Electronic Communications of the EASST
Volume 71 (2015)

Graph Computation Models
Selected Revised Papers from GCM 2014

More on Graph Rewriting With Contextual Refinement

Berthold Hoffmann

20 pages

Guest Editors: Rachid Echahed, Annegret Habel, Mohamed Mosbah
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

More on Graph Rewriting With Contextual Refinement

Berthold Hoffmann

Fachbereich Mathematik und Informatik, Universität Bremen, Germany

Abstract: In GRGEN, a graph rewrite generator tool, rules have the outstanding feature that variables in their pattern and replacement graphs may be refined with meta-rules based on contextual hyperedge replacement grammars. A refined rule may delete, copy, and transform subgraphs of unbounded size and of variable shape. In this paper, we show that rules with contextual refinement can be transformed to standard graph rewrite rules that perform the refinement incrementally, and are applied according to a strategy called residual rewriting. With this transformation, it is possible to state precisely whether refinements can be determined in finitely many steps or not, and whether refinements are unique for every form of refined pattern or not.

Keywords: graph rewriting – rule rewriting – contextual hyperedge replacement

1 Introduction

Everywhere in computer science and beyond, one finds systems with a structure represented by graph-like diagrams, whose behavior is described by incremental transformation. Model-driven software engineering is a prominent example for an area where this way of system description is very popular. Graph rewriting, a branch of theoretical computer science that emerged in the seventies of the last century [EPS73], is a formalism of choice for specifying such systems in an abstract way [MEDJ05]. Graph rewriting has a well developed theory [EEPT06] that gives a precise meaning to such specifications. It also allows to study fundamental properties, such as termination and confluence. Over the last decades, various tools have been developed that generate (prototype) implementations for graph rewriting specifications. Some of them do also support the analysis of specifications: AGG [ERT99] allows to determine confluence of a set of rules by the analysis of finitely many critical pairs [Plu93], and GROOVE [Ren04] allows to explore the state space of specifications.

This work relates to GRGEN, an efficient *graph rewrite generator* [BGJ06] developed at Karlsruhe Institute of Technology. Later, Edgar Jakumeit has extended the rules of this tool substantially, by introducing recursive refinement for sub-rules and application conditions [HJG08]. A single refined rule can match, delete, replicate, and transform subgraphs of unbounded size and variable shape. These rules have motivated the research presented in this paper. Because, the standard theory [EEPT06] does not cover recursive refinement, so that such rules cannot be analyzed for properties like termination and confluence, and tool support concerning these questions cannot be provided.

Our ultimate goal is to lift results concerning confluence to rules with recursive refinement. So we formalize refinement by combining concepts of the existing theory, on two levels: We define a GRGEN rule to be a schema – a plain rule containing variables. On the meta-level, a schema is refined by replacing variables by sub-rules, using meta-rules based on contextual

hyperedge replacement [DHM12]. Refined rules then perform the rewriting on the object level. This mechanism is simple enough for formal investigation. For instance, properties of refined rules can be studied by using induction over the meta-rules. Earlier work [Hof13] has already laid the fundaments for modeling refinement. Here we study conditions under which the refinement behaves well. We translate these rules into standard rules that perform the refinement in an incremental fashion, using a specific strategy, called residual rewriting, strategy, and show the correctness of this translation.

The examples in this paper arise in the area of model-driven software engineering. *Refactoring* shall improve the structure of object-oriented software without changing its behavior. Graphs are a straight-forward representation for the syntax and semantic relationships of object-oriented programs (and also of models). Many of the basic refactoring operations proposed by Fowler [Fow99] do require to match, delete, copy, or restructure program fragments of unbounded size and variable shape. Several plain rules are needed to specify such an operation, and they have to be controlled in a rather delicate way in order to perform it correctly. In contrast, we shall see that a single rule schema with appropriate meta-rules suffices to specify it, in a completely declarative way.

The paper is organized as follows. The next section defines graphs, plain rules for graph rewriting, and contextual rules for deriving languages of graphs. In Sect. 3 we define schemata, meta-rules, and the refinement of schemata by applying meta-rules to them, and state under which conditions refinements can be determined in finitely many steps, and the replacements of refined rules are uniquely determined by their patterns. In Sect. 4, we translate schemata and meta-rules to standard graph rewrite rules,and show that the translation is correct. We conclude by indicating future work, in Sect. 5. The appendix recalls some facts about graph rewriting.

2 Graphs, Rewriting, and Contextual Grammars

We define labeled graphs wherein edges may not just connect two nodes – a source to a target – but any number of nodes. Such graphs are known as hypergraphs in the literature [DHK97].

Definition 1 (Graph) Let $\Sigma = (\dot{\Sigma}, \bar{\Sigma})$ be a pair of finite sets containing *symbols*.

A *graph* $G = (\dot{G}, \bar{G}, att, \ell)$ consists of two disjoint finite sets $\dot{G}$ of *nodes* and $\bar{G}$ of *edges*, a function $att\colon \bar{G} \to \dot{G}^*$ that *attaches* sequences of nodes to edges,[1] and of a pair $\ell = (\dot{\ell}, \bar{\ell})$ of *labeling functions* $\dot{\ell}\colon \dot{G} \to \dot{\Sigma}$ for nodes and $\bar{\ell}\colon \bar{G} \to \bar{\Sigma}$ for edges. We will often refer to the attachment and labeling functions of a graph G by att_G and ℓ_G, respectively.

A (*graph*) *morphism* $m\colon G \to H$ is a pair $m = (\dot{m}, \bar{m})$ of functions $\dot{m}\colon \dot{G} \to \dot{H}$ and $\bar{m}\colon \bar{G} \to \bar{H}$ that preserve attachments and labels: $att_H \circ \bar{m} = \dot{m}^* \circ att_G$, $\dot{\ell}_H = \dot{\ell}_G \circ \dot{m}$, and $\bar{\ell}_H = \bar{\ell}_G \circ \bar{m}$.[2] The morphism m is *injective*, *surjective*, and *bijective* if its component functions have the respective property. If $\dot{G} \subseteq \dot{H}$, $\bar{G} \subseteq \bar{H}$, m is injective, and maps nodes and edges of G onto themselves, this defines the *inclusion* of G as a subgraph in H, written $G \hookrightarrow H$. If m is bijective, we call G and H *isomorphic*, and write $G \cong H$.

[1] A^* denotes *finite sequences* over a set A; the empty sequence is denoted by ε.

[2] For a function $f\colon A \to B$, its extension $f^*\colon A^* \to B^*$ to sequences A^* is defined by $f^*(a_1 \ldots a_n) = f(a_1) \ldots f(a_n)$, for all $a_i \in A$, $1 \leqslant i \leqslant n$, $n \geqslant 0$; $f \circ g$ denotes the composition of functions or morphisms f and g.

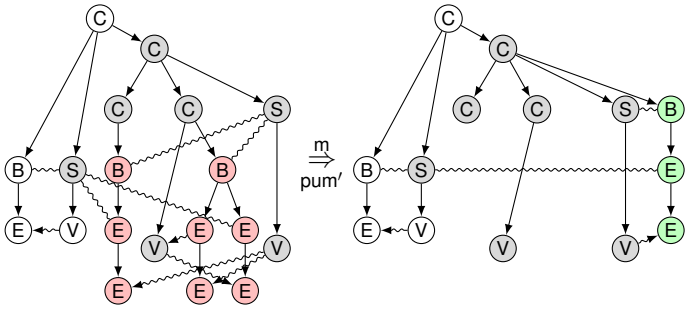

Figure 1: Two program graphs

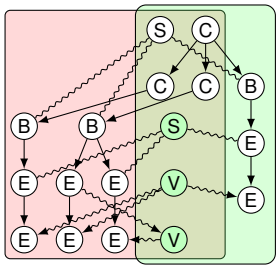

Figure 2: A refactoring rule

Example 1 (Program Graphs) Figure 1 shows two graphs G and H representing object-oriented programs. Circles represent nodes, and have their labels inscribed. In these particular graphs, edges are always attached to exactly two nodes, and are drawn as straight or wave-like arrows from their source node to their target node. (The filling of nodes, and the colors of edges will be explained in Example 2.)

Program graphs have been proposed in [VJ03] for representing key concepts of object-oriented programs in a language-independent way. In the simplified version that is used here, nodes labeled with C, V, E, S, and B represent program entities: classes, variables, expressions, signatures and bodies of methods, respectively. Straight arrows represent the syntactical composition of programs, whereas wave-like arrows relate the use of entities to their declaration in the context.

