Electronic Communications of the EASST
Volume 61 (2013)

Selected Revised Papers from the
4th International Workshop on
Graph Computation Models
(GCM 2012)

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

Graph Rewriting with Contextual Refinement

Berthold Hoffmann

Fachbereich Mathematik und Informatik, Universität Bremen, Germany

Abstract: In the standard theory of graph transformation, a rule modifies only subgraphs of constant size and fixed shape. The rules supported by the graph-rewriting tool GRGEN are far more expressive: they may modify subgraphs of unbounded size and variable shape. Therefore properties like termination and confluence cannot be analyzed as for the standard case. In order to lift such results, we formalize the outstanding feature of GRGEN rules by using plain rules on two levels: *schemata* are rules with variables; they are refined with *meta-rules*, which are based on contextual hyperedge replacement, before they are used for rewriting. We show that every rule based on single pushouts, on neighborhood-controlled embedding, or on variable substitution can be modeled by a schema with appropriate meta-rules. It turns out that the question whether schemata may have overlapping refinements is not decidable.

Keywords: graph rewriting – two-level rewriting – critical overlap

1 Introduction

Everywhere in computer science and beyond, one finds systems with a structure represented by graph-like diagrams, and with a behavior 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 is a natural formalism, which has been used to specify such systems in an abstract way, ever since this branch of theoretical computer science emerged in the seventies of the last century [EPS73]. 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: Does the system terminate? Is it *confluent*? I.e., does rewriting of some start state lead to a unique final state?

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 by the analysis of critical pairs [Plu93], and GROOVE [Ren04] allows to explore the state space. Some years ago, the very efficient *graph rewrite generator* GRGEN has been developed in the group of Gerhard Goos at Karlsruhe Institute of Technology [BGJ06]. This tool supports an object-oriented graph model with subtyping and attributes, and rewrite rules with negative application conditions. Recently, Edgar Jakumeit has extended the rules of GRGEN radically, by introducing recursive refinement of sub-patterns, application conditions, and sub-rules [Jak08, HJG08]. A single rule of this kind 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 plain theory [EEPT06] covers only graphs and rules with attributes and subtyping; recently, it has been extended to cover application conditions, even in nested form [EHL+10, EGH+12]. However, it does not capture recursive refinement, so that such rules cannot be analyzed for properties like termination and confluence, and no tool support concerning these questions can be provided.

Our ultimate goal is to lift results concerning termination and confluence to rules with recursive refinement. As a first step, 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, using meta-rules based on contextual hyperedge replacement [HM10, 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. In this paper, we explore the expressiveness of rule refinement in comparison to other kinds of rules, which are based on single pushouts, neighborhood-controlled embedding, and variable substitution. It turns out that the expressiveness has a price: it can, in general, not be decided whether two schemata can have refinements that are parallelly dependent.

The examples shown in this paper arise in the area of model-driven software engineering. *Refactoring* shall improve the structure of object-oriented software systems (models) without changing their behavior. Graphs are a straight-forward representation for the syntax and semantic relationships of object-oriented programs (and 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 it can be specified by a single rule schema with appropriate contextual meta-rules, in a completely declarative way.

The paper is organized as follows. The next section defines graphs, plain rules for graph rewriting, and restricted rules – called "contextual" – for deriving sets, or languages, of graphs. In Sect. 3 we recall related work on substitutive rules, and define schemata, meta-rules, and the refinement mechanism. Then we relate rule schemata to other kinds of graph rewrite rules, in Sect. 4. It turns out that a rule defined by single pushouts, by neighborhood-controlled embedding, or by substitution of graph variables can be modeled by a single schema with appropriate meta-rules. In Sect. 5 we study the existence of overlaps for schemata. We conclude by indicating future work, in Sect. 6.

2 Graphs, Rewriting, and Derivation

We define 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 [Hab92]. A pair $\mathscr{C} = (\dot{\mathscr{C}}, \vec{\mathscr{C}})$ of finite sets of *colors* is used to label nodes and edges of a graph, where a subset $\mathscr{X} \subseteq \vec{\mathscr{C}}$ with a *type function type*: $\mathscr{X} \to \dot{\mathscr{C}}^*$ shall be used to name *variables*, which only appear in rules.

Definition 1 (Graph) A *graph* $G = (\dot{G}, \vec{G}, att, \ell)$ consists of disjoint finite sets $\dot{G}$ of *nodes* and $\vec{G}$ of *edges*, a function $att\colon \vec{G} \to \dot{G}^*$ that *attaches* sequences of nodes to edges; and of a pair $\ell = (\dot{\ell}, \vec{\ell})$ of *labeling functions* $\dot{\ell}\colon \dot{G} \to \dot{\mathscr{C}}$ for nodes and $\vec{\ell}\colon \vec{G} \to \vec{\mathscr{C}}$ for edges so that, for every *variable*, i.e., every edge $x \in \vec{G}$ with $\ell_G(x) \in \mathscr{X}$, the attached nodes $att_G(x)$ are distinct to each other, and labeled so that $\dot{\ell}^*_G(att_G(x)) = type(\ell_G(x))$.[1] We will often refer to the component functions of a graph G by att_G and ℓ_G.

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

For a graph G, $X_G = \{x \in \vec{G} \mid \vec{\ell}_G(x) \in \mathscr{X}\}$ is the set of *variables*, and its *kernel* $\underline{G}$ is the subgraph without X_G; G is called *terminal* if $X_G = \emptyset$. The subgraph of G consisting of a variable $x \in X_G$ and its attached nodes is called a *star* of x, and denoted by $\langle x \rangle$. The kernel of a star is discrete, it consists of the attached nodes of x, and is denoted as $\langle \underline{x} \rangle$.

Example 1 (Program Graphs) *Figure 1 shows a program graph. Circles represent nodes, and have their labels inscribed. In our examples, terminal edges are always attached to exactly two nodes, a* source *and a* target, *and are drawn as straight or wave-like arrows between these nodes. Boxes represent variables, to be seen in Fig. 3 to Fig. 6; they have variable names inscribed, and are connected with their attached nodes by lines and arrows. The order of the attached nodes will always be clear from the context.*

Program graphs have been proposed in [VJ03] to represent object-oriented programs. In the simplified version of [DHJ$^+$08], 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 hierarchical syntactical structure of entities, whereas wave-like arrows represent semantic relations between them.

We define rewriting of graphs as in [EEPT06], but represent rules in a slightly different way. We also insist on injective matching of rules. This is not a restriction, see [HMP01].

Definition 2 (Rewriting) A (*graph rewrite*) *rule* $\rho = (P \hookrightarrow B \hookleftarrow R)$ consists of a *body* graph B with subgraphs P and R, which are called *pattern* and *replacement*, respectively. Their common subgraph $I = P \cap R$ is called the *interface*. Nodes and edges in B are called *obsolete* if they lie in $P \setminus I$, and *new* if they lie in $R \setminus I$. We sometimes refer to graphs of ρ by P^ρ, B^ρ etc.

An injective morphism $m\colon P \to G$ is a *match* of ρ in a terminal graph G if it satisfies the following *gluing condition*: Every edge of G that is attached to the match of an obsolete node in $m(P \setminus I)$ lies in the match of the pattern $m(P)$. Then the rule ρ *rewrites* G under m into a graph H that can be constructed uniquely (up to isomorphism) by (1) removing the match $m(P \setminus I)$ of obsolete nodes and edges, giving the *kept graph* K, and (2) uniting K disjointly with a fresh copy of R, and gluing the interface $I \hookrightarrow R$ to its match $m(I)$ in K, giving H.[2] Then we write $G \Rightarrow_{\rho,m} H$; K and H can be constructed so that there are inclusions $G \hookleftarrow K \hookrightarrow H$.

