ belonging to pattern P with edge set es whose pattern port $P - V$ has A as its working terminal, let ps_0 be those ports of ps which are internal ports of $P0/M$. Then the following two properties are equivalent:

      i. $pt \in ps_0$ and $\langle A, pt\rangle$ faces a working port.
      ii. Some link contains the match port $\langle V, pt\rangle$ of $P0/M$.

(c) (Message condition) Each edge raising message must satisfy its respective conditions:

i. (For ER messages) Let the pattern ports of the ER message be $P - X$ and $P - Y$. The ER message must satisfy these three conditions:

A. It is not connected on both $P - X$ and $P - Y$ to another ER message raising the same edge.

B. It is not connected on either $P - X$ or $P - Y$ by a match path to a match port of a pattern vertex or CER message that contains the same edge in its edge set.

C. It is not connected on $P - X$ by a match path to a match port of a pattern vertex or CER message whose matching Y terminal is an RV message; nor vica versa with $P - Y$ and the X terminal.

ii. (For LR messages) No loop raising message is connected on its pattern port along a match path either (i) to another loop raising message raising the same edge, or (ii) to a pattern that contains the same looping edge in its edge set.

iii. (For CER messages) For any CER message $M(es)$, es contains exactly one edge e_0 connected to V. Furthermore if M has pattern port $P - U$ which is connected by a match path to the match port U of a pattern vertex $P0(es')$, then$\mathsf{basis}(es) \cap \mathsf{basis}(es') \subseteq \{U\}$. Alternatively if $P - U$ is connected by a match path to the match port U of another CER message $N(es')$ with pattern port $P - V$, then either $\mathsf{basis}(es) \cap \mathsf{basis}(es') \subseteq \{U\}$ or $\mathsf{basis}(es) \cap \mathsf{basis}(es') = \{U, V\}$ and $P - V$ is connected to the V match port of M by a match path (ie. in the latter case M and N form a loop along match–pattern edges).

iv. (For RV messages) If an RV message is connected by a match path from its pattern port to a match port U of a pattern, and the edge set of the pattern contains an edge $\{\langle U, pt_0\rangle, \langle V, pt_1\rangle\}$, then either the matching terminal of V is an RV message, or its matching terminal is a working vertex whose pt_1 is either unset, or faces the reset port of a PR message.

(d) (Weak basis condition)

i. Every match port corresponding to a vertex in the basis of a pattern vertex or CER message has either a working vertex or RV message as its matching terminal. If the basis is non-empty, then every match port not in the match port faces a bung vertex; otherwise exactly one match port has a working vertex or RV message as its matching terminal, and all the others face a bung vertex.

ii. If two distinct vertices, either pattern vertices or CER messages, belonging to pattern P each contribute a match port to a common link, and where there is no match path upon which both vertices occur, then the basis of each of their edges sets are disjoint.

(e) (Link condition)

i. No internal edge of a pattern has two equal links.

ii. For any link $\{\langle P0, U\rangle, \langle P0, V\rangle\}$ of an bridging edge $e = \{\langle U, pt_0\rangle, \langle V, pt_1\rangle\}$, the working ports $\langle A, pt_0\rangle$ and $\langle B, pt_1\rangle$ are both set and face each other, where A is the matching terminal of U and B is the matching terminal of V.

iii. For any link $\{\langle P0, U\rangle\}$ of a looping edge $e = \{\langle U, pt_0\rangle, \langle U, pt_1\rangle\}$, the working ports $\langle A, pt_0\rangle$ and $\langle A, pt_1\rangle$ are both set and face each other, where A is the matching terminal of U.

The following proposition establishes the fundamental relationship between the regularity and weak regularity condition:

PROPOSITION 26 Regularity and weak regularity are equivalent for well-formed graphs without messages.

PROOF For well-formed graphs without messages:

1. The port set condition is equivalent to the weak port set condition, since each working port faces another working port.

2. Links can only be match ports of a pattern vertex in the absence of edge raising messages, so links correspond exactly to elements of edge sets. Modulo this correspondence, the edge-raising condition is immediately equivalent to the weak edge-raising condition.

3. The message condition is vacuously true.

4. By the same correspondence between links and elements of edge sets, the first clause of the basis condition is equivalent modulo the correspondence to the weak basis condition. The second concerns links spread across two pattern vertices/CER messages: there are none in graphs without messages, so this clause is vacuously true.

5. Again, by the correspondence between links and elements of edge sets, the pattern condition is equivalent up to the correspondence to the link condition.

Consequently, for these graphs regularity and weak regularity coincide. □

LEMMA 27

1. Admissible graphs are weakly regular.

2. $\rightarrow^1_M$ is confluent over weakly regular graphs.