For rewriting graphs, we use the standard definition [EEPT06], but insist on injective matching of rules; it is shown in [HMP01] that this is no restriction. We choose an alternative representation of rules discussed in [EHP09] so that the rewriting of rules in Sect. 3 can be easier defined, see also in Appendix A.

Definition 2 (Graph Rewriting) A *graph rewrite rule* (*rule* for short) $r = (P \hookrightarrow B \hookleftarrow R)$ consists of graph inclusions, of a *pattern* P and a *replacement* R in a common *body* B. The intersection $P \cap R$ of pattern and replacement graph is called the *interface* of r. A rule is *concise* if the inclusions are jointly surjective. By default, we refer to the components of a rule r by P_r, B_r, and R_r.

The rule r *rewrites* a *source graph* G into a *target graph* H if there is an injective morphism $B \to U$ to a *united graph* U so that the squares in the following diagram are pushouts:

$$\begin{array}{lccc} r\colon & P & \hookrightarrow & B & \hookleftarrow & R \\ & {\scriptstyle m}\downarrow & & \downarrow & & \downarrow{\scriptstyle \tilde{m}} \\ & G & \longrightarrow & U & \longleftarrow & H \end{array}$$

The diagram exists if the morphism $m\colon P \to G$ is injective, and satisfies the following *gluing condition*: Every edge of G that is attached to a node in $m(P \setminus R)$ is in $m(P)$. Then m is a *match* of r in G, and H can be constructed by (i) uniting G disjointly with a fresh copy of the body B, and gluing its pattern subgraph P to its match $m(P)$ in G, giving U, and (ii) removing the

nodes and edges of $m(P \setminus R)$ from U, yielding H with an *embedding* morphism $\tilde{m}: R \to H$.[3] The construction is unique up to isomorphism, and yields a *rewrite step*, which is denoted as $G \Rightarrow_r^m H$. Note that the construction can be done so that there are inclusions $G \hookrightarrow U \hookleftarrow H$; we will assume this wlog. in the rest of this paper.

Example 2 (A Refactoring Rule) Figure 2 shows a rule pum′. Rounded shaded boxes enclose its pattern and replacement, where the pattern is the box extending farther to the left. Together they designate the body. (Rule pum′ is concise.) We use the convention that an edge belongs only to those boxes that contain it entirely; so the "waves" connecting the top-most S-node to nodes in the pattern belong only to the pattern, but not to the replacement of pum′.

The pattern of pum′ specifies a class with two subclasses that contain method implementations for the same signature. The replacement specifies that one of these methods shall be moved to the superclass, and the other one shall be deleted. In other words, pum′ *pulls up methods*, provided that both bodies are semantically equivalent. (This property cannot be checked automatically, but has to be verified by the user before applying this refactoring operation.)

The graphs in Figure 1 constitute a rewrite step $\mathsf{G} \Rightarrow^{\mathsf{m}}_{\mathsf{pum}'} \mathsf{H}$. The shaded nodes in the source graph G distinguish the match m of pum′, and the shaded nodes in the target graph H distinguish the embedding $\tilde{m}$ of its replacement. (The red nodes in G are removed, and the green nodes in H are inserted, with their incident edges, respectively.)

Rule pum′ only applies if the class has exactly two subclasses, and if the method bodies have the particular shape specified in the pattern. The general *Pull-up Method* refactoring of Fowler [Fow99] works for classes with any positive number of subclasses, and for method bodies of varying shape and size. This cannot be specified with a single plain rule, which only has a pattern graph of fixed shape and size. The general refactoring will be specified by a single rule schema (with a set of meta-rules) in Example 5 further below.

We introduce further notions, for rewriting graphs with sets of rules. Let $\mathscr{R}$ be a set of graph rewrite rules. We write $G \Rightarrow_{\mathscr{R}} H$ if $G \Rightarrow_r^m H$ for some match m of a rule $r \in \mathscr{R}$ in G, and denote the transitive-reflexive closure of this relation by $\Rightarrow^*_{\mathscr{R}}$. A graph G is *in normal form* wrt. $\mathscr{R}$ if there is no graph H so that $G \Rightarrow_{\mathscr{R}} H$. A set $\mathscr{R}$ of graph rewrite rules *reduces* a graph G to some graph H, written $G \Rightarrow^!_{\mathscr{R}} H$, if $G \Rightarrow^*_{\mathscr{R}} H$ so that H is in normal form. $\mathscr{R}$ (and $\Rightarrow_{\mathscr{R}}$) is *terminating* if it does not admit an infinite rewrite sequence $G_o \Rightarrow_{\mathscr{R}} G_1 \Rightarrow_{\mathscr{R}} \ldots$, and *confluent* if for every diverging rewrite sequence $H_1 \overset{*}{\underset{\mathscr{R}}{\Leftarrow}} G \Rightarrow^*_{\mathscr{R}} H_2$, there exists a graph K with a joining rewrite sequence $H_1 \Rightarrow^*_{\mathscr{R}} K \overset{*}{\underset{\mathscr{R}}{\Leftarrow}} H_2$. Graph rewrite rules $\mathscr{R}$ can be used to compute a partial nondeterministic function $f_{\mathscr{R}}$ from graphs to sets of their normal forms, i.e., $f_{\mathscr{R}}(G) = \{H \mid G \Rightarrow^!_{\mathscr{R}} H\}$. The function $f_{\mathscr{R}}$ is total if $\mathscr{R}$ is terminating, and deterministic if $\mathscr{R}$ is confluent.

A set of graph rewrite rules, together with a distinguished start graph, forms a grammar, which can be used to derive a set of graphs from that start graph. Such sets are called *languages*, as for string grammars. Graph grammars with unrestricted rules have been shown to generate the recursively enumerable languages[Ues78]. So there can be no general algorithms recognizing whether a graph belongs to the language of such a grammar. Until recently, the study of restricted grammars with recognizable languages has focused on the context-free case, where the

[3] Even if r is not concise, the nodes and edges of B that are not in the subgraph $(P \cup R)$ are not relevant for the construction as they are removed immediately after adding them to the union.

pattern of a rule is a syntactic variable (or nonterminal). Two different ways have been studied to specify how the neighbor nodes of a variable are connected to the replacement graph. In node replacement [ER97], this is done by embedding rules for neighbor nodes that depend on the labels of the neighbors and of the connecting edges. In hyperedge replacement [DHK97], the variable is a hyperedge with a fixed number of attached nodes that may be glued to the nodes in the replacement. Unfortunately, the languages derivable with these grammars are restricted in the way their graphs may be connected: neither a language as simple as that of all graphs, nor the language of program graphs introduced in Example 1 can be derived with a context-free graph grammar. To overcome these limitations, Mark Minas and the author have proposed a modest extension of hyperedge replacement where the replacement graph of a variable x may not only be glued to the former attachments of x, but also to further nodes in the source graph [HM10]. This way of *contextual hyperedge replacement* does not only overcome the restrictions of context-free graph grammars (both the languages of all graphs and of program graphs can be derived), but later studies in [DHM12, DH14] have shown that many properties of hyperedge replacement are preserved, in particular, the existence of a recognition algorithm. Furthermore, they are suited to specify the refinement of rules by rules (in the next section).

For the definition of these grammars, we assume that the symbols Σ contain a set $X \subseteq \bar{\Sigma}$ of *variable names* that are used to label placeholders for subgraphs. $X(G) = \{e \in \bar{G} \mid \ell_G(e) \in X\}$ is the set of *variables* of a graph G, and $\underline{G}$ is its *kernel*, i.e., G without $X(G)$. For a variable $e \in X(G)$, the *variable subgraph* G/e consists of e and its attached nodes.

Graphs with variables are required to be *typed* in the following way: Variable names $x \in X$ are assumed to come with a *signature graph* $\mathsf{Sig}(x)$, which consists of a single edge labeled with x, to which all nodes are attached exactly once; in every graph G, the variable subgraph G/e must be isomorphic to the signature graph $\mathsf{Sig}(\ell_G(e))$, for every variable $e \in X(G)$.

Definition 3 (Contextual Grammar) A rule $r\colon (P \hookrightarrow B \hookleftarrow R)$ is *contextual* if the only edge e in its pattern P is a variable, and if its replacement R equals the body B, without e.

With some *start graph* Z, a finite set $\mathscr{R}$ of contextual rules forms a *contextual grammar* $\Gamma = (\Sigma, \mathscr{R}, Z)$ over the labels Σ, which derives the *language* $\mathscr{L}(\Gamma) = \{G \mid Z \Rightarrow^*_{\mathscr{R}} G, X(G) = \emptyset\}$.