In [EEPT06], a rule is represented as a "span" $(P \hookleftarrow I \hookrightarrow R)$. Our representation as a "join"

[1] A^* denotes *finite sequences* over a set A; the empty sequence is denoted by ε, and $|w|$ denotes the *length* of a sequence $w \in A^*$. For a function $f\colon A \to B$, its extension $f^*\colon A^* \to B^*$ to sequences is defined by $f^*(a_1 \dots a_n) = f(a_1) \dots f(a_n)$, for all $a_i \in A$, $1 \leqslant i \leqslant n$, $n \geqslant 0$. For functions or morphisms f and g, $f \circ g$ denotes their composition.

[2] Only the nodes and edges in the subgraph $(P \cup R) \hookrightarrow B$ are relevant for rewriting.

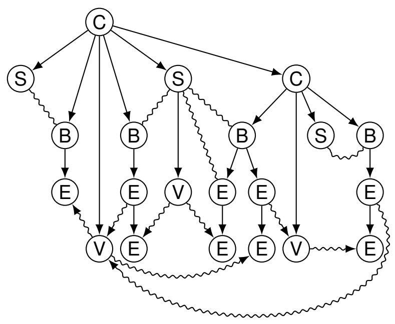


Figure 1: A program graph

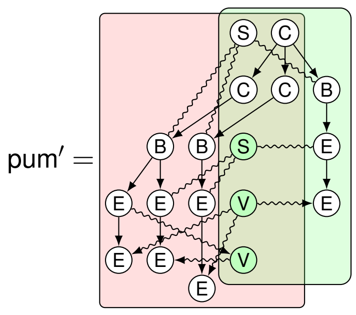


Figure 2: A rule performing Pull-up Method

$(P \hookrightarrow B \hookleftarrow R)$ is equivalent. However, the fact that the body B contains both pattern and replacement makes it easier to define the refinement of a rule (schema), which is a central concept of this work, see Def. 4 below.

Example 2 (A Refactoring Rule) *Figure 2 shows a rule pum′. Blobs painted red and green designate its pattern and replacement, respectively. Together they designate the body, and their overlap defines the interface. An edge belongs only to those blobs that contain it entirely; so the straight arrow connecting the top-most C-node in the interface to a B-node in the replacement belongs only to the replacement, but not to the pattern (and the interface) of pum′.*

The pattern of pum′ specifies a situation where two subclasses of a class contain obsolete bodies for the same signature. The replacement specifies that one of these bodies is new in the superclass. If the obsolete bodies are semantically equivalent (which cannot be checked automatically, in general, but has to be confirmed by verification), pum′ performs a Pull-up method refactoring. However, pum′ only applies if the class has exactly two subclasses, and if the method bodies have the particular shape seen in the pattern.

The general Pull-up Method *refactoring of Fowler [Fow99], which works for an unbounded number of subclasses and for method bodies of varying shapes, cannot be specified with a simple rule like this, but with schema refinement, see Example 7 below.*

Graph rewriting can be used for computing by *reduction*: a set of rules is applied to some input graph as long as possible. If no rule applies, a graph is a result (called normalform). Reduction is terminating if every sequence of steps leads to a normalform. The normalform for some graph will always be unique if the rewriting relation is confluent.

Rules can also be used to derive sets, or *languages*, of graphs. A simple form of rules, where the pattern is a variable with its attached nodes, is known as context-free (hyperedge) replacement [Hab92]. Here we extend these rules so that their pattern may contain isolated nodes. This increases their generative power considerably.

Definition 3 (Derivation) A rule $\pi = (P \hookrightarrow B \hookleftarrow R)$ is *contextual* if x is the only edge in P and R equals B without x.

For a set Π of contextual rules, we write $G \Rightarrow_\Pi H$, and say that Π *directly derives* G to H if there is a rewrite $G \Rightarrow_{\pi,m} H$ for some match m of some $\pi \in \Pi$. As usual, $\Rightarrow^+_\Pi$ denotes the

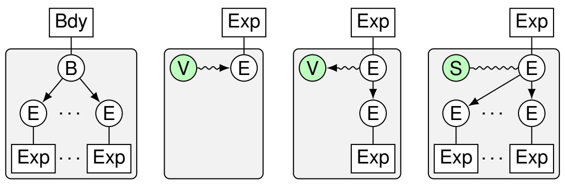


Figure 3: Contextual rules M for method bodies

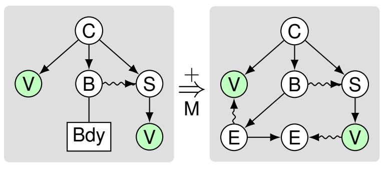


Figure 4: Deriving a method body

transitive closure of this relation. With some *start graph* G_0, Π defines a *contextual grammar* $(\mathscr{C},\Pi,G_0)$ over the labels $\mathscr{C}$, which derives the *language* of terminal graphs derived from G_0: $\mathscr{L}(\mathscr{C},\Pi,G_0) = \{H \mid G_0 \Rightarrow^{+}_{\Pi} H, X_H = \emptyset\}$.

The pattern P of a contextual rules π is the disjoint union of a star $\langle x \rangle$ with a discrete *context graph*, say C. We call π *context-free* if C is empty. The (context-free) *borrowing rule* $\bar{\pi} = (\langle x \rangle \hookrightarrow B \hookleftarrow R[C])$ of π is obtained by removing C from P, and distinguishing the subgraph $C \hookrightarrow R$ as a *debt* in R. $\bar{\Pi}$ is the set of *borrowing rules* for Π. We write derivations with $\bar{\Pi}$, which we call *endebted*, as $G \Rightarrow^{+}_{\bar{\Pi}} H[D]$ where D is the *debt*, i.e., the embeddings of the nodes borrowed in the rules used in the derivation.

Derivations always extend the kernels of graphs.

Fact 1. *(1) For every contextual rule $\pi = (P \hookrightarrow B \hookleftarrow R)$, $\underline{P} \hookrightarrow \underline{R}$.*
(2) For every derivation $G \Rightarrow_{\pi} H$, it holds that $\underline{G} \hookrightarrow \underline{H}$.

Proof. Let $x \in \vec{P}$. (1) Since $\underline{P} = I$ by definition, $I \hookrightarrow \underline{R}$. (2) Thus, by definition of $\Rightarrow_{\pi}$, $K = G - m(x)$. Then $\underline{K} = \underline{G}$, and $K \hookrightarrow H$ implies $\underline{K} \hookrightarrow \underline{H}$. □

Example 3 (Method Bodies) *When drawing contextual rules like those in Fig. 3, we omit the blob around the pattern, and paint that of the replacement in grey. The variable outside the replacement box designates the star of the pattern, and green filling (appearing grey in B/W print) designates the context nodes.*

Figure 3 defines the set M of contextual rules that derive method bodies of program graphs, or rather their data flow graphs: A body consists of expressions, which in turn either use the value of a variable, define it as the value of an expression, or call a method signature with actual parameters. The ellipses "..." in two of the rules abbreviate replicative (context-free) rules that derive $n \geqslant 0$ copies of Exp-stars. In the rules for Exp, the nodes representing variables and signatures are context nodes.

Figure 4 shows a derivation of a method body with M. Note that the body can only be derived from a graph that contains appropriate context nodes. For a start graph $G_0 = \langle x \rangle$ with $\ell_{G_0}(x) =$ Bdy, we have $\mathscr{L}(\mathscr{C}, M, G_0) = \emptyset$. The grammar for program graphs discussed in [HM10] does derive appropriate context nodes.