PROOF Part 1. Since regular graphs are weakly regular, it suffices to show that weakly regular graphs are closed under $\rightarrow^1_{ABCFM}$ reduction. Then by the definition of admissibility, all admissible graphs are weakly regular.

We must verify this for each rewrite. We shall verify each condition holds in the residual graph in turn, assuming all hold in the antecedent graph, and assuming the labels on the ports are exactly as described when the rewrites were introduced in section 5:

Well-formedness This property is conserved under graph rewrites in case (i) all of the internal edges of the replacement graph of the rewrite satisfy the well-formedness criteria for edges, (ii) the edges composed of the external ports facing anything that could have faced the corresponding external port of the rewrite's pattern in a well-formed graph also satisfy the well-formedness criteria for edges, and (iii) no looping match paths are created. By inspection, the criteria on internal edges are satisfied for all graphs, and the condition on external edges and non-looping match paths is easily verified for A1-3, B1-3, C, F, M-ER1a, M-ER2, M-ER4b, M-PR(1-4), M-RV(2-3) and M-GC. The remaining cases are:

1. M-ER1b: Note that this rewrite can create a loop consisting of two match paths, in the case that there is a match path from the pattern port $P-Y$ to a match port of another CER message, and one from the other CER message to one of the new vertex's match port. However the other match port cannot be $P-Y$, else there would have been a loop in the antecedent, forbidden by well-formedness, so neither match path can loop. We must also verify that the new bung vertex in the residual faces a bung vertex: this is ensured by the basis condition holding for the antecedent graph, since $Y \notin \mathsf{basis}(es)$.

2. M-ER3: The basis condition ensures that all of the created bung vertices face bung vertices.

3. M-ER4a: By the message condition for the antecedent, since $P-X, P-Y$ are in the intersection of the bases, there must be a match path from d to a. Consequently the match paths to Y in the CER message with pattern port $P-X$ and to X in the CER message with pattern port $P-Y$ are both conserved and lead to the pattern vertex.

4. M-RV1b: Since the basis of the redex's pattern vertex was empty, by the basis condition all of the bung vertices face bung vertices.

Weak port set condition The condition is quantified over working ports and auxiliary ports of PR messages. It is enough, therefore to verify the condition holds for just the ports occurring in the rewritten portion of the graph, and the ports facing the external ports of the rewritten graph. This can be trivially verified for the rewrites A(1-3),B(1-3), B2, B3, F, M-ER(1-4b), M-PR1, M-PR4, M-RV(1a-3) and M-GC. The remaining cases are:

1. C: The PR messages are created on those edges where the external port of the locked pattern is set. In this case, the facing port must be either a set working port or the reset port of another PR message. In either case the condition holds for the auxiliary port of the PR message, in the former case it holds for the facing working port.

2. M-PR2: The auxiliary port must have faced an unset port in the antecedent graph; now this unset port faces an unset port.

3. M-PR3: Both auxiliary ports must have faced unset ports in the antecedent graph; now these unset ports face each other and we are done.

Weak edge-raising condition This condition is quantified over working vertices and pattern vertices that share a match path, and over links that involve a match path of the pattern vertex. The condition may become untrue if (i) a working vertex changes its port set, (ii) an interaction changes the edge set or terminals of a pattern vertex, or (iii) the set of links in the graph is changed. A1, B(1-3), F, M-PR(1,3-4), M-RV(1b-3) and M-GC leave these unaffected. The remaining cases are:

1. A2: Since in the antecedent graph, the edge occurring in the redex has both ports unset, then there can be no links terminating in the match ports which have the two redex vertices as their matching terminal. After the rewrite, for each pattern vertex for which the edge is internal there is therefore exactly one link. Thus the edge raising condition is true for these two working ports; for other working ports the condition is preserved from th antecedent.

2. A3: Analogous reasoning to the case A2.

3. C: Set working ports that face the external ports of the redex now face chained PR messages; any links that corresponded with these ports now have RV messages as matching terminals, and so are not links in the residual graph. Consequently for these ports the condition holds. The rewritten subgraph consists of plain working vertices, and so the condition holds for these too.

4. M-ER1a: If the matching terminal of both match ports of the ER message are working vertices, then this rewrite simply changes the state of one of the links, from occurring due to an ER message to occurring due to an edge in the edge set of the pattern. If one of the matching terminals is an RV message then no links are affected.

5. M-ER1b, M-ER2: Analogous reasoning to the case M-ER1a.

6. M-ER3: The disjointness of $\vec{U}$ and $\vec{X}$ are ensured by the basis condition, consequently there is a bijective correspondence between the links of the antecedent and residual graph.