The pattern P of a contextual rule r is the disjoint union of a signature graph $\mathsf{Sig}(x)$ with a discrete *context graph*, which is denoted as C_r. We call r *context-free* if C_r is empty. (Grammars with only such rules have been studied in the theory of hyperedge replacement [DHK97].)

Example 3 (A contextual grammar for program graphs) Figure 3 shows a set P of contextual rules. Variables are represented as boxes with their variable names inscribed; they are connected with their attached nodes by lines, ordered from left to right. (Later, in Sect. 3, we will also use arrows in either direction.) When drawing contextual rules like those in Fig. 3, we omit the box enclosing their pattern. The variable outside the replacement box is the unique edge in the pattern, and green filling (appearing grey in B/W print) designates the contextual nodes within the box enclosing the replacement graph.

The rules P in Figure 3 define a contextual grammar $\mathsf{PG} = (\Sigma, \mathsf{P}, \mathsf{Sig}(\mathsf{Cls}))$ for program graphs. The grammar uses four variable names; they are attached to a single node, which is labeled with C for variables named Cls and Fea, with B for variables named Bdy, and with E for variables named

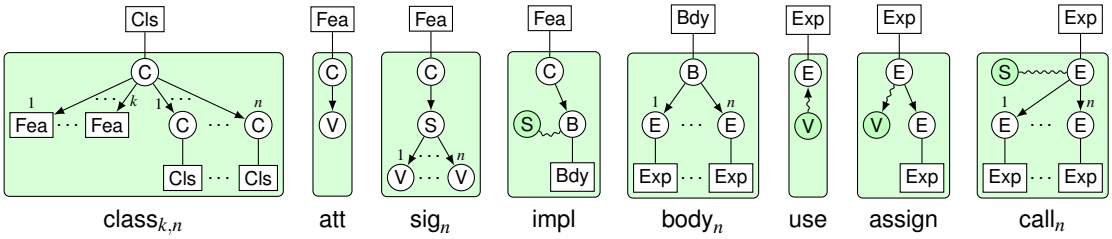

Figure 3: Contextual rules P for generating program graphs

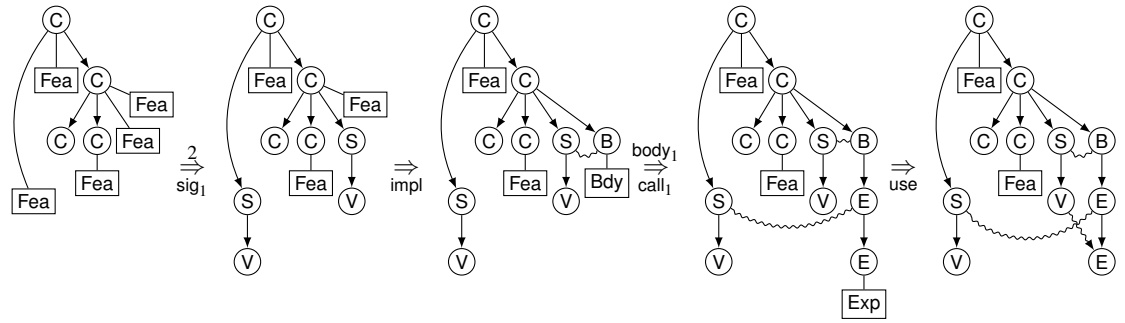

Figure 4: Snapshots in a derivation of a program graph

Exp. The C-node of the start graph Sig(Cls) represents the root class of the program, and the structure of a program is derived by the rules, considered from left to right, as follows. Every class has features, and may be extended by subclasses. A feature is either an attribute variable, or a method signature with parameter variables, or a method body that implements some existing signature. A method body (or rather, its data flow) consists of a set of expressions, which either *use* the value of some existing variable, or *assign* the value of an expression to some existing variable, or *call* some existing method signature with expressions as actual parameters. Actually, class$_{k,n}$, sig$_n$, body$_n$, and call$_n$ are templates for infinite sets of rule instances that generate classes with $k \geqslant 0$ features and $n \geqslant 0$ subclasses, signatures with $n \geqslant 0$ parameters, bodies with $n \geqslant 0$ expressions, and calls with $n \geqslant 0$ actual parameters, respectively. The instances of a template can be composed with a few replicative rules, so this is just a short hand notation, like the repetitive forms of extended Backus-Naur form of context-free string grammars.

Figure 4 shows snapshots in a derivation of a program graph. The first graph is derived from the start graph by applying four instances of the template class$_{k,n}$, which generate a root class with two features and one subclass, which in turn has two features and two subclasses, whereof only one has a feature, and both do not have subclasses. The second graph is obtained by applying rule instance sig$_1$ at two matches, deriving two signatures with one parameter each. Applying rule impl yields the third graph, with the root of a body for one of the signatures. The fourth is obtained by applications of the instances body$_1$ and call$_1$, refining the body to a single call. Then application of rule use derives the actual parameter for the call. Four further derivation steps, applying rules impl and att to the remaining Fea-variables, and rules body$_1$ and then use to the resulting Bdy-variable, yield the target graph of the rewriting step shown in Figure 1.

As for context-free string grammars, it is important to know whether a contextual grammar is ambiguous or not. Unambiguous grammars define unique (de-) compositions of graphs in their language. Parsing of unambiguous grammars is efficient as no backtracking is needed, and the transformation of graphs can be defined over their unique structure. This property will be exploited in Lemma 1 further below.

Definition 4 (Ambiguity) Let $\Gamma = (\Sigma, \mathscr{R}, Z)$ be a contextual grammar.

Consider two rewrite steps $G \Rightarrow^{m}_{r} H \Rightarrow^{m'}_{r'} K$ where $\tilde{m}\colon R \to H$ is the embedding of r in H. The steps may be *swapped* if $m'(P') \hookrightarrow \tilde{m}(P \cap R)$, yielding steps $G \Rightarrow^{m'}_{r'} H' \Rightarrow^{m}_{r} K$. Two rewrite sequences are *equivalent* if they can be made equal up to isomorphism, by swapping their steps repeatedly.

Then Γ is *unambiguous* if all rewrite sequences of a graph $G \in \mathscr{L}(\Gamma)$ are equivalent to each other; if some graph G has at least two rewrite sequences that are not equivalent, Γ is *ambiguous*.

Example 4 (Unambiguous Grammars) The program graph grammar PG in Example 3 is unambiguous.

3 Schema Refinement with Contextual Meta-Rules

Refining graph rewrite rules means to rewrite rules instead of graphs. A general framework for "meta-rewriting" can be easily defined. We start by lifting morphisms from graphs to rules.

Definition 5 (Rule Morphism) For (graph rewrite) rules r and s, a graph morphism $m\colon B_r \to B_s$ on their bodies is a *rule morphism*, and denoted as $m\colon r \to s$, if $m(P_r) \hookrightarrow P_s$ and $m(R_r) \hookrightarrow R_s$.

Graph rewrite rules and rule morphisms form a category. This category has pushouts, pullbacks, and unique pushout complements along injective rule morphisms, just as the category of graphs. As with graphs, we write rule inclusions as "$\hookrightarrow$", and let $\underline{r}$ be the *kernel* of a rule r wherein all variables are removed.

Definition 6 (Rule Rewriting) A pair $\delta\colon (p \hookrightarrow b \hookleftarrow r)$ of rule inclusions is a *rule rewrite rule*, or *meta-rule* for short. With δ_B we denote its *body rule*, which is a graph rewrite rule $(B_p \hookrightarrow B_b \hookleftarrow B_r)$ consisting of the bodies of p, b, and r.

Consider a rule s, a meta-rule δ as above, and a rule morphism $m\colon p \to s$. The meta-rule δ *rewrites* the source rule s at m to the target rule t, written $s \,{}_{\delta}\!\Downarrow_{m} t$, if there is a pair of pushouts

$$\begin{array}{ccccc} p & \hookrightarrow & b & \hookleftarrow & r \\ \downarrow & & \downarrow & & \downarrow \\ s & \hookrightarrow & u & \hookleftarrow & t \end{array}$$

The pushouts above exist if the underlying body morphism $m_B\colon B_p \to B_s$ of m satisfies the graph gluing condition wrt. the body rule δ_B and the body graph B_s; the target rule t is constructed by rewriting B_s to the body B_t with the body rule δ_B, and extending it to a rule $(P_t \hookrightarrow B_t \hookleftarrow R_t)$. As for graph rewriting, we assume that the pushouts are constructed so that all horizontal rule morphisms are rule inclusions, i.e., $s \hookrightarrow u \hookleftarrow t$.