Note that the language of program graphs cannot be derived with context-free rules alone: in infinitely many program graphs, nodes labeled S, V, or B have an unbounded connectivity [Hab92, Theorem IV.3.3].

3 Schema Refinement with Contextual Meta-Rules

The idea to refine rules with meta-rules before applying them originated in two-level string grammars [CU77] that have been devised for defining the programming language ALGOL-68. The idea has been transferred to graphs – with the aim to derive graph languages – rather early [Hes79, Gö79]. In [PH96], Detlef Plump and Annegret Habel have used the same idea for graph rewriting. Their definition of a *substitutive rule* is inspired by term rewriting [Ter03]: the pattern and replacement may contain graph variables. Before applying the rule, every variable in the pattern is substituted with some graph; thus the rule may *match* a subgraph of unbounded size and arbitrary shape. If a variable does not occur in the replacement, the rule will just *delete* its substitution; otherwise the rule will *replicate* it as many times as the variable occurs in the replacement. In [PH96], a variable may be substituted with any graph. Then rule matching becomes highly non-deterministic, so that reduction may involve a lot of backtracking. This can be very inefficient. If the substitution of a variable is required to be "shaped", meaning that it is a graph derived with some set of contextual rules, matching gets feasible, as the rules guide the recursive search for a matching substitution. The context-free rules proposed for this purpose in [Hof01] turned out to be too restrictive in practice: method bodies, for instance, cannot be derived with context-free rules so that they could not be used to define the shape of graph variables. (In [DHJ$^+$08], adaptive star rules [DHJM10] have been proposed to derive substitutions. This is powerful enough, but adaptive star rules tend to be complicated.)

The graph rewriting tool GRGEN [BGJ06] supports rules that are even more expressive [Jak08, HJG08]:

- Substitutions of variables may be defined by rules that are contextual, not just context-free. Thus we may substitute variables with graphs that cannot be derived by context-free rules, e.g., with graphs of method bodies as defined in Fig. 3.
- A variable may be attached to both, pattern nodes and replacement nodes. Its substitution refines the pattern and the replacement of a rule at the same time. This does not only allow to match, delete, or replicate subgraphs of unbounded size and arbitrary shape: the rules that derive recursive sub-rules that *transform* such subgraphs in a single rule application.

This way of *rewriting with contextual refinement* shall be studied in this paper.

Example 4 (Pull-up Method Revisited) *In order to generalize the Pull-up method rule of Fig. 2 so that it applies to classes with a varying number of subclasses, and method bodies of varying shape, we turn a rule into a* schema *with variables. Variables are attached to nodes in the pattern and/or replacement of the schema. Figure 5 shows a schema pum for Pull-up Method. Two variables named Bdy^{10} and one variable named Bdy^{11} indicate spots of refinement. Contextual* meta-rules *are used to refine a schema. Their replacement is not just a graph, but a schema (called the embodied schema), i.e., a body graph with distinguished pattern and replacement, and with variables for further refinement if necessary. Figure 6 shows some meta-rules for the variables in pum. The meta-rules for Bdy^{10} and Exp^{10} refine the obsolete part of pum according to the contextual rules M of Fig. 3, whereas the meta-rules for Bdy^{11} and Exp^{11} replicate equal method bodies in the obsolete and new part of the schema. (The superscripts in the variable names indicate how many copies of a method body are made in its pattern and replacement.)*

pum =

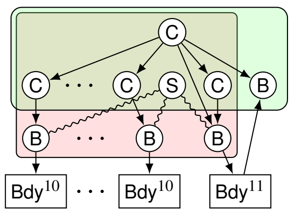


Figure 5: A schema for Pull-up method

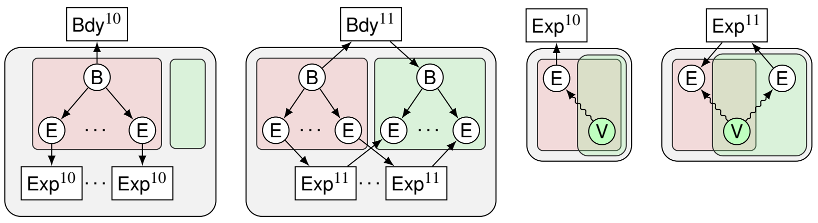


Figure 6: Meta-rules for Bdy^{10}, Bdy^{11}, Exp^{10}, and Exp^{11}

The meta-rules for the remaining contextual rules of M are defined analoguously. (In pum, the dots between the subgraphs containing the variable Bdy^{10} are a shortcut for any number of these subgraphs. These can be generated easily by auxuliary meta-rules.)

Refinement of a schema generates rules, called instances, wherein all variables are gone. Refining pum with the meta-rules in Fig. 6 can generate the rule pum′ in Fig. 2 as a particular instance. It may generate instances for any number of subclasses, and for method bodies of any shape. So it implements a general Pull-up method refactoring according to [Fow99].

Contextual meta-rules, like those for Exp^{10} and Exp^{11}, can only be applied to a schema that contains a node matching its context node (a V-node in this example). This is too restrictive: the context node could also be generated as an "ordinary" refinement of the rule in order to become an interface node that is matched when the instance is applied to a source graph. This can be achieved by using the borrowing version of every "truly" contextual meta-rule. In borrowing meta-rules for Exp^{10} and Exp^{11}, the V-nodes would not be context nodes. In rule pum′, three nodes (painted green) have been derived from pum with borrowing meta-rules for M.

We can now formally define schemata and meta-rules. In that context, variables must be used in a way that is consistent with this distribution: for all variables carrying the same name, corresponding attached nodes must belong to the same part of the schema: to the pattern, to the replacement, or to both.

Definition 4 (Schema, Meta-Rule) We assume that the variable names $\mathscr{X}$ come with two functions $pat, rep\colon \mathscr{X} \to \mathbb{N}$ so that $pat(\xi) \leqslant |type(\xi)|$ and $1 \leqslant rep(\xi) \leqslant pat(\xi)$ for every variable name $\xi \in \mathscr{X}$. For every variable x named ξ with $att_G(x) = v_1 \dots v_k$ in a graph G, these functions designate discrete subgraphs of G: the *pattern attachment* $\langle \underline{x}]$ consists of the leading attached nodes $v_1 \dots v_{pat(\xi)}$, and the *replacement attachment* $[\underline{x}\rangle$ consists of the trailing attached nodes

$v_{rep(\xi)} \ldots v_k$.[3]

A rule $\sigma = (P \hookrightarrow B \hookleftarrow R)$ is a *schema* if $P \cup R = \underline{B}$ and every variable $x \in X_B$ is *consistently distributed*, which means that $\langle \underline{x}] \hookrightarrow P$ and $[\underline{x}\rangle \hookrightarrow R$.

A *meta-rule* $\delta = (\pi, \varepsilon)$ has a contextual rule $\pi = (P^\pi \hookrightarrow B^\pi \hookleftarrow R^\pi)$ as a *spine*, with a schema $\varepsilon = (P^\varepsilon \hookrightarrow B^\pi \hookleftarrow R^\varepsilon)$, called the *embodied schema* of δ.

A match $m \colon P^\pi \to B$ is a *meta-match* of the spine rule π in the body B of σ if it maps the context C^π of π to the interface of σ, i.e., if $m(C^\pi) \hookrightarrow I$.

The following properties are direct consequences of this definition.