7. M-ER4a, M-ER4b: Analogous reasoning to the case M-ER3.

8. M-PR2: The set working port that is reset by the PR message does not face a set working port, so by the condition applying to the antecedent, the port does not apply. Thus the condition holds for the residual unset port.

9. M-RV1a: The edges unset did not contribute links, since they had as a terminal vertex the RV message. Note that links can change type (eg. a link whose match ports are on the same pattern vertex or CER vertex becomes a link spanning two different pattern vertices), we must show that the set of links is unchanged. All edges that contained the port V facing the RV message were not links, so there is a bijective correspondence between the other edges before and after, and so the condition is verified.

Message condition This condition is quantified over the edge raising and RV messages of the graph, and additionally concerns the match path to the pattern port of these vertices. Thus A1, B(1-3), F, M-PR(1-4), M-RV(1a-3) and M-GC are easily seen to leave the condition unchanged.

1. A2: To verify the message condition for the new edge raising condition, we must ensure firstly, that no raised edge has the two working vertices as its terminals: this is so since any such edge would make a link, which would violate the basis condition since the two ports in the antecedent graph were unset. The second condition follows in a similar manner by observing the edge must be a link: since the other vertex the edge is between is in the basis it must be either a working terminal or an RV message (by the basis condition in the antecedent); in the former case it is a link, the latter case is forbidden by the message condition on the RV message.

2. A3: Analogous to the first part of the reasoning for case A2.

3. C: The condition on the RV messages is satisfied by the creation of the PR messages for the external ports; for internal edges of each pattern both terminals will be RV messages, also satisfying the condition.

4. M-ER1a: As before, the links are in a bijective correspondence (the ER message was a link iff the new edge of the pattern is a link, both cannot be links by the message condition for the antecedent graph).

5. M-ER1b: Again, the links of the antecedent graph are in a bijective correspondence with those of the residual, and correspond to the same set of working edges.

6. M-ER2: Analogous to case M-ER1a.

7. M-ER3: The disjointness of $\vec{U}$ and $\vec{X}$ are ensured by the basis condition, consequently the rewritten graph is well-formed, and there is a bijective correspondence between the links of the antecedent and residual graph.

8. M-ER4a: By the message condition for the antecedent, since $P - X, P - Y$ are in the intersection of the bases, there must be a match path from d to a. Consequently the match paths to Y in the CER message with pattern port $P - X$ and to X in the CER message with pattern port $P - Y$ are both conserved and lead to the pattern.

9. M-ER4b The edge raising condition for the new residual is an easy consequence of the for the two antecedent messages and the weak basis condition.

Weak basis condition This condition is quantified over the pattern vertices and CER messages of the graph, and is affected by the changes to pattern vertices, the matching terminals of pattern vertices and CER messages, and the creation of new bridging links. Thus A3, B(1-3), F, M-ER1a, M-ER2, M-PR(1-4), M-RV(1b-3) and M-GC are easily seen to leave the condition unchanged.

1. A1: The condition is immediately verified for all the patterns created by this rewrite.
2. M-ER1b: Well-formedness of the antecedent graph ensures that the matching Y and Z terminals of the ER message are not bung, from which the condition follows.
3. M-ER3: The new basis is $\vec{U} \cup \vec{X} \cup \{Y\}$, and the existence of their terminals is ensured by the basis condition in the antecedent, so the condition immediately follows.
4. M-ER4a, M-ER4b: Analogous to M-ER3.
5. M-RV1a: The second clause of the basis condition follows from the disjointness of all the created vertices. The matching terminals of the ports $\vec{W}$ are RV messages or working vertices; these become the sole non-bung vertex of their respective pattern in the rewrite, which has empty edge set. The terminals of the ports $\vec{V_i}$ are also RV messages or working vertices; these are the basis of the respective messages which also have bungs on all other ports, thus all the created patterns satisfy the condition.

Link condition This condition is quantified over links of the graph, and is affected by changes to the edge raising messages, and to the edge sets and matching terminals of pattern vertices and CER messages. Thus B(1-3), F, M-PR(1-4), M-RV(1a-3) and M-GC are easily seen to leave the condition unchanged.

1. A1: Since the created pattern vertices have empty edge sets, the pattern condition is vacuously satisfied for them.
2. A2: We saw when showing the edge raising condition that the new edge does not duplicate a link, so the uniqueness of links is maintained. We see immediately that the clause concerning port sets is satisfied for these new edges.
3. A3: Similar reasoning to case A2.
4. C: All links whose terminals are vertices in the C redex are destroyed by the rewrite, from which the condition immediately follows.
5. M-ER1a: Two subcases: if the raised edge in the redex did not induce a link, then the condition follows immediately. Otherwise, then we need that the matching terminal of the Y part of the link corresponds to the same working port before and after the rewrite. This follows by well-formedness (specifically, there is a match path from the Y match port of the pattern vertex to the P-Y vertex of the ER message), and ensures uniqueness of the link and the condition regarding port sets are both inherited in the residual graph.
6. M-ER1b: There is an easily seen bijection between the links of the left and right hand sides of the rewrite; all links must conserve the same matching terminals.
7. M-ER2: Analogous to M-ER1a.
8. M-ER3, M-ER4a, M-ER4d: Analogous to M-ER1b.