It is straight-forward to define general rule rewriting on the more abstract level of adhesive categories. However, this is not useful for this paper, as we will use concrete meta-rules that are based on the restricted notion of contextual hyperedge replacement.

Let us recall the outstanding feature of rules in the graph rewriting tool GRGEN [BGJ06], the "*recursive pattern refinement*" devised by Edgar Jakumeit [Jak08], which we want to model.

- A rule may contain "*subpatterns*", which are names that are parameterized with nodes of the pattern and of the replacement graph of the rule. (If some parameter really is of the replacement graph, the term "*subrule*" would be more adequate.)
- The refinement of a subpattern is defined by a "*pattern rule*" that adds nodes and edges to the pattern and replacement graphs of a rule. Pattern rules may define alternative refinements, and may contain subpatterns so that they can be recursive.
- The refinements of different subpatterns must be disjoint, i.e., their matches in the source graph must not overlap. If a node shall be allowed to overlap with another node in the match, it is specified to be "*independent*".

We shall model subpatterns by allowing variables to occur in the body of a rule (but neither in its pattern, nor in its replacement); we call such a rule a schema. Pattern rules are modeled by alternative meta-rules where the body rule is contextual. This supports recursion, since the body of the meta-rule may contain variables. Rewriting with context-free meta-rules derives disjoint refinements for different variables in the context-free case; independent nodes can be modeled as the contextual nodes of contextual meta-rules.

Definition 7 (Schema Refinement) A *schema* $s\colon (P \hookrightarrow B \hookleftarrow R)$ is a graph rewrite rule with $P \cup R = \underline{B}$.

Every schema $s\colon (P \hookrightarrow B \hookleftarrow R)$ is required to be *typed* in the following sense: every variable name $x \in X$ comes with a *signature schema* $\mathsf{Sigschema}(x)$ with body $\mathsf{Sig}(x)$ so that for every variable $e \in X(B)$, the variable subgraph B/e is the body of a subschema that is isomorphic to $\mathsf{Sigschema}(x)$.

A meta-rule $\delta\colon (p \hookrightarrow b \hookleftarrow r)$ is *contextual* if p, b, and r are schemata, and if its body rule $\delta_B\colon (B_p \hookrightarrow B_b \hookleftarrow B_r)$ is a contextual rule so that the contextual nodes C_{δ_B} are in $P_p \cap R_p$.

In a *less contextual variation* δ' of a meta-rule δ, some contextual nodes are removed from B_p, but kept in B_r. Let Δ be a finite set of meta-rules that is closed under less contextual variations.[4] Then ${}_\Delta\Downarrow$ denotes refinement steps with one of its meta-rules, and ${}_\Delta\Downarrow_*$ denotes repeated refinement, its reflexive-transitive closure. $\Delta(s)$ denotes the *refinements* of a schema $s\colon (P \hookrightarrow B \hookleftarrow R)$, containing its refinements without variables:

$$\Delta(s) = \{r \mid s\,{}_\Delta\Downarrow_* r, X(B_r) = \emptyset\}$$

We write $G \Rightarrow_{\Delta(s)} H$ if $G \Rightarrow_r H$ for some $r \in \Delta(s)$, and say that the refinements $\Delta(s)$ *rewrite* G to H.

Note that the application of a refinement $r \in \Delta(s)$, although it is the result of a compound meta-derivation, is an indivisible rewriting step $G \Rightarrow_r H$ on the source graph G, similar to a transaction

[4] We explain in Example 5 why these less contextual variations are needed.

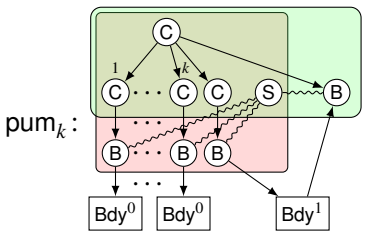

Figure 5: Pull-up Method: schema

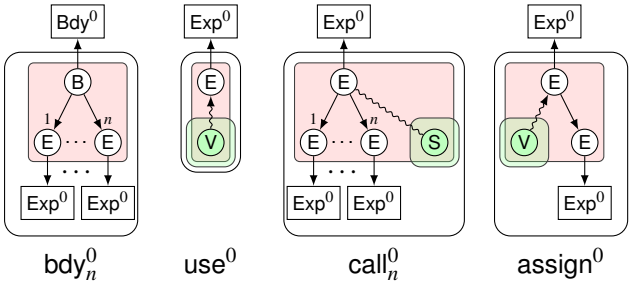

Figure 6: Deleting meta-rules for method bodies

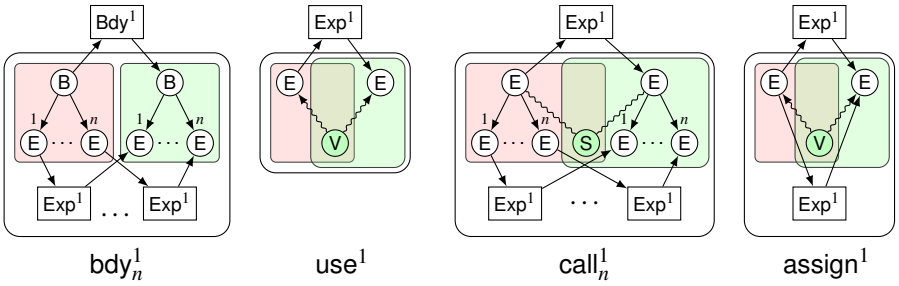

Figure 7: Replicating meta-rules for method bodies

in a data base. Note also that the refinement process is completely rule-based, and performs standard rewriting, just on rules instead of graphs.

We illustrate schema refinement by two operations from Fowler's catalogue of basic refactorings [Fow99].

Example 5 (Pull-Up Method) The *Pull-up Method* refactoring applies to a class c where all direct subclasses contain implementations for the same method signature that are semantically equivalent.[5] Then the refactoring pulls one of these implementations up to the superclass c, and removes all others.

Figures 5-7 show the schema pum_k and the meta-rules that shall perform this operation. In schemata and meta-rules, the lines between a variable e and a node v attached to e get arrow tips (i) at e if v occurs in the pattern, and (ii) at v if v occurs in the replacement. (Thus the line will have tips at both ends if v is both in the pattern and in the replacement. However, this occurs only in Fig. 8 of Example 6.) The pattern of the schema pum_k (in Fig. 5) contains a class with $k+1$ subclasses, where every subclass implements a common signature, as they contain B-nodes connected to the same S-node. (Actually, pum_k is a template for $k \geqslant 0$ schemata, like some of the contextual rules in Fig. 3. Analogously to contextual rules, the instances of the schema template can be derived with two contextual meta-rules.)

The variables specify what shall happen to the method bodies: k of them, those which are attached to a Bdy^0-variable, shall just be deleted, and the body attached to a Bdy^1-variable shall be moved to the superclass. The meta-rules can be mechanically constructed from the contextual

[5] This application condition cannot be decided mechanically; it has to be confirmed by the user when s/he applies the operation, by *a priori* verification or *a-posteriori* testing.

rules $\mathsf{M} = \{\mathsf{body}_n, \mathsf{use}, \mathsf{call}_n, \mathsf{assign}\} \subseteq \mathsf{P}$ in Fig. 3. The deleting meta-rules $\{r^0 \mid r \in \mathsf{M}\}$ in Fig. 6 delete all edges and all nodes but the contextual nodes of a method body from the pattern. The replicating meta-rules $\{r^0 \mid r \in \mathsf{M}\}$ in Fig. 7 delete a method body from the pattern, and insert a copy of this body in the replacement while preserving the contextual nodes. (Meta-rules for making $i > 1$ copies of a method body can be constructed analogously.) A context-free meta-rule, like those for Bdy^0 and Bdy^1, applies to every schema containing a variable of that name. A contextual meta-rule (like the other six), however, applies only if its contextual nodes can be matched in the schema. So the meta-rules call_n^0 and call_n^1 apply to the schema pum_k as it contains an S-node, but the others do not: neither does pum_k contain any V-node, nor does any of the meta-rules derive one. This is the reason for including less contextual variations of a contextual meta-rule r. In our case, where the rules have one contextual node only, the only less contextual variation is context-free, and denoted by $\bar{r}$. We do not not show them here, because the difference is small: just the green (contextual) nodes in Figure 6 and 7 turn white. Applying less contextual meta-rules to a schema adds the former contextual nodes to the interface of a schema, i.e., to the intersection of its pattern and replacement graphs.