Proposition 1 *Consider a schema $\sigma = (P \hookrightarrow B \hookleftarrow R)$, a meta-rule $\delta = (\pi, \varepsilon)$ with a meta-match $m \colon P^\pi \to B$ so that there is a derivation $B \Rightarrow_{\pi,m} B'$.*

Then there are unique graphs P' and R' so that $\sigma' = (P' \hookrightarrow B' \hookleftarrow R')$ is a schema with $P \hookrightarrow P' \hookleftarrow m(P^\varepsilon)$ and $R \hookrightarrow R' \hookleftarrow m(R^\varepsilon)$.

Proof. The morphisms between components of a meta-rule δ in Fig. 7 are used to prove the claim. Actually, all of them are inclusions, but we draw them simply as arrows. The solid arrows exist by Def. 4. Since R^π equals B^π without the single variable x in P^π, the terminal graphs P^ε and R^ε are included in R^π, and $(P^\varepsilon \hookrightarrow R^\pi \hookleftarrow R^\varepsilon)$ is a schema as well. The intersections $P^{\varepsilon\pi} = P^\varepsilon \cap P^\pi$ and $R^{\varepsilon\pi} = P^\pi \cap R^\varepsilon$ are discrete, as P^π does not contain terminal edges. Let C^π be the context graph of π. Then $P^\pi = \langle x \rangle \uplus C^\pi$ and $I^\pi = \langle \underline{x} \rangle \uplus C^\pi$. Since variable attachments are consistently distributed, $P^{\varepsilon\pi} = \langle \underline{x}] \cup C^\pi$, $R^{\varepsilon\pi} = [\underline{x}\rangle \cup C^\pi$, and $(P^{\varepsilon\pi} \hookrightarrow P^\pi \hookleftarrow R^{\varepsilon\pi})$ is a schema as well. As $\underline{P^\pi} = I^\pi$, $(P^{\varepsilon\pi} \hookrightarrow I^\pi \hookleftarrow R^{\varepsilon\pi})$ is another schema. This gives the dashed morphisms in Fig. 7.

$$\begin{tikzcd}
 & & P^\pi \arrow[ddd] & \\
P^{\varepsilon\pi} \arrow[urr, dashed] \arrow[ddd, dashed] & & & R^{\varepsilon\pi} \arrow[ul, dashed] \arrow[ddd, dashed] \\
 & I^\pi \arrow[uur] \arrow[ul, dashed] \arrow[urr, dashed] \arrow[ddd] & & \\
 & & B^\pi & \\
P^\varepsilon \arrow[urr] \arrow[dr, dashed] & & & R^\varepsilon \arrow[ul] \arrow[dll, dashed] \\
 & R^\pi \arrow[uur] & &
\end{tikzcd}$$

Figure 7: A meta-rule

Thus the derivation $B \Rightarrow_{\pi,m} B'$ has the solid morphisms shown in Fig. 8, where $\bar{m}$ shall denote the *embedding* $R^\pi \to B'$ of m. Consistent distribution of variable attachments in B implies that $m(P^{\varepsilon\pi}) \hookrightarrow P$ and $m(R^{\varepsilon\pi}) \hookrightarrow R$. Since K equals B without $m(x)$, we get inclusions forming the schema $P \hookrightarrow K \hookleftarrow R$. Now P' is the unique subgraph of B', containing the subgraph P and the subgraph $\bar{m}(P^\varepsilon)$; R' is determined likewise. The resulting inclusions $\sigma' = (P' \hookrightarrow B' \hookleftarrow R')$ are a schema if $P' \cup R' = \underline{B'}$ and all variables in B' are consistently distributed. In R^π and K, all terminal edges are in the pattern or replacement subgraphs (or in both), and all variables are consistently distributed. So it is easy to see that so are the terminal edges and variables in B'. □

$$\begin{tikzcd}
 & P^\pi \arrow[rrr, "m"] & & & B & \\
 & & R^{\varepsilon\pi} \arrow[ul] \arrow[rrr, dashed] \arrow[dl] \arrow[dd] & & & R \arrow[ul] \arrow[dl, dashed] \arrow[dd, dashed] \\
 & I^\pi \arrow[u] \arrow[rrr] \arrow[ddd] & & & K \arrow[uu] \arrow[ddd] & \\
 & & R^\varepsilon \arrow[rrr, dashed] \arrow[ddl, dashed] & & & R' \arrow[ddl, dashed] \\
P^{\varepsilon\pi} \arrow[uuuur] \arrow[uur] \arrow[rrr, dashed] \arrow[dd] & & & P \arrow[uuuur] \arrow[uur, dashed] \arrow[dd, dashed] & & \\
 & R^\pi \arrow[rrr, "\bar{m}"'] & & & B' & \\
P^\varepsilon \arrow[ur] \arrow[rrr, dashed] & & & P' \arrow[ur, dashed] & &
\end{tikzcd}$$

Figure 8: Schema refinement

This proposition allows to define the refinement of schemata by meta-rules. Refinement proceeds until all variables have disappeared, and the resulting rule can be used to rewrite a graph.

[3] Thus the nodes $v_{rep(\xi)} \ldots v_{pat(\xi)}$ are in the interface. In figures like Fig. 5 and Fig. 9 , the lines between an attached node v and a variable x get an arrow tip at x if v is obsolete, at v if v is new, and none otherwise.

Definition 5 (Schema Refinement and Rewriting) Consider a schema $\sigma = (P \hookrightarrow B \hookleftarrow R)$ and a meta-rule $\delta = (\pi, \varepsilon)$ as above.

For the derivation $B \Rightarrow_{\pi,m} B'$, Fig. 1 allows to extend B' to a unique schema $\sigma' = (P' \to B' \leftarrow R')$. Then we write $\sigma \Downarrow_{\delta,m} \sigma'$, and say that δ *refines* σ to σ' (at m).

For a contextual meta-rule $\delta = (\pi, \varepsilon)$, the *borrowing meta-rule* is $\bar{\delta} = (\bar{\pi}, \varepsilon)$. Let Δ be a finite set of meta-rules so that $\delta \in \Delta$ implies $\bar{\delta} \in \Delta$. Then $\Downarrow_\Delta$ denotes refinement steps with one of its meta-rules, and $\Downarrow_\Delta^+$ denotes repeated refinement, its transitive closure. $\Delta(\sigma)$ denotes the *derivates* of a schema $\sigma = (P \hookrightarrow B \hookleftarrow R)$, containing its refinements without variables:

$$\Delta(\sigma) = \{\rho \mid \sigma \Downarrow_\Delta^+ \rho, X_{B^\rho} = \emptyset\}$$

We write $G \Rightarrow_{\Delta,\sigma} H$ and say that σ, refined with Δ, *rewrites* G to H if $G \Rightarrow_\rho H$ for some derivate $\rho \in \Delta(\sigma)$.

By lifting Proposition 1 to refinement sequences, we see that derivates do indeed refine a schemata.

Fact 2. *For derivates $\rho \in \Delta(\sigma)$, $P^\sigma \hookrightarrow P^\rho$, $\underline{B^\sigma} \hookrightarrow B^\rho$, and $R^\sigma \hookrightarrow R^\rho$.*

It can be checked whether schemata do have refinements, or not.