Part 2. We must verify that weakly regular graphs rewrite to well-formed graphs under $\rightarrow_M^1$, and then inspect the critical pairs. By inspection of the rules, we observe that the critical pairs fall into three classes:

1. Either an ER message joined to two pattern vertices, or on both edges to the same pattern vertex: the critical pair then contains two overlapping M-ER1(ab) redexes. In the first case (where there are two pattern vertices), by the weak basis condition we have that both rewrites yield an M-ER1b redex, whilst in the second both yield an M-ER1a redex. In each case we see that the critical pair is resolvable.

2. Second, two PR messages joined to a working vertex or locked pattern: in the former case we may apply the M-PR1 rewrites in either order and obtain the same residual, and in the second case, exploding the pattern obtains the former case.

3. Third, messages facing two match ports of a pattern. The tricky cases are the ones involving an edge raising message and a remove vertex message: these critical pairs are only resolvable in case the RV message cannot reset the edge set by the edge raising message.

   (a) (M-ER1a,M-RV1): Let the pattern ports of the ER message be $P - Z$ and $P - Y$, and WLOG assume the pattern faces the $P - Z$ terminal. The message condition ensures that the Y match port of the pattern vertex does not face the RV message, and so we see that the two rewrite rules may be applied in either order to yield a common residual.

   (b) (M-ER1b,M-RV1): Let $P - Z$ and $P - Y$ be as above; again the message condition forbids the RV message being the matching Y terminal; so again the two rewrite rules may be applied in either order to yield a common residual.

   (c) (M-ER2,M-RV1) There can be no overlap, so this case is easily resolved.

   (d) (M-ER3,M-RV1): Let $P - Z$ be the pattern port of the CER message, whose edge set is es. Then the intersection of the bases of es and es' must be at most the singleton consisting of the pattern port vertex Z by the message condition, so the RV message cannot remove any edges from es.

   (e) Critical pairs involving two edge raising vertices are resolvable just in case there are no overlaps between the edge sets (ie. the edge set does not include any edge carried by an edge raising message, and the edge raising messages carry different edges). This situation is forbidden by the weak message condition .

   (f) (M-RV1,M-RV1), (M-RV2,M-RV2), (M-RV3,M-RV3): The critical pairs involving two vertex remove messages are all easily resolved.

4. Lastly we have cases involving an overlap between a CER merging rewrite and either another CER merging rewrite or the M-ER3 rewrite. There are three cases:

   (a) (M-ER4a, M-ER4a): we must show that in a weakly regular graph, this must consist of two vertices joined in a loop. Suppose not, ie. there are three vertices, the middle of which is part of both M-ER4a redexes. WLOG this case has the

first rewrite along a Y — $P - Y$ match–pattern edge (between a CER vertex $M(es)$ whose pattern port is $P - X$ and one $N(es')$), and the second redex along a Z — $P - Z$ match–pattern edge (where the Z match port belongs to the same message with the $P - Y$ pattern port). The condition on the M-ER4a ensures that $X \neq Y$ and $X, Y \in \mathsf{basis}(es')$, but the message condition allows both X and Y to be in the basis in case the two vertices are involved in a loop. By reductio ad absurdum, we have the vertices must form a loop. (similar reasoning forbids the (M-ER3, M-ER4a) critical pair).

So there are two vertices joined in a loop. WLOG let the pattern ports be $P - X$ and $P - Y$ (the rewrite rule forbids $X = Y$, we have P-Y of one connected to Y of the other, whose port $P - X$ is connected to the X port of the first). Applying either rewrite obtains a common form.

(b) (M-ER4b, M-ER4b): The message condition ensures that the three basis sets are distinct, so we may apply the rewrite rules in either order to obtain a common form.

(c) (M-ER3, M-ER4b): The message condition ensures that the three basis sets are distinct, so we may apply either the M-ER4b rule followed by the M-ER3 rule, or two M-ER3 rules to obtain a common form.

Since all the critical pairs are solvable and $\rightarrow_M^1$ is strongly normalising, $\rightarrow_M^1$ is confluent. □

THEOREM 28 Each admissible graph has a unique, regular, M-normal form.