If Δ_M is the closure of the meta-rules in Figure 6 and 7 under less contextual variation, refinement of the schema pum_k may yield method bodies with recursive calls to the signature in the schema, and calls to further signatures, and (read or write) accesses to variables in the interface. For instance, the rule pum' in Fig. 2 is a refinement of pum_k with Δ_M, i.e., $\mathsf{pum}' \in \Delta_\mathsf{M}(\mathsf{pum}_k)$. For deriving pum', only the context-free variations of meta-rules have been used, adding the former contextual nodes (drawn in green in Fig. 2) to the interface. The upper row in Fig. 13 on page 15 below shows a step in the refinement sequence $\mathsf{pum}_k \Downarrow_{*\,\Delta_\mathsf{M}} \mathsf{pum}'$; it applies the context-free variation $\overline{\mathsf{assign}}^1$ of the replicating meta-rule assign^1 in Fig. 7.

Example 6 (Encapsulate Field) The *Encapsulate Field* refactoring shall transform all non-local read and write accesses to an attribute variable by calls of getter and setter methods.

Figure 8 shows a schema ef, two meta-rule templates, and a refinement of the schema. The schema ef (on the left-hand side) adds a getter and a setter method definition for a variable to a

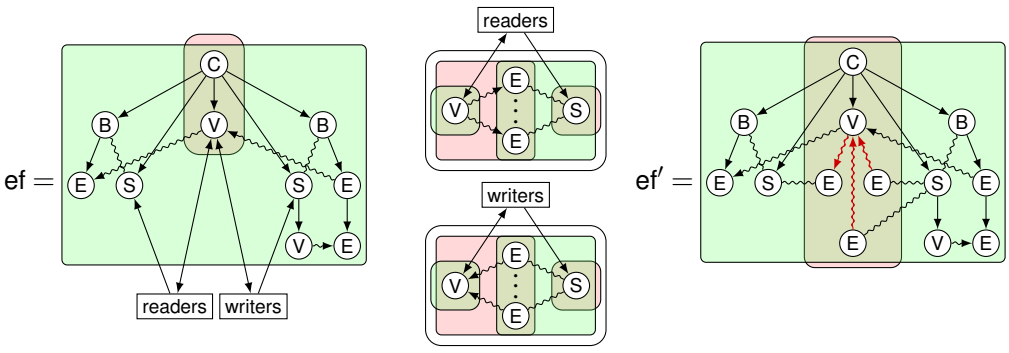

Figure 8: The schema, the embedding meta-rules, and an instance of *Encapsulate Field*

class, and introduces variables readers and writers that take care of the read and write accesses. The (context-free) embedding meta-rule templates (in the middle) then replace any number of read and write accesses to the variable by calls of its getter and setter method, respectively. If Δ_E denotes the embedding meta-rules, the rule on the right-hand side of Fig. 8 is a derivative $\mathsf{ef}' \in \Delta_E(\mathsf{ef})$, encapsulating one read access and two write accesses.

A single rewriting step with a refinement of *Pull-up Method* copies one method body of arbitrary shape and size, and deletes an arbitrarily number of other bodies, also of variable shape and size. Refinements of *Encapsulate Field* transforms the neighbor edges of an unbounded number of nodes. This goes beyond the expressiveness of plain rewrite rules, which may only match, delete, and replicate subgraphs of constant size and fixed shape. Many of the basic other refactorings from Fowler's catalogue [Fow99] cannot be specified by a single plain rule, but by a schema with appropriate meta-rules.

Operationally, we cannot construct all refinements of a schema s first, and apply one of them later, because the set $\Delta(s)$ is infinite in general. Rather, we interleave matching and refinement, in the next section. Before, we study some properties of schema refinement.

The following assumption excludes useless definitions of meta-rules.

Assumption 1 The set $\Delta(s)$ of refinements of a schema s shall be non-empty.

Non-emptiness of refinements can be reduced to the property whether the language of a contextual grammar is empty or not. It is shown in [DHM12, Corollary 2] that this property is decidable.

We need a mild condition to show that schema refinement terminates.

Definition 8 (Pattern-Refining Meta-Rules) A meta-rule $\delta\colon (p \hookrightarrow b \hookleftarrow r)$ *refines its pattern* if $X(R_r) = \emptyset$ or if $P_r \ncong P_p$. A set Δ of meta-rules that refine their patterns is called *pattern-refining*.

Theorem 1 *For a schema s and a set Δ of pattern-refining meta-rules, it is decidable whether some refinement $r \in \Delta(s)$ applies to a graph G, or not.*

Proof. By Algorithm 1 in [Hof13], the claim holds under the condition that meta-rules "do not loop on patterns". It is easy to see that pattern-refining meta-rules are of this kind. □

We now turn to the question whether the (infinite) set of graph rewrite rules obtained as refinements of a schema are uniquely determined by their patterns.

Definition 9 (Right-Unique Rule Sets) A set $\mathscr{R}$ of graph rewrite rules is *right-unique* if different meta-rules $r_1\colon (P_1 \hookrightarrow B_1 \hookleftarrow R_1), r_2\colon (P_2 \hookrightarrow B_2 \hookleftarrow R_2) \in \mathscr{R}$ have different patterns, i.e., $P_1 \cong P_2$ implies that $r_1 \cong r_2$.

We define an auxiliary notion first. The *pattern rule* δ_P of a meta-rule $\delta\colon (p \hookrightarrow b \hookleftarrow r)$ is a contextual rule obtained from the body rule $\delta_B : (B_p \hookrightarrow B_b \hookleftarrow B_r)$ by removing all nodes and edges in $B_b \setminus R_b$, and by detaching all variables in δ_B from the removed nodes. Let Δ_P denote the set of (contextual) *pattern rules* of a set Δ of meta-rules. (The graphs in Δ_P are typed as well, but

in the type graph $\mathsf{Sig}(x)$ of a variable name x, all nodes that do not belong to the pattern of the signture schema $\mathsf{Sigschema}(x)$ are removed.)

Lemma 1 (Right-Uniqueness of Refinements) *A set $\Delta(s)$ of refinements is right-unique if the pattern grammar (Σ, Δ_P, P_s) of their meta-rules Δ is unambiguous.*

Proof Sketch. *Consider rules $r_1, r_2 \in \Delta(s)$ with $P_1 \cong P_2$. Then $P_s \Rightarrow_{\Delta_P} P_1$ and $P_s \Rightarrow_{\Delta_P} P_2$. The rewrite sequences can be made equal since Δ_P is unambiguous. This rewriting sequence has a unique extension to a meta-rewrite sequence so that $r_1 \cong r_2$.* □

Example 7 (Properties of Meta-Rules) It is easy to see that the deleting and replicative meta-rules Δ_{M} in Figures 6-7 of Example 5 satisfy Assumption 1. Because, it has been shown in [DHM12, Example 3.23] that all rules of the program graph grammar in Example 3 are useful. Thus its language is non-empty. This property can easily be lifted to the meta-rules Δ_{M}, in particular as they also contain context-free variations of the rules in P. It is easy to check that the rules in Δ_{M} are also pattern-refining. The contextual rules P for method bodies in Fig. 3 are unambiguous, and so are the rules M, which correspond to the pattern rules of the deleting and replicating meta-rules Δ_{M} in Figures 6-7 of Example 5, so that Δ_{M} is right-unique.

The embedding meta-rules Δ_{E} in Fig. 8 of Example 6 derive a non-empty set of rules, and are pattern-refining and right-unique as well.

4 Modeling Refinement by Residual Rewriting

The refinement of a schema s with some meta-rules Δ yields instances $\Delta(s)$, which are ordinary rules for rewriting graphs. However, the set $\Delta(s)$ is infinite in general. Unfortunately, many analysis techniques, e.g., for termination, confluence, and state space exploration of graph rewriting, do only work for finite sets of graph rewrite rules. To make these techniques applicable, we translate each schema and every contextual meta rule into a standard graph rewrite rule:

- We turn every schema into an ordinary rule that postpones refinement, by adding its meta-variables to its replacement.
- We turn every contextual meta-rule $\delta\colon (p \hookrightarrow b \hookleftarrow r)$ into an graph rewrite rule that refines the translated schema incrementally, by uniting its pattern rule r component-wise with the variable graphs of its body rule δ_B.

The resulting rule set is always finite.

Definition 10 (Incremental Refinement Rules) Let $s\colon (P \hookrightarrow B \hookleftarrow R)$ be a schema for meta-rules Δ.