Theorem 1 *For a schema σ, it is decidable whether $\Delta(\sigma) = \emptyset$, or not.*

Proof. A rule ρ is in $\Delta(\sigma)$ if and only if its body B^ρ is in the language of the contextual grammar $(\mathscr{C}, \Pi, B^\sigma)$ of spine rules $\Pi = \{\pi \mid (\pi, \varepsilon) \in \Delta\}$ of Δ that has the body of σ as a start graph. Emptiness of the language is decidable for contextual grammars, by [DHM12, Corollary 1]. □

Example 5 (Encapsulate-Field) *Another refactoring operation,* Encapsulate Field, *shall replace using and defining accesses to an attribute (or field) a of a class by invocations of getter or setter methods, resepctively. Since a program graph may contain an unbounded number of accesses to a, the operation cannot be specified by a single rule.*

However, it can be defined by a schema with meta-rules. The pattern of the schema ef *in Fig. 9 consists of a class node connected to a variable node. To this pattern, the replacement adds graphs representing signatures and bodies of the setter and getter methods. The variables* uses *and* defs *indicate points of refinement.*

Their meta-rules are context-free, and have a similar structure: One using (defining) access of some expression to a in the pattern is replaced by a call of the getter (setter) method in the replacement, and the meta-rule is called recursively. The other meta-rules terminate this iteration.

A single rewriting step with some derivate of ef *transforms an arbitrary number of using and defining accesses to a variable into calls to the getter and setter method, respectively. This cannot be achieved by a plain rewrite rule; it goes even beyond the expressiveness of the substitutive rules that will be considered in Section 4.2 further below.*

Note that the meta-rules derive rules that encapsulate an arbitrary subset of variable accesses, and leaves the remaining ones direct. In practice, an access to v shall be encapsulated if and only if it is non-local; this could be specified by extending meta-rules with application conditions, as proposed in [HM10].

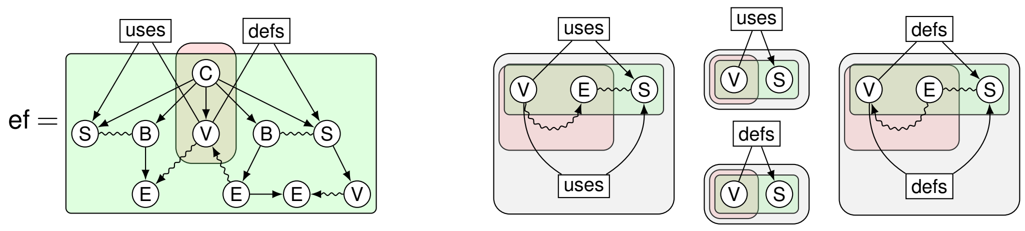


Figure 9: A schema for *Encapsulate-Field* with four meta-rules

Note that the application of a derivate $\rho \in \Delta(\sigma)$, although it is the result of a compound meta-derivation, is a single rewriting step $G \Rightarrow_\rho H$, like a transaction in a data base. Note also that the refinement process is completely rule-based.

Operationally, we cannot construct the derivates of a schema σ first, and apply one of them later, because $\Delta(\sigma)$ is infinite in general. Rather, we interleave matching and refinement in a goal-oriented algorithm.

Algorithm 1 (Applying a Schema to a Graph) *Input*: A terminal graph G, a schema $\sigma_0 = (P^0 \hookrightarrow B^0 \hookleftarrow R^0)$, and a set Δ of meta-rules.

Output: Either a refinement $\sigma_i \in \Delta(\sigma_0)$ with a match $m_i\colon P^i \to G$, or the answer "*no*" if no derivate in $\Delta(\sigma_0)$ applies to G.

1. Search for an injective morphism $m_0\colon P^0 \to G$ of the original schema σ_0, and set i to 1.
2. Search for a context-free meta-rule $\delta_i \in \Delta$ with a refinement $(P^{i-1} \hookrightarrow B^{i-1} \hookleftarrow R^{i-1}) \Downarrow^+_{\delta_i} (P^i \hookrightarrow B^i \hookleftarrow R^i)$ such that the morphism m_{i-1} can be extended to a morphism $m_i\colon P^i \to G$. If δ_i is the borrowing rule of some contextual meta-rule $\delta' = (\pi', \varepsilon')$, m may identify context nodes of the spine rule π' with nodes in the interface $P^{i-1} \cap R^{i-1}$; otherwise it must be injective.
3. If no such meta-rule can be found, undo refinements to the closest step $j < i$ where some meta-rule exists that has not been inspected for σ_j previously; if already all meta-rules have been inspected, for all previous steps, and for all initial matches m_0, answer "*no*".
4. If σ_i does contain further variables, increase i by one and goto Step 2.
5. Otherwise, σ_i is a rule with a morphism $m_i\colon P^i \to G$. If m_i violates the gluing condition, goto Step 3. Otherwise, output m_i as a match of the derivate σ_i in G.

Actually, *all* matches for *all* applicable derivates can be enumerated, so that potential nondeterminism in the refinement can be handled by backtracking.

GRGEN rules are executed this way. But, the match of a particular pattern P^i is determined according to an efficient search plan that is pre-computed, taking into account the shape of the type graph, as well as that of the particular host graph G, and the whole rule is compiled into sequences of low level C# instructions that do graph matching and construction in an efficient way [Gei08].

We need a mild condition to show that the algorithm terminates.

Definition 6 (Pattern Loops in Meta-Rules) A set Δ of meta-rules *does not loop on patterns* if $\sigma \Downarrow_{\Delta}^{+} \sigma'$ implies $P^{\sigma} \not\cong R^{\sigma'}$.

Note that the meta-rules in Fig. 9 of Example 5 do not loop on patterns, nor do those in Fig. 10 and Fig. 11 of Def. 7, nor those in Fig. 6 of Example 7.

Theorem 2 *Algorithm 1 is correct, and terminates if Δ does not loop on patterns.*

Proof Sketch. In its course, the algorithm constructs a tree of refinements. This tree is finite: It has finite breadth, as Step 1 chooses among finitely many initial morphisms m_0 in G, and Step 2 chooses among a finite number of morphisms for a finite number of meta-rules. Its depth is finite as well, because every δ_i in Step 2 satisfies $|\underline{P^{i-1}}| \leqslant |\underline{P^{i}}|$, and the length of chains with $|\underline{P^{i-1}}| = |\underline{P^{i}}|$ is bound by $|\Delta|$ since Δ does not loop on patterns.

For correctness, consider the following arguments: The identification condition in Step 2 makes sure that the morphism m_i either is an injective morphism for the borrowing meta-rule $\bar{\delta}_i$, or an injective morphism for the original meta-rule δ_i. Thus Step 5 determines an injective morphism of a derivate of σ_0, and checks whether it is a match. The backtracking in Step 3 and Step 5 makes sure that every possible candidate for a derivate with a match is considered. $\square$

Corollary 1 *For a schema σ and meta-rules Δ as above, it is decidable whether some derivate $\rho \in \Delta(\sigma)$ applies to a graph G, or not.*

4 Relation to Other Ways of Graph Rewriting

In this section, we compare schemata to other ways of graph rewriting known in the literature: graph rewriting by single pushout, by neighborhood-controlled embedding, and by substitutive rules. In all three cases, a rule can be modeled by a single schema with appropriate meta-rules, whereas they cannot be modeled by a single plain rule.

4.1 Graph Rewriting with Node [Dis-]Connection

We investigate ways of graph rewriting where a single rule may delete and redirect a variable number of edges, respectively. First we define generic meta-rules that allow to model such rules.