PROOF Firstly we show that weakly regular M-normal forms can't have messages. This follows from (i) the weak port set condition ensures that all port reset messages occur as part of a redex, (ii) all other message vertices must occur on match paths, and (ii) all edges occurring on maximal match paths must either be between the match port of a pattern vertex and the pattern port of a working vertex (in which case this match path contains no messages), or they must contain an edge passing message. By a simple case analysis, the topmost such edge must occur in a redex. Since $\rightarrow_M^1$ is strongly normalising and confluent for weakly regular graphs, there is a unique, message-free M-normal form for each admissible graph.

Then by the equivalence of regularity and weak regularity, we have that this normal form is regular. □

COROLLARY 29 Let F, R and $R^{\rightarrow}$ be as defined at the beginning of the section. Then $\langle F, R, R^{\rightarrow} \rangle : G \rightarrow H$ is an translation.

PROOF We need to show that:

1. For all graphs $g \in \mathcal{G}(G)$, $F(g)$ is regular: $F(g)$ has no messages, no pattern vertices and no working vertices with port set attributes. Since $F(g)$ is well-formed, it is therefore vacuously regular.

2. R is total on $\mathcal{G}(H)$: this follows from well-formedness.

3. For each g, $R(F(g)) = g$: both functions are identities on graphs of Σ_G.

4. For any $g_0 \rightarrow^1_{ABCM} g_1$, $R(g_0) = R(g_1)$: by inspection, none of these reductions affect working graph structure for well-formed graphs, with the exception of the rules C1, M-PR1 and M-PR2, which only affect the set of PR messages on a chain of working edges, all of which maps to a working edge under R.

5. For each $g_0 \rightarrow^1_r g_1$, where $r \in \mathcal{R}_F$, $R(g_0) \rightarrow^1 R(g_1)$: this is immediate.

□

PROPOSITION 30

1. If a regular graph h is an A-normal form which contains no locked or 'fire pattern' vertices, then there is a bijective correspondence between the set of partial homomorphisms of the G-patterns into $R(h)$ and the set of complete pattern vertices in h.

2. $\rightarrow^1_{ABCM}$ is strongly normalising.

3. If a regular graph h is an ABM-normal form which contains no 'fire pattern' vertices, then for each group of overlapping patterns of G into $R(h)$, at least one corresponds to a fully locked pattern vertex of h.

4. $\rightarrow^1_{ACFM}$ is strongly normalising and confluent.

PROOF Part 1. We prove this by constructing mutually inverse mappings. Each complete pattern vertex $P0$ can easily be seen to specify a homomorphism from the associated G-pattern P into $R(h)$: in the case where P has internal ports each internal port of P corresponds to an internal port of a working vertex, and in the case where P has none the vertex faces the match port of the pattern vertex. By the edge raising condition edges are invariant under this map; we obtain our homomorphism by composition with $R(-)$.

The reverse map is constructed by finding a candidate pattern vertex associated with any partial homomorphism and showing it to be complete. Let P be the pattern, and choose some vertex V in P. Then the homomorphism associates a vertex A_0 of $R(h)$ with V, this is the image of a vertex A of h under R. The associated pattern vertex with this homomorphism is whatever pattern vertex $P0$ faces A on its $P - V$ pattern port. Suppose, for a contradiction, that $P0$ is not complete. Then there must be a vertex U of P and internal edge of P $e = \{\langle U, p_0\rangle, \langle V, p_1\rangle\}$ (U, V possibly the same vertex), where the match port U faces a working vertex and e is not in the edge set of $P0$. By the edge raising condition we have that $\langle U, p_0\rangle$ is unset, and by the port set condition their must be an A2 or A3 redex, contradicting A-normality of the graph.

It is easy to see that the two functions just specified are inverses.

Part 2. We show that there is a well-founded measure[7] on graphs that is decreasing under $\rightarrow^1_{ABCM}$.

[7]Recall that an ordered set is well-founded when it admits no infinite decreasing sequences. A well-founded measure is a function whose codomain has a well-founded order relation.

The measure is calculated as follows: let N_1 be the number of partial homomorphisms of the patterns of G into the graph, N_2 be the maximal number of ports in any pattern of G (say this is one if there are no patterns of G), N_3 be the total number of ports of the graph and N_4 be the number of patterns of G. n_1 be the number of fire pattern vertices in the graph, n_2 be the number of dead patterns, n_3 be the number of PR and RV messages in the graph, n_4 be the number of locked working vertices times the priority of the pattern the vertex is locked to (ie. the number of patterns less than the pattern under $<_{\mathsf{pat}}$, n_5 be the number of set ports of working vertices in the graph, n_6 be the number of set edges of pattern vertices in the graph, n_7 is the number of pattern nodes and n_8 be twice the number of ER messages plus the number of CER and LR messages in the graph. Then each of $N_1 - n_1$, $N_1 - n_2$, $N_3 \times N_4 - n_4$, $N_3 - n_5$, $N_1 \times N_2 - n_6$ and $N_1 \times N_2 - n_7$ is a non-negative integer, so the lexical order $\langle N_1 - n_1, N_1 - n_2, n_3, N_3 \times N_4 - n_4, N_3 - n_5, N_1 \times N_2 - n_6, N_1 \times N_2 - n_7, n_8 \rangle$ is well-founded (with leftmost number being most significant). By inspection we see that this measure on graphs is decreasing under $\rightarrow^1_{ABCM}$.