The *incremental rule* $\tilde{s}\colon (P \hookrightarrow B \hookleftarrow R_{\tilde{s}})$ of the schema s has the same pattern P and body B as s, and its replacement $R_{\tilde{s}} = R \cup \{B/e \mid e \in X(B)\}$ is obtained by extending R with the graphs of all variables in B.

For a meta-rule $\delta = (p \hookrightarrow b \hookleftarrow r)$ in Δ, the *incremental rule* $\tilde{\delta}\colon (\tilde{P} \hookrightarrow \tilde{B} \hookleftarrow \tilde{R})$ is the component-wise union of its replacement rule $r\colon (R_p \hookrightarrow R_b \hookleftarrow R_r)$ with the variable graphs of its body rule $\delta_B : (B_p \hookrightarrow B_b \hookleftarrow B_r)$:

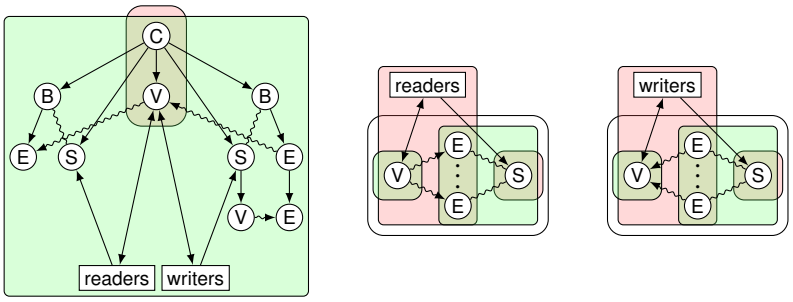

Figure 9: Incremental rules for *Encapsulate Field* in Fig. 8

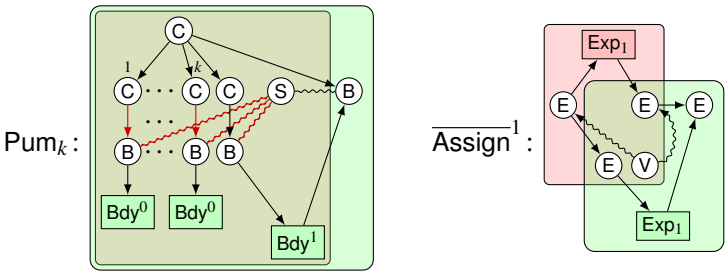

Figure 10: Incremental rules for *Pull-up Method* in Fig. 5 and Fig. 7

(i) $\tilde{P} = P_r \cup \{B_p/e \mid e \in X(B_p)\}$ (which equals $B_p \cup P_r$),
(ii) $\tilde{B} = B_b \cup \{B_b/e \mid e \in X(B_b)\}$ (which equals B_b), and
(iii) $\tilde{R} = R_r \cup \{B_r/e \mid e \in X(B_r)\}$.

$\tilde{\Delta}$ shall denote the incremental rules of the meta-rules Δ.

Example 8 (Incremental Refinement) Figure 9 shows the incremental rule Ef for the schema ef and of the meta-rules Δ_{E} in Fig. 8 of Example 6.

Figure 10 shows how the schema pum_k for the *Pull-up Method* refactoring in Fig. 5 is translated into an incremental rule Pum_k, and how the context-free variation $\overline{\mathsf{assign}}^1$ of the meta-rule assign^1 in Fig. 7 is translated into an incremental rule $\overline{\mathsf{Assign}}^1$. (In the incremental rule Pum_k, red arrow and waves indicate edges that do not belong to the replacement.)

If a schema s is refined with a meta-rule δ to a schema t, the composition $\tilde{s} \circ_d \tilde{\delta}$ of its incremental rules (as defined in Def. 12 of the appendix) equals the incremental rule $\tilde{t}$ (for a particular dependency d).

Lemma 2 *Consider a schema $s = (P \hookrightarrow B \hookleftarrow R)$ and a meta-rule $\delta\colon (p \hookrightarrow b \hookleftarrow r)$.*
Then $s \Downarrow_{\delta m} t$ for some schema t iff there is a composition $r^d = \tilde{s} \circ_d \tilde{\delta}$ for a dependency

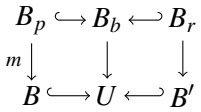

Figure 11: $B \Rightarrow^{m}_{\delta_B} B'$

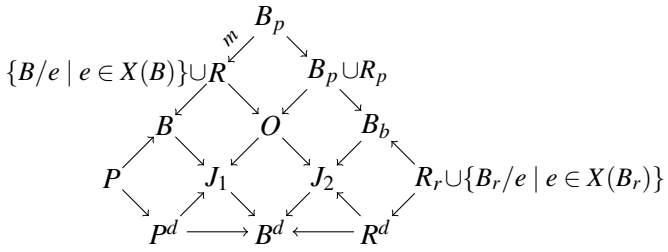

Figure 12: $r^d = \tilde{s} \circ_d \tilde{\delta}$

$d\colon (R \xleftarrow[m]{} B_p \to (B_p \cup R_p))$ *so that* $r^d = \tilde{t}$.

Proof Sketch. *Let* s, δ *be as above,* $t\colon (P' \hookrightarrow B' \hookleftarrow R')$, $\tilde{s}\colon (P \hookrightarrow B \hookleftarrow R_{\tilde{s}})$ *with* $R_{\tilde{s}} = R \cup \{B/e \mid e \in X(B)\}$, *and* $\tilde{\delta}\colon (\tilde{P} \hookrightarrow \tilde{B} \hookleftarrow \tilde{R})$ *with* $\tilde{P} = B_p \cup P_r$, $\tilde{R} = R_r \cup \{B_r/e \mid e \in X(B_r)\}$, *and* $\tilde{B} = B_b$, *see Def. 10. Their composition according to the dependency* $d\colon (R \xleftarrow[m]{} B_p \to (B_p \cup R_p))$ *is constructed as in Def. 12, and shown in Fig. 12.*

Consider the underlying body refinement $B \Rightarrow^{m}_{\delta_B} B'$. *(See Fig. 11, where we assume that the lower horizontal morphisms are inclusions.) By uniqueness of pushouts,* $U \cong B_d$. *Then* $(B_b \setminus B_p) = X(B_p)$ *since* δ_B *is contextual, and* $B' = U \setminus \bar{m}(X(B_p))$.

It is then easy to show that the body $B'_{\tilde{\delta}}$ *equals the body* B^d *of the composed incremental rule, and an easy argument concerning the whereabouts of variables shows that* $\tilde{t} = r^d$. □

Example 9 (Schema Refinement and Incremental Rules) Figure 13 illustrates the relation between schema refinement and the composition of their incremental rules established in Lemma 2. As already mentioned in Example 5, the upper row shows a step in the refinement sequence $\mathsf{pum}_k \Downarrow_{\Delta_M *} \mathsf{pum}'$ that applies the context-free variation $\overline{\mathsf{assign}}^1$ of the meta-rule assign^1 in Fig. 7. The original meta-rule does not apply to the source schema, as it does not contain a node labeled V. The less contextual rule does apply; the refined rule is constructed so that The V-node will be matched in the context when it is applied to a source graph.

The lower row shows the composition of the corresponding incremental rule with the corresponding incremental refinement rule $\overline{\mathsf{Assign}}^1$, where the dashed box specifies the dependency d for the composition. The composed rule equals the incremental rule for the refined schema.

Using a refined schema has the same effect as applying its incremental rule, and the incremental rules of the corresponding meta-rules. This must follow a strategy that applies incremental rules as long as possible, matching the residuals of the source graphs, before another incremental rule is applied.

We define the subgraph that is left unchanged in refinement steps and sequences. The *track* of G in H (via the match m of the rule r) is then defined as $tr^m_r(G) = (G \cap H)$.[6] For a rewrite sequence $d = G_0 \Rightarrow^{m_1}_{r_1} G_1 \Rightarrow^{m_2}_{r_2} \ldots \Rightarrow^{m_n}_{r_n} G_n$, the track of G in H is given by intersecting the tracks

[6] Recall that $G \hookrightarrow U \hookleftarrow H$ for the graphs and morphisms of a rewrite step.