Definition 7 ([Dis-]Connecting Meta-Rules) Figure 10 shows a *disconnecting* meta-rule δ_d^+ that extends a schema by a B-node in its interface, which is connected by an obsolete a-edge to an obsolete A-node. A *disconnecting configuration* $d \in \dot{\mathscr{C}} \times \vec{\mathscr{C}} \times \{\text{in}, \text{out}\} \times \dot{\mathscr{C}}$ specifies the labels of the refined nodes and the label and direction of the refined edge. Its borrowing meta-rule $\bar{\delta}_d^+$ is not shown. The (context-free) meta-rule δ_d^0 terminates disconnection. A so refined schema will remove edges that connect nodes in the interface to the obsolete node that is attached to the variable $\mathsf{disconnect}_d$. A set of cooperating disconnecting meta-rules for all possible disconnecting configurations will remove all edges to the indicated obsolete nodes. It is important that the B-node is contextual: only then several edges connected to such a node can be disconnected.

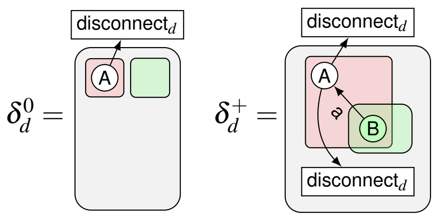


Figure 10: Meta-rules for a disconnecting configuration $d = (\mathsf{A}, \mathsf{a}, \mathrm{in}, \mathsf{B})$

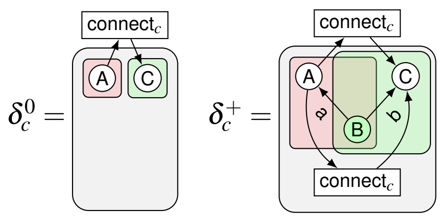


Figure 11: Meta-rules for a connecting configuration $c = (\mathsf{A}, \mathsf{a}, \mathrm{in}, \mathsf{B}, \mathsf{b}, \mathrm{in}, \mathsf{C})$

Figure 11 shows a *connecting* meta-rule δ_c^+ that extends a schema by nodes in its interface, which are connected to an obsolete node and to a new node. A *connecting configuration* $c \in \dot{\mathscr{C}} \times \vec{\mathscr{C}} \times \{\mathrm{in}, \mathrm{out}\} \times \dot{\mathscr{C}} \times \vec{\mathscr{C}} \times \{\mathrm{in}, \mathrm{out}\} \times \dot{\mathscr{C}}$ specifies the labels of the refined nodes and the labels and directions of the refined edges. Again, the borrowing meta-rule $\bar{\delta}_c^+$ is not shown. The context-free meta-rule δ_c^0 terminates disconnection. A so refined schema will remove, for the obsolete node v and the new node v' attached to the variable, edges connecting v to interface nodes, and insert edges connecting these nodes to v' instead.[4] Again, it is important that the B-nodes are contextual in order to redirect several edges incident with a node.

Δ^{dc} shall denote the set of all disconnecting and connecting meta-rules, for all disconnecting and connecting configurations over the label alphabet $\mathscr{C}$.

Graph rewriting defined with single pushouts [Löw93] (SPO, for short) is a variation of standard graph rewriting [EEPT06] as in Def. 2. Apart from technical details of their definition, SPO rules have the same form as DPO rules. However, rewriting is defined differently: no dangling condition needs to be checked when matching a rule; the match $m(v)$ of every obsolete node v of τ is always deleted, together with all its incident edges, including those outside the match $m(P)$ that would violate the dangling condition in the standard case.

We define how a plain rule according to Def. 2 is turned into a schema that performs SPO graph rewriting.

Definition 8 (Modeling SPO Rewriting with Schema Refinement) For a rule $\rho = (P \hookrightarrow B \hookleftarrow R)$, construct the *SPO schema* $\sigma_\rho = (P \hookrightarrow B \hookleftarrow R)$ by attaching, to every obsolete node v in $P \setminus R$, variables x named $\mathsf{disconnect}_d$, for every disconnecting configuration $d = (\dot{\ell}_P(v), a, io, A)$ where $a \in \vec{\mathscr{C}}$, $io \in \{\mathrm{in}, \mathrm{out}\}$, and $A \in \dot{\mathscr{C}}$.

Fact 3. *Consider a rule ρ with an injective morphism $m\colon P \to G$.*

Then the SPO schema σ_ρ has a derivate $\rho' \in \Delta^{\mathrm{dc}}(\sigma_\rho)$ with a rewriting step $G \Rightarrow_{\rho'} H$ so that all obsolete nodes of π are deleted, together with their incident ("dangling") edges in the subgraph $G \setminus m(P)$.

Proof Sketch. Consider $m\colon P \to G$. Then $P \hookrightarrow P'$ by Proposition 1. Now, at every obsolete node v of ρ, the schema can be refined with disconnecting meta-rules so that it covers every "dangling"

[4] The meta-rules for uses and defs in Example 5 are similar to connecting rules; only the node being disconnected, which is labeled V, is not obsolete, but in the interface.

edge e incident in $\dot{m}(v)$ and its "neighborhood node" – say v' – that is outside $m(P)$. As the pre-image of v' in the schema is in the interface of the refined schema, and the edge is in the obsolete part of the pattern, the dangling edges are removed while the nodes in the neighborhood are preserved. The refined rule satisfies the gluing condition, and does exactly what SPO rewriting does. [5] □

Graph grammars with neighborhood-controlled embedding are "the other notion" of graph rewriting. Their definition is set-theoretic, and their most widely investigated subclass has rules where the pattern consists of a single node [ER97]. Here, we consider the more general case where the pattern is a graph. More precisely, *neighborhood-controlled embedding rewriting* (NCE rewriting, for short) is defined as follows. [6]

Definition 9 (NCE Graph Rewriting) An *NCE rule* $\eta = (M, D, I)$ consists of terminal graphs M and D, called *mother graph* and *daughter graph* respectively, and of a set of *connecting instructions*

$$I \subseteq \dot{M} \times \vec{\mathscr{C}} \times \{\text{in}, \text{out}\} \times \dot{\mathscr{C}} \times \vec{\mathscr{C}} \times \{\text{in}, \text{out}\} \times \dot{D}.$$

Let $\tilde{I}$ denote the tuples $(v, \mathsf{a}, io, \mathsf{A}) \in \dot{M} \times \vec{\mathscr{C}} \times \{\text{in}, \text{out}\} \times \dot{\mathscr{C}}$ without a connecting instruction $c = (v, \mathsf{a}, io, \mathsf{A}, \mathsf{b}, io', v') \in I$ (for some $(\mathsf{b}, io', v') \in \vec{\mathscr{C}} \times \{\text{in}, \text{out}\} \times \dot{D}$).

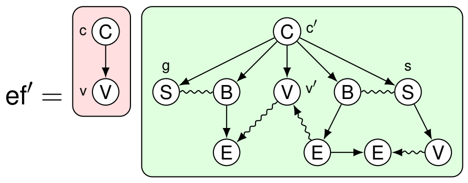
A rule η as above *applies* to a graph G if there is an injective morphism $m\colon M \to G$ so that no edge in $G \setminus m(M)$ is incident with a node in $m(M)$. Then the application of η at m deletes the subgraph $m(M)$ from G, adds a fresh copy of the daughter graph D, and obtains the graph H by executing the connecting instructions as follows:

- If $c = (v, \mathsf{a}, io, \mathsf{A}, \mathsf{b}, io', v') \in I$, for all A-nodes z in the subgraph $G \setminus m(M)$ where G contains an a-edge e connecting z with $\dot{m}(v)$ in direction io, a b-edge e connecting z with the node $v' \in \dot{D}$ in direction io' is inserted in H.
- For every $d = (v, \mathsf{a}, io, \mathsf{A}) \in \tilde{I}$, a-edges between v and A-nodes in direction io are deleted from G.