Part 3. We observe that an ABM-normal form may only fail to induce the bijection between pattern vertices and G-pattern partial homomorphism described in part 1 in the case that edges are not raised due to one of the involved working vertices being locked to a pattern vertex associated with an overlapping pattern; this pattern must be complete. For a given partial homomorphisms, let us examine the set of overlapping patterns (we say two pattern vertices are 'touching' if they are both face –either by a match–pattern edge or by being locked to– the same working vertex; the set of overlapping pattern vertices are the set of complete patterns that are belong to the same equivalence class induced by the transitive closure of this touching relation): there must be a maximal such pattern under the order relation on partially locked pattern vertices given for rule B2. Since the graph ABM-normal and the pattern vertex is maximal, it must be locked to all of its working vertices (else there would be a B-redex); none of the working vertices can be identical and so the locked pattern must correspond to a full homomorphism by the first function described in part 1.

Part 4. To see that ACFM is strongly normalising, we construct a further measure that is decreasing under $\rightarrow^1_{CF}$ and non-increasing under $\rightarrow^1_{AM}$: let n_9 be the number of fully-locked pattern vertices: then $\langle N_1 - n_8, N_1 - n_9 \rangle$ provides this measure (where N_1 is as given in the original graph; the set of partial homomorphisms of patterns is likely to change under $\rightarrow^1_F$. The lexical order of the two measures (with the one just given having priority) is therefore well-founded and decreasing under $\rightarrow^1_{ACFM}$.

We show that $\rightarrow^1_{ACFM}$ is confluent by observing that there are no critical pairs involving C or F, that $\rightarrow^1_M$ is confluent and jointly confluent with $\rightarrow^1_{ACF}$ by 27(2), and by inspection of the critical pairs of $\rightarrow^1_A$ (these are all easily resolvable). □

The following is immediate.

COROLLARY 31 $\langle F, R, R^{\rightarrow} \rangle : G \rightarrow H$ is livelock and deadlock free.

DEFINITION 32 Define $\bar{F}(g)$ to be the AM-normal form of $F(g)$.

PROPOSITION 33
There is a map $\bar{F}^{\rightarrow}$ such that $\langle \bar{F}, \bar{F}^{\rightarrow}, R, R^{\rightarrow} \rangle : G \rightarrow H$ is a functorial translation.

PROOF By construction: if $g \rightarrow_r g'$, then there is a complete pattern vertex P0 of $\bar{F}(g)$ which corresponds to the pattern of r. Define $\bar{F}^{\rightarrow}(r)$ to be the catenation of:

1. A sequence of class B rewrites fully locking the pattern vertex P0, which only contain B1 interactions between it and a working vertex.

2. A sequence of ACFM rewrites that compute the ACFM-normal form of this graph (which we see must contain exactly one C and where we insist the F rewrite is associated with r).

Call the resulting graph h'. We need to show that $\bar{F}(g') = h'$ and then we are done. Observe that distinct ACFM normal forms are distinguished by their locked pattern vertices and dead patterns: since both $\bar{F}(g')$ and h' have none we are done.

It remains to show that, firstly $\langle \bar{F}, R, R^{\rightarrow} \rangle : G \rightarrow H$ is an translation, and that $\bar{F}^{\rightarrow}$ extends the translation to a functorial translation. The first follows from the observation that $\forall g \in \mathcal{G}(\Sigma_G).R(F(g)) = R(\bar{F}(g))$, and the second is a consequence of each $\bar{F}^{\rightarrow}(r)$ containing just the one F rewrite associated with r. □

COROLLARY 34 $\langle F, R, R^{\rightarrow} \rangle : G \rightarrow H$ is a faithful translation.

7 Conclusions

This paper solves a problem that is relatively simply stated: the provision of correct and complete (ie. concurrency-conserving) compilation schemes from general static-port graph grammars into basic static-port graph grammars. Such a scheme, if sufficiently efficient, is of use in a compiler whose architecture and goals are as described in the introduction and section two.