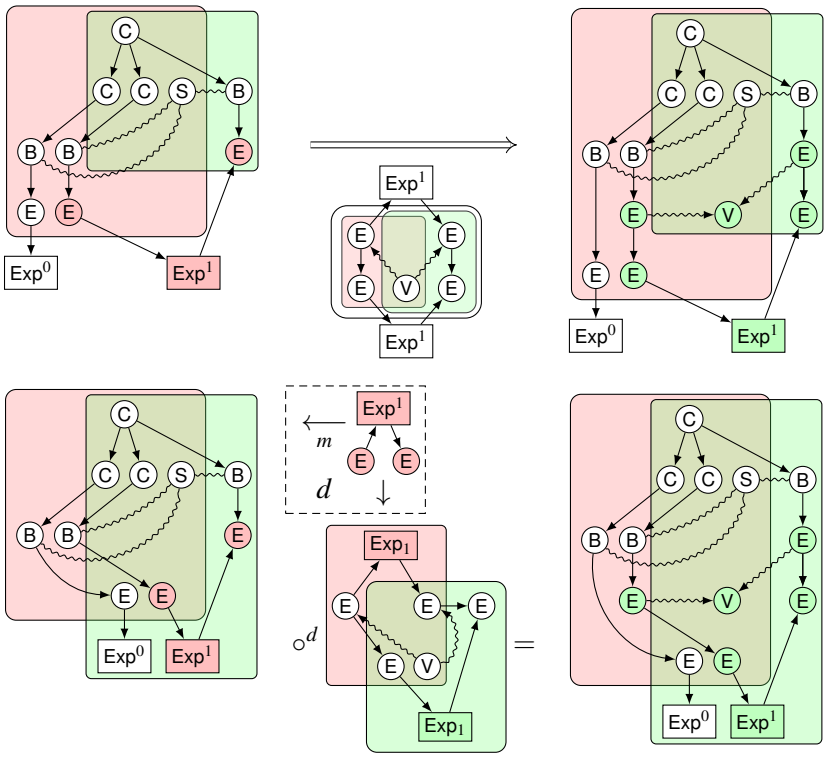

Figure 13: Schema refinement and incremental composition

of its steps:

$$tr_d(G) = tr_{r_1}^{m_1}(G_0) \cap \cdots \cap tr_{r_n}^{m_n}(G_{n-1})$$

The incremental rules have to be applied so that the patterns of the refinements of the original meta-rules do not overlap.

Definition 11 (Residual Incremental Refinement) Consider an incremental refinement sequence

$$G_0 \overset{m_1}{\underset{\tilde{\delta}_1}{\Rightarrow}} G_1 \overset{m_2}{\underset{\tilde{\delta}_2}{\Rightarrow}} \ldots \overset{m_n}{\underset{\tilde{\delta}_n}{\Rightarrow}} G_n$$

with incremental rules $\tilde{\delta}_i$ for meta-rules $\delta_i \colon (p_i \hookrightarrow b_i \hookleftarrow r_i)$ (for $1 \leqslant i \leqslant n$).

The step $G_{i-1} \overset{m_i}{\underset{\tilde{\delta}_i}{\Rightarrow}} G_i$ is *residual* if $m_i(P_{r_i}) \subseteq tr_{r_1 \ldots r_{i-1}}^{m_1 \ldots m_{i-1}}(G)$. The sequence is residual if every of its steps is residual. Residual steps and sequences are denoted as $\Rightarrow$ and $\Rightarrow^*$, respectively.

Lemma 3 *Consider a schema s for meta-rules Δ with incremental rule $\tilde{s}$ and incremental rules $\tilde{\Delta}$.*

Then a rule $r \colon (P \hookrightarrow B \hookleftarrow R)$ is a refinement in $\Delta(s)$ if and only if $P \Rightarrow_{\tilde{s}} P' \overset{!}{\Rightarrow}_{\tilde{\Delta}} R$.

Proof. By induction over the length of meta-derivations, using Lemma 2 and the fact that compositions correspond to residual rewrite steps. □ □

Theorem 2 *Consider a schema s with meta-rules Δ as above. Then, for graphs G, H, and K, $G \Rightarrow_{\Delta(s)} H$ if and only if $G \Rightarrow_{\bar{s}} K \Rightarrow^{!}_{\bar{\Delta}} H$.*

Proof. Combine Lemma 3 with the embedding theorem [EEPT06, Sect. 6.2]. □

5 Conclusions

In this paper we have continued earlier attempts in [HJG08, Hof13] to model graph rewriting with recursive refinement, which are the outstanding feature of rules in the graph rewriting tool GRGEN [BGJ06]. The definition here is by standard graph rewriting—contextual hyperedge replacement on the meta-level and standard graph rewriting on the object level—and allows to specify conditions under which refinement "behaves well", i.e., terminates, and yields unique refinements. It is simple enough so that it can be translated to standard graph rewriting rules that perform the refinement incrementally, using a strategy—residual rewriting—where matches do overlap only in contextual nodes (and in attached nodes of variables).

Related Work has occurred with two respects. On the one hand, expressive rules allowing transform subgraphs of variable shape and size have been proposed by several authors: D. Janssens has studied graph rewriting with node embedding rules [Jan83]. The *Encapsulate Field* refactoring in Example 6 could be defined in this way. D. Plump and A. Habel have proposed rules where variables in the pattern and the replacement graph can be substituted with isomorphic graphs [PH96]. There, variables could be substituted by arbitrary graphs, which is rather powerful, but difficult to use (and to implement). The author has later proposed substitutions with context-free (hyperedge replacement) languages [Hof01]. This turned out to be too restricted so that we now decided to propose contextual hyperedge replacement. The *Pull-Up Method* refactoring in Example 5 is a candidate for substitutive graph rewriting. In [Hof13] we have shown that embedding and substitutive rules are special cases of rules with contextual refinement.

On the other hand, the core of standard graph rewriting theory [CEH+97], with its results on parallel and sequential independence, critical pair lemma [Plu93] etc., has been extended considerably over the years. The framework now covers graph with attributes and subtyping, rules with positive and negative application conditions [EEPT06], and, as of recently, also nested application conditions [EHL+10, EGH+12].

Future Work should attempt to integrate the extensions of the standard theory to rule refinement, as all these concepts are supported by the graph rewriting tool GRGEN [BGJ06] as well. This should be straight-forward for attributes and subtyping. Application conditions require more work, in particular when conditions shall be translated to incremental rules. Obviously, application conditions are useful for modeling complex operations like refactorings: (i) The definition of program graphs in Example 1 could be more precise if the choice of a contextual node could be subject to a condition. E.g., the rule impl should require that the signature being implemented is contained in a super-class of the body. (See [HM10] for a definition of program graphs using application conditions.) (ii) The *Pull-up Method* refactoring in Example 5 should require that

the method body to be pulled up does not access variables or methods outside the name space of the superclass. (iii) The *Encapsulate Field* refactoring in Example 6 should be required to encapsulate all non-local accesses of a variable. For some of the conditions mentioned here, application conditions need to specify the (non)-existence of paths in a graph. This cannot be done by nested application conditions, but only if the conditions allow recursive refinement, as studied by H. Radke in [HR10]. But this is not (yet?) integrated into the standard theory.

Our ultimate goal is to provide support for analyzing GRGEN rules, e.g., for the existence of critical pairs. The negative result shown in [Hof13, Thm. 3] indicates that considerable restrictions have to be made to reach this aim. Our idea now is to restrict rewriting with contextual refinement to graphs that are *shaped* according to a contextual grammar like that for program graphs.

Acknowledgments.

The author thanks Annegret Habel and Rachid Echahed for their encouragement, and the reviewers for their detailed constructive comments.

Bibliography

[BGJ06] J. Blomer, R. Geiß, E. Jakumeit. GRGEN.NET: A Generative System for Graph-Rewriting, User Manual. www.grgen.net, Universität Karlsruhe, 2006. Version 4.4 (29.07. 2014).

[CEH+97] A. Corradini, H. Ehrig, R. Heckel, M. Löwe, U. Montanari, F. Rossi. Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*. Chapter 3, pp. 163–245. World Scientific, 1997.

[DH14] F. Drewes, B. Hoffmann. Contextual Hyperedge Replacement. Uminf report 14.04, Institutionen för datavetenskap, Umeaå universitet, 2014. 28 pages.

[DHK97] F. Drewes, A. Habel, H.-J. Kreowski. Hyperedge Replacement Graph Grammars. Chapter 2, pp. 95–162 in [Roz97].

[DHM12] F. Drewes, B. Hoffmann, M. Minas. Contextual Hyperedge Replacement. In Schürr et al. (eds.), *Applications of Graph Transformation with Industrial Relevance (AGTIVE'11)*. Lecture Notes in Computer Science 7233, pp. 182–197. Springer, 2012. Long version as UMINF report 14.04, Institutionen för datavetenskap,Umeå universitet.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs. Springer, 2006.