Example 6 (An NCE Rule for Encapsulate-Field) *The NCE rule ef′ in Fig. 12 defines a variant of the* Encapsulate Field *operation ef shown in Fig. 9. The mother graph is drawn left of the daughter graph. As the nodes c and v of the mother graph are obsolete, we have to add new nodes c′ and v′ with the same label to the daughter graph. The connecting instructions $I_{\mathsf{ef}'}$ are as follows:*

- *All wave-like arrows from the node v to E-nodes, which represent direct using accesses to v, are redirected to the getter method g. (This corresponds to the meta-rules for uses in Fig. 9.)*
- *All wave-like arrows from E-nodes to the node v, which represent direct defining accesses to v, are "turned around" and redirected to the setter method s. (This corresponds to the meta-rules for defs in Fig. 9.)*

[5] We have ignored the fact that matches of SPO rules need not be injective. However, we can construct the (finite) set the *quotients* $Q(\rho)$ by identifying arbitrary isomorphic subgraphs in P. Then for every non-injective match of ρ, $Q(\rho)$ contains a quotient with an equivalent injective match.
[6] The abbreviation "edNCE" used in [ER97] emphasizes the fact that the directed graphs being rewritten have labeled edges.



$$
\begin{array}{rcl}
I_{\mathsf{ef}'} & = & \{(\mathsf{v},\rightsquigarrow,\mathrm{out},\mathsf{E},\rightsquigarrow,\mathrm{out},\mathsf{g}')\} \\
& \cup & \{(\mathsf{v},\rightsquigarrow,\mathrm{in},\mathsf{E},\rightsquigarrow,\mathrm{out},\mathsf{s}')\} \\
& \cup & \left\{ (\mathsf{c},a,d,\ell,a,\mathsf{c}') \;\middle|\; \begin{array}{l} a \in \vec{\mathscr{C}}, \\ d \in \{\mathrm{in},\mathrm{out}\}, \\ \ell \in \dot{\mathscr{C}} \end{array} \right\}
\end{array}
$$

Figure 12: An NCE rule for *Encapsulate-Field*

- *All edges incident with the node* c *are redirected to the node* c′ *without changing their label or direction. (These instructions are overhead caused by the fact that the mother graph is completely obsolete.)*

Rule ef′ *will replace all direct accesses to the variable* v *– local accesses cannot be preserved in this framework.*

Definition 10 (Modeling NCE Rewriting with Schema Refinement) An NCE rule $\eta = (M, D, I)$ as above can be transformed into a schema $\sigma_\eta = (M \to B \leftarrow D)$ where the underlying rule $\underline{\sigma_\eta}$ has an empty interface (i.e., $\underline{B} = M \uplus D$), and B contains, for every connecting instruction $c = (v, \mathsf{a}, io, \mathsf{A}, \mathsf{b}, io', v') \in I$, a variable x named $\mathsf{connect}_{c'}$ with $att_B(x) = vv'$ and $c' = (\dot{\ell}_M(v), \mathsf{a}, io, \mathsf{A}, \mathsf{b}, io', \dot{\ell}_D(v'))$, and for every $d = (v, \mathsf{a}, io, \mathsf{A}) \in \tilde{I}$, a variable x named $\mathsf{disconnect}_{d'}$ with $att_B(x) = v$, and $d' = (\dot{\ell}_M(v), \mathsf{a}, io, \mathsf{A})$.

Fact 4. *For an NCE rule η as above, there exists an NCE graph rewriting step $G \Rightarrow_\eta H$ if and only if there is a rewriting step $G \Rightarrow_\rho H$ using some derivate $\rho \in \Delta^{\mathrm{dc}}(\sigma_\eta)$ of its schema σ_η.*

Proof Sketch. As in the case of modeling SPO rewriting, a match $m\colon M_\eta \to G$ can be taken as a basis to refine σ_η so that all connection instructions are "performed", and all other "dangling" edges are deleted.

There are other derivates that do not satisfy the dangling condition although the NCE rule applies. This is the case when the terminating meta-rules are applied before the recursive meta-rules have been applied exhaustively. If this is done, the NCE rule applies iff the schema applies. Even then, a refinement of the schema may violate the dangling condition. This is the case exactly when the match of the refined pattern contains nodes that are connected by edges outside the match. This, however, violates the application condition for NCE rules as well, where the match must be the induced subgraph of the matched nodes. □

4.2 Substitutive Graph Rewriting

We recall the definition of substitutive rules [Hof01], re-phrase it slighty according to our definition of plain rules, and extend it to borrowing substitutions:

Definition 11 (Substitutive Rule) Consider a finite set Π of contextual rules.

A *substitutive rule* $\gamma = (P \hookrightarrow B \hookleftarrow R)$ consists of two graphs P, R with variables so that

(1) all variables in B belong to P or to R, but not to both, and
(2) every variable name occurring in R occurs in P as well.

A *substitution* σ is a mapping from variable names to borrowing rules so that, for every variable name $\xi \in \mathscr{X}$, $\sigma(\xi) = (\langle\xi\rangle \hookrightarrow B \hookleftarrow R[D])$ with $\mathscr{X}(R) = \emptyset$.

A rule $\rho = (P' \hookrightarrow B' \hookleftarrow R')$ is a *σ-instance* of γ if its body B' is obtained from the (unique) terminal Graph B^* with $B \Rightarrow_\sigma^+ B^*$ by identifying the debts of all variables carrying the same names, and by including them both in P' and in R'. By identifying arbitrary nodes in $P \cap R$ with the debt, one obtains further σ-instances of γ.

A substitutive rewrite step $G \Rightarrow_\rho H$ applies some σ-instance ρ of γ, for some substitution σ.

Substitutive rules can be modeled as schemata refined by meta-rules as follows.

Definition 12 (Meta-Rules and Schemata for Substitutive Rules) Let γ be a substitutive rule for contextual rules Π.

Then $r = (k,m)$ is a *replication instruction* of a variable name $\xi \in \mathscr{X}$ if:

(a) ξ labels, in $\gamma = (P \hookrightarrow B \hookleftarrow R)$, k variables in P and m variables in R, or
(b) (inductively) Π contains a rule $(P' \hookrightarrow B' \hookleftarrow R')$ where the variable name ξ occurs in R', and the variable name ξ' in P has a replication instruction r.

We transform γ and Π into a schema σ_γ and meta-rules $\Delta_{\gamma,\Pi}$ as follows:

1. Whenever a variable ξ with $type(\xi) = l_1 \dots l_n$ has a replication instruction $r = (k,m)$, $\mathscr{X}$ shall contain a fresh *replication variable name* ξ^r with $type(\xi^r) = l_1^{k+m} \dots l_n^{k+m}$ and $pat(\xi^r) = k \cdot pat(\xi)$, $rep(\xi^r) = m \cdot rep(\xi)$.
2. We transform every substitutive rule $\gamma = (P \hookrightarrow B \hookleftarrow R)$ into a schema $\sigma_\gamma = (\underline{P} \hookrightarrow B' \hookleftarrow \underline{R})$ by replacing all variables $x_1, \dots, x_{k+m}$ named ξ in B (where we assume that x_i is in P iff $1 \leqslant i \leqslant k$) by a *replication variable* y named ξ^r with $att_{B'}(y) = \mathsf{pcat}\,(att_B(x_1), \dots, att_B(x_{k+m}))$.[7]
3. For every rule $\pi = (P \hookrightarrow B \hookleftarrow R) \in \Pi$ with $R = B - x$ and x named $\xi \in P$, and for every replication instruction $r = (k,m)$ of ξ, $\Delta_{\gamma,\Pi}$ shall contain a context-free *reproducing meta-rule* $\delta_\pi^r = (\pi^r, \varepsilon)$ so that
   (a) the kernel $\underline{B^{\pi^r}}$ of its spine rule consists of $k+m$ disjoint copies $B^1, \dots, B^{k+m}$ of $\underline{B}$ so that $B^1, \dots, B^k$ are in its embodied pattern P^ε and $B^{k+1}, \dots, B^{k+m}$ are in its embodied replacement R^ε;
   (b) for every variable x named ξ in B, where we assume that w_i is the copy of the attached node sequence $att_B(x)$ in B^i (for $1 \leqslant i \leqslant k+m$), B^{π^r} contains a variable y named ξ^r with $att_{B^{\pi^r}}(y) = \mathsf{pcat}\,(w_1, \dots, w_{k+m})$.[7]