The aims of the discussion here in the conclusion are firstly, to review the proposed scheme as a prototype design of an implementation, which principally revolves around a discussion of the schemes efficiency, and second, to discuss alternative schemes that might also achieve the same result. We end with a brief description of some related unsolved problems.

We shall begin with an informal analysis giving an upper bound on time complexity; a crude worst-case performance estimate. Given an SPGG G and a source working graph g, what is the worst possible number of rewrites that could be associated with a reduction chain from g of length n, disregarding the costs of garbage collection? The rewrites in group M are, with the exception of the M-GC rewrites[8], bounded by the number of non-bung messages in the working graph: this bound is dependent upon the rules that generate the message passing rewrites, so we shall account for the M rewrites as attendant upon the A,B,C and F class rewrites. It is easy to see that the total cost associated with the C and F class rewrites will be dominated by the number of A and B class rewrites performed (the

[8]We do not count the cost of GC rewrites towards our performance estimate, as in practice garbage collection can be performed efficiently without these rewrites.

C rewrites generate a number of M rewrites bounded by twice the number of edges in the pattern associated with the locked pattern vertex).

The A class rewrites are initially expensive: A1 rewrites are performed for every vertex in the graph, and A2 and A3 rewrites are performed for every working edge of the graph. Suppose there are k rewrite rules in the source grammar, p is maximum number of edges (internal and external) of any of the pattern graphs of the rewrite rules, r is likewise the maximum number of edges in the replacement graphs, and e is the number of internal edges of the source working graph. Then $2kpe$ provides an upper bound to the number of rewrites incurred through A2/A3 reductions and their attendant message passing rewrites (kp provides an upper bound to the number of edge raising vertices created for each A2 rewrite). However, the B and C rewrites generally conserve the pattern structure so created: in the wake of each F rewrite the complexity associated with the new class A rewrites is bounded by $2kpr + v_n$, where v_n is the number of new vertices. Since the number of vertices of a graph is bounded by twice the number of edges, the total cost of the A class rewrites is bounded by $2(kp + 1)(e + rn)$.

The class B rewrites are also potentially expensive, though not so much as the class A rewrites. Potentially, due to A2 rewrites, a vertex can be locked by all k different patterns before it finds itself in a fully locked pattern or a dead vertex: in the latter case the process can start all over again, also we can have interactions between patterns of equal precedence... The total number of B rewrites is bounded by $2(k^2 + kp)(e + rn)$.

Overall the total cost of the rewrite complexity is polynomial in both the size of the source grammar and the size of the working graph, so the translation described is feasible, though not particularly efficient in the form presented. Several improvements to the performance of the system are available, however, amongst these are the following:

1. (Partial evaluation) We may reduce the number of reduction steps by selectively expanding rewrites whose patterns appear in the working graph or in the replacement graphs of the grammars rewrite rules, as described in section two. Simply expanding all A1 redexes in the working graph and in the replacement graph of 'C' rewrites eliminates the initial overhead of generating patterns.

2. (Create pattern vertices on demand) Alternatively we may attempt to avoid having unnecessary pattern vertices present. The A1 redexes may be performed though there are no relevant edges bordering the term. By allowing A2 rewrites to be performed between plain working vertices, and between plain working vertices and working vertices with match ports (and likewise with A3 rewrites), we can eliminate the A1 rewrite. Combined with the simple propagation of rewrites described above, this technique will automatically collapse the rewrite chain in the target calculus associated with binary interactions and loop absorptions of a single loop from the source calculus into a single step; thus the overhead of pattern structure associated with the pattern matching vertices will be incurred only where it is needed.

3. (Share subpatterns) The resource complexity of both A2/A3 rewrites and of the locking conflicts carried out by B2 rewrites can be minimised by allowing the patterns to

be structured hierarchically, with pattern vertices having match ports both for working vertices and for common subpatterns. It is quite easy to redesign the A and C class rewrites to handle subpatterns; quite a few subtleties plague the design of B class rewrites, however. The savings offered by this optimisation is clearly related to the number of repeated occurrences of a subpattern in the patterns of the rewrite system; this saving can be high if there are large, similar patterns with a few minor variations, or large patterns built out of the repeated composition of several small subpatterns.

The above two optimisations will go a long way towards making the performance of the algorithm acceptable. Let us have a look at possible alternative approaches to coding graphs:

1. (Ad hoc translation schemes) Here we do not attempt to solve the coding of general grammars in their full generality, but only to the generality required for the compilation of a specific programming language. This approach should achieve much better performance than the general purpose approach described here, since we can make use of type information and what we know about the structure of the restricted class of graphs to introduce special techniques. For example, in the common case that the graphs possess a DAG structure, it becomes practicable to fold updates over the whole graph by propagating changes from root to leaves.