[EGH+12] H. Ehrig, U. Golas, A. Habel, L. Lambers, F. Orejas. Adhesive Transformation Systems with Nested Application Conditions. Part 2: Embedding, Critical Pairs and Local Confluence. *Fundam. Inform.* 118(1-2):35–63, 2012.

[EHL+10] H. Ehrig, A. Habel, L. Lambers, F. Orejas, U. Golas. Local Confluence for Rules with Nested Application Conditions. In Ehrig et al. (eds.), *ICGT*. Lecture Notes in Computer Science 6372, pp. 330–345. Springer, 2010.

[EHP09] H. Ehrig, F. Hermann, U. Prange. Cospan DPO Approach: An Alternative for DPO Graph Transformations. *Bulletin of the EATCS* 98:139–149, 2009.

[EPS73] H. Ehrig, M. Pfender, H. Schneider. Graph Grammars: An Algebraic Approach. In *IEEE Conf. on Automata and Switching Theory*. Pp. 167–180. Iowa City, 1973.

[ER97] J. Engelfriet, G. Rozenberg. Node Replacement Graph Grammars. Chapter 1, pp. 1–94 in [Roz97].

[ERT99] C. Ermel, M. Rudolf, G. Taentzer. The AGG Approach: Language and Environment. In Engels et al. (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. II: Applications, Languages, and Tools*. Chapter 14, pp. 551–603. World Scientific, Singapore, 1999.

[Fow99] M. Fowler. *Refactoring—Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, Reading, MA, 1999.

[HJG08] B. Hoffmann, E. Jakumeit, R. Geiß. Graph Rewrite Rules with Structural Recursion. In Mosbah and Habel (eds.), *2nd Intl. Workshop on Graph Computational Models (GCM 2008)*. Pp. 5–16. 2008.

[HM10] B. Hoffmann, M. Minas. Defining Models – Meta Models versus Graph Grammars. *Elect. Comm. of the EASST* 29, 2010. Proc. 6th Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'10), Paphos, Cyprus.

[HMP01] A. Habel, J. Müller, D. Plump. Double-Pushout Graph Transformation Revisited. *Mathematical Structures in Computer Science* 11(5):637–688, 2001.

[Hof01] B. Hoffmann. Shapely Hierarchical Graph Transformation. In *Proc. IEEE Symposia on Human-Centric Computing Languages and Environments*. Pp. 30–37. IEEE Computer Press, 2001.

[Hof13] B. Hoffmann. Graph Rewriting with Contextual Refinement. *Electr. Comm. of the EASST* 61:20 pages, 2013.

[HR10] A. Habel, H. Radke. Expressiveness of graph conditions with variables. *Elect. Comm. of the EASST* 30, 2010. International Colloquium on Graph and Model Transformation (GraMoT'10).

[Jak08] E. Jakumeit. *Mit* GRGEN *zu den Sternen*. Diplomarbeit (in German), Universität Karlsruhe, 2008.

[Jan83] D. Janssens. *Node Label Controlled Graph Grammars*. PhD thesis, Antwerp, 1983.

[MEDJ05] T. Mens, N. V. Eetvelde, S. Demeyer, D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance* 17(4):247–276, 2005. doi:10.1002/smr.316 http://dx.doi.org/10.1002/smr.316

[PH96] D. Plump, A. Habel. Graph Unification and Matching. In Cuny et al. (eds.), *Proc. Graph Grammars and Their Application to Computer Science*. Lecture Notes in Computer Science 1073, pp. 75–89. Springer, 1996.

[Plu93] D. Plump. Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence. In Sleep et al. (eds.), *Term Graph Rewriting, Theory and Practice*. Pp. 201–213. Wiley & Sons, Chichester, 1993.

[Ren04] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In Nagl et al. (eds.), *Applications of Graph Transformation with Industrial Relevance (AGTIVE'03)*. Lecture Notes in Computer Science 3062, pp. 479–485. Springer, 2004.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations*. World Scientific, Singapore, 1997.

[Ues78] T. Uesu. A System of Graph Grammars which Generates all Recursively Enumerable Sets of Labelled Graphs. *Tsukuba J. Math.* 2:11–26, 1978.

[VJ03] N. Van Eetvelde, D. Janssens. A Hierarchical Program Representation for Refactoring. *Electronic Notes in Theoretical Computer Science* 82(7), 2003.

A Double-Pushout Rewriting

The standard theory of graph rewriting is based on so-called *spans* of (injective) graph morphisms [EEPT06], where a rule consists of two morphisms from a common interface I to a pattern P and a replacement R. An alternative proposed in [EHP09] uses so-called co-spans (or joins) of morphisms where the pattern and the replacement are both included in a common supergraph, which we call the body of the rule.

Rewriting is defined by double pushouts as below:

$$\hat{r}\colon \begin{array}{ccccc} P & \hookleftarrow & I & \hookrightarrow & R \\ {\scriptstyle m}\downarrow & & \downarrow & & \downarrow \\ G & \hookleftarrow & C & \hookrightarrow & H \end{array} \qquad \check{r}\colon \begin{array}{ccccc} P & \hookrightarrow & B & \hookleftarrow & R \\ {\scriptstyle m}\downarrow & & \downarrow & & \downarrow \\ G & \hookrightarrow & U & \hookleftarrow & H \end{array}$$

Intuitively, rewrites are constructed via a match morphism $m\colon P \to G$ in a source graph G; for a span rule $\hat{r}$, removing the match of obsolete pattern items $P \setminus I$ yields a context graph C to which the new items $R \setminus I$ of the replacement are then added; for a co-span rule $\check{r}$, the new items $B \setminus P$ are added first, yielding the united graph U before the obsolete pattern items $B \setminus R$ are removed. The constructions work if the matches m satisfy certain *gluing conditions*.

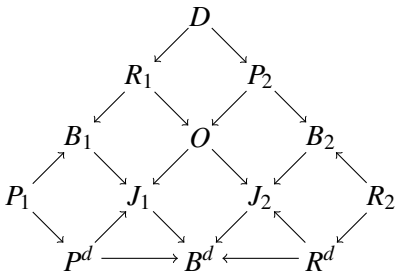

Figure 14: Sequential composition of graph rewrite rules

The main result of [EHP09] says that $\check{r}$ is the pushout of $\hat{r}$, making these rules, their rewrite steps, and gluing conditions dual to each other. Therefore we feel free to use the more intuitive gluing condition for $\hat{r}$ together with a rule $\check{r}$.

The following definition and theorem adapt well-known concepts of [EEPT06] to our notion of rules.

Definition 12 (Sequential Rules Composition) Let $r_1 \colon (P_1 \hookrightarrow B_1 \hookleftarrow R_1)$ and $r_2 \colon (P_2 \hookrightarrow B_2 \hookleftarrow R_2)$ be rules, and consider a graph D with a pair $d \colon (R_1 \leftarrow D \rightarrow P_2)$ of injective morphisms.

1. Then d is a *sequential dependency* of r_1 and r_2 if $D \not\leftrightarrow P_1$ (which implies that $D \neq \langle\rangle$).
2. The *sequential composition* $r_1 \circ_d r_2 \colon (P^d \hookrightarrow B^d \hookleftarrow R^d)$ of r_1 and r_2 along d is the rule constructed as in the commutative diagram of Fig. 14, where all squares are pushouts.
3. Two rewrite steps $G \Rightarrow_{r_1} H \Rightarrow_{r_2} K$ are *d-related* if d is the pullback of the embedding $R_1 \rightarrow H$ and of the match $P_2 \rightarrow O$.[7]

Proposition 1 *Let r_1 and r_2 be rules with a dependency d and a sequential composition r^d as in Def. 12.*

Then there exist d-related rewrite steps $G \Rightarrow_{r_1} H \Rightarrow_{r_2} K$ if and only if $G \Rightarrow_{r^d} K$.

Proof. Straightforward use of the corresponding result for "span rules" [EEPT06, Thm. 5.23] and of the duality to "co-span rules" [EHP09]. □

[7] A *pullback* of a pair of morphisms $B \rightarrow D \leftarrow C$ with the same codomain is a pair of morphisms $B \leftarrow A \rightarrow C$ that is *commutative*, i.e., $A \rightarrow B \rightarrow D = A \rightarrow C \rightarrow D$, and *universal*, i.e., for every pair of morphisms $B \leftarrow A' \rightarrow C$ so that $A' \rightarrow B \rightarrow D = A' \rightarrow C \rightarrow D$, there is a unique morphism $A' \rightarrow A$ so that $A' \rightarrow A \rightarrow B = A' \rightarrow B$ and $A' \rightarrow A \rightarrow C = A' \rightarrow C$. See [EEPT06, Def. 2.2].