2. (Internal metadata) We briefly discussed in section four the alternative, simpler, possibility of representing all metadata in extended attributes within the working vertices. The possibility was not developed further, principally on circumstantial grounds: my experience with the complexity of designing mechanisms for propagating updates to information: these tend to be fraught with race conditions. Nonetheless, difficulty of design does not mean impossibility, and it may be the case that the obstacles to such a solution can be efficiently overcome.

3. ('Impure' solutions) The design of compilation schemes based upon the internal representation of metadata becomes easier if the run time environment allows certain programming extensions. Two particularly are of interest: we may allow first-class access to names for vertex identities, so that vertices can communicate to their neighbours information about vertex identifiers (a caveat: this extension is quite useless if vertex identities change through any rewrite, and correctness is clearly sensitive to which rewrites conserve vertex identity). A second method is to allow access to timestamps, making it easy to decide precedence between conflicting sources of information.

   Techniques making use of impure primitives, like the ad hoc methods, do not solve the problem specified problem, but they may be useful. Another drawback is that impure primitives (that is, ones likes those above which involve some form of non-local state) make it harder to reason about graphs for the purposes of compiler optimisation, a situation comparable to that in the compilation of functional programming languages.

Lastly, I would like to bring attention to a few places the results of this paper may be extended, by describing some problems that are so far unsolved:

1. While the account of faithful translations is sufficient for the purposes of this paper, it is not adequate for reasoning about the correctness of concurrency-conserving optimisation steps, since it only allows one rewrite in the source calculus to be associated with chains of at least rewrite in the target calculus; there is no way for two rewrites in the source calculus to be reduced to just one in the target. There are several technical difficulties that face the precise definition of this broader notion of faithful translation.

2. We have no idea of which PSGGs can be reduced to either linear graph grammars (ie. without loop absorption rules) or interaction nets. Particularly the latter is an interesting and difficult problem.

3. It would be useful to have general schemes for reducing general hyperedge replacement grammars to basic PSGGs. Here it my be worth recording a conjecture: all hyperedge replacement grammars all of whose rules have connected patterns can so be compiled. Since graphs of hyperedge replacement grammars are not generally PSGGs, it is necessary to provide a nontrivial embedding scheme for such a grammar.

References

[ASU85] A. V. Aho, R. Sethi and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[AG97] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, 1997.

[Baw86] Alan Bawden. Connection Graphs. In *Proc. Conf. on LISP and Functional Programming*, pages 258–265. ACM Press, 1986.

[Baw93] Alan Bawden. *Implementing Distributed Systems Using Linear Naming*.. PhD thesis, MIT, 1993.

[BM] A. Bawden and H. Mairson. Linear naming and computation: experimental software for optimizing communication protocols. Manuscript. Available online `http://achilles.bu.edu/linear`.

[CR91] W. Clinger and J. Rees, editors. *Revised^4 Report on the Algorithmic Language Scheme*. Technical report, MIT AI laboratory, MIT AI Memo 848b, 1991.

[GAL92a] G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proc. 19th Annual Symposium on Principles of Programming Languages*, pages 15–26. ACM Press, 1992.

[GAL92b] G. Gonthier, M. Abadi, and J.-J. Lévy. Linear logic without boxes. In *7th Annual Symposium on Principles of Programming Languages*, pages 15–26. IEEE Press, 1992.

[Hil85] W. D. Hillis. *The Connection Machine*. MIT Press, Cambridge MA, 1985.

[Kel95] R. A. Kelsey. A correspondence between Continuation Passing Style and Static Single Assignment form. ACM SIGPLAN Notices 30(3), 1995.

[Laf90] Yves Lafont. Interaction nets. In *Proc. Annual 17th Symp. Principles of Programming Languages*, pages 95–108. ACM Press, 1990.

[Lam90] John Lamping. *An algorithm for optimal lambda calculus reduction.* In *Proc. Annual 17th Symp. Principles of Programming Languages*, pages 16–30. ACM Press, 1990.

[Lev80] Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. In *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, edited by J. P. Seldin and J. R. Hindley, editors), pages 159–191. Academic Press, 1980.

[Mac94] Ian Mackie. *The Geometry of Optimal Implementation*. PhD thesis, Imperial College, London, 1994.

[Mat00] Satoshi Matsuoka. Additive interaction nets. Unpublished manuscript.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[Roz97] G. Rozenberg (editor). *Volume I:Foundations* of the *Handbook of Graph Grammars and Computing by Graph Transformations*. World Scientific, 1997.

[Rud98] M. Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation* (TAGT '98), 1998.

[Sch97] A. Schürr. Programmed graph replacement systems. In [Roz97], 1997.

[Ste76] Guy L. Steele. Lambda: the ultimate declarative. AI memo 379, MIT AI Laboratory, 1976.