   Note that the replication rule $\delta_{\bar\pi}^r$ of the borrowing rule $\bar\pi$ of a contextual rule π equals the borrowing rule of δ_π^r so that $\Delta_{\gamma,\Pi}$ does also contain all borrowing meta-rules.
   Thus $\langle y\rangle \hookrightarrow P^{\pi^r}$ if x is the variable in P, and all other variables belong to R^{π^r}. In B^{π^r}, a copy B^i belongs to P^ε if $1 \leqslant i \leqslant k$, and to R^ε if $k+1 < i \leqslant k+m$. If δ is contextual, its contextual nodes belong both to P^ε and to R^ε, are shared by all copies, and are contextual nodes in π^r.

[7] If, for $1 \leqslant i \leqslant k$, the words $w_i = a_{i,1} \dots a_{i,n}$ have equal length $n \geqslant 0$, their *pointwise concatenation* is $\mathsf{pcat}\,(w_1, \dots, w_k) = a_{1,1} \dots a_{k,1} \dots a_{1,n} \dots a_{k,n}$.

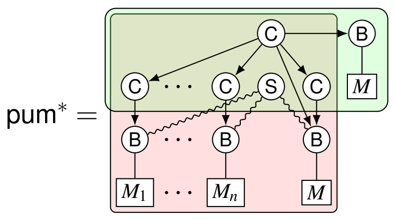


Figure 13: Pull-up method: substitutive rule

Note that the replications of meta-rules do not loop on patterns if the original meta-rules do not have looping derivations $P \Rightarrow^+_\Pi P$.

Example 7 (Pull-Up Method) *Figure 13 shows a substitutive rule pum* for Pull-up method. All variables have the shape Bdy defined by the contextual rules M in Fig. 3. The substitutions of $M_0,\dots,M_n,M$ are obsolete, but the variable named M is also new so that its substitution is re-inserted in the replacement.*[8] *The rule pum′ in Fig. 2 is an instance of pum*.*

The schema of pum, and the meta-rules for M equal those in Fig. 5* and *Fig. 6, respectively.*

Fact 5. *A rule ρ is the instance of a substitutive rule γ if and only if it is a derivate of the schema ρ of γ.*

5 Existence of Critical Overlaps

In this section, we study whether the existence of critical overlaps, a decidable property for plain rewriting, is decidable for schemata as well.

Definition 13 (Critical Overlap) Two rules $\rho = (P \hookleftarrow B \hookrightarrow R)$ and $\rho' = (P' \hookleftarrow B' \hookrightarrow R')$ *overlap critically* if there exists a graph G with matches $m \colon P \to G$ and $m' \colon P' \to G$ that intersect in deleted nodes or edges, i.e., if $m(P) \cup m'(P') \not\to m(P \cap R) \cup m'(P' \cap R')$.

An overlap is *minimal* if every node and edge of G is in the image either of P or of P'.

It suffices to check whether the finite number of minimal overlaps are parallel independent in the sense of [EEPT06, Def. 5.9] or not. In the case of schema refinement, the more interesting question is whether two schemata may have derivates that overlap critically, or not. Since schemata have infinitely many derivates in general, this question is harder to answer. Unfortunately, we obtain:

Theorem 3 *For rule schemata σ_1 and σ_2 with meta-rules Δ, it is, in general, undecidable whether or not σ_1 and σ_2 have derivates $\rho_1 \in \Delta(\sigma_1)$ and $\rho_2 \in \Delta(\sigma_2)$ with critical overlaps.*

Proof. (By contradiction.) We reduce this problem to deciding whether context-free languages are disjoint, which is known to be undecidable [HU79, Thm. 14.3].

[8] It is not forseen to have variables like M in the interface of a rule.

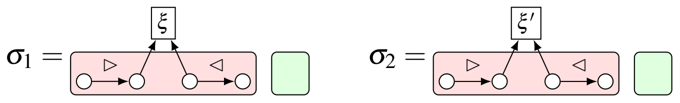
We define a representation of words as string graphs, and context-free rules as context-free rules over string graphs.

(1) Let $\dot{\mathscr{C}} = \{\circ\}$ and let all variable names $\xi \in \mathscr{X}$ have arity $arity(\xi) = \circ\circ$, $pat(\xi) = 2$, and $rep(\xi) = 0$.

(2) A graph G with nodes $\dot{G} = \{v_o, v_1, \ldots, v_n\}$, edges $\vec{G} = \{e_o, e_1, \ldots, e_n\}$ where $att_G(e_i) = v_{i-1}v_i$ for $1 \leqslant i \leqslant n$ is a *string graph* representing the word $w = \vec{\ell}^*_G(e_1 \ldots e_n) \in \vec{\mathscr{C}}^*$. The string graph representing a word w is unique up to isomorphism, and denoted by $w^\bullet$.

(3) Now a proper context-free rule $r = (\xi, w) \in \mathscr{X} \times \vec{\mathscr{C}}^+$ can be represented as a meta-rule δ_r with a spine rule $\pi = (P^\pi \to B^\pi \hookleftarrow R^\pi)$, where $P = \xi^\bullet$, $R = w^\bullet$, and, in B, P and R are disjoint up to their start and end nodes. It is shown in [Hab92] that these rules perform context-free derivations on string graphs. In the embodied schema $\varepsilon = (P^\varepsilon \to B^\varepsilon \hookleftarrow R^\varepsilon)$ of δ, $P^\varepsilon = B^\pi$ and R^ε is empty.

Now consider schemata where the string graphs for ξ and ξ' are enclosed in edges labeled with triangles (as start and end markers):



Then derivates $\rho_1 = (P_1' \to B_1' \hookleftarrow R_1') \in \Delta(\sigma_1)$ and $\rho_2 = (P_2' \to B_2' \hookleftarrow R_2') \in \Delta(\sigma_2)$ have critical overlaps if $P_1' \cong P_2'$. By the definition of derivates, and meta-rules, this implies that $P_1' \cong P_2' \cong w^\bullet$ for some word $w \in (\vec{\mathscr{C}} \setminus \mathscr{X})^*$, i.e., if w is in the context-free string languages derived from ξ and from ξ', respectively. In other words, σ_1 and σ_2 have critical overlaps if the languages derived from ξ and ξ' have an empty intersection. (There is no loss of generality if we assume that the context-free rules are proper, i.e., have non-empty right-hand sides, since every context-free grammar can be transformed into one where only the start symbol has a rule with an empty right-hand side. Then the question is still undecidable for the sublanguages without the empty word.) □

This result holds even if the meta-rules are context-free.

6 Conclusions

In this paper we have defined how schemata of plain graph rewriting rules can be refined with contextual meta-rules. This models the outstanding feature of GRGEN. Until now, we have not considered attributed graphs and subtyping. As they are included in the foundation [EEPT06], we expect that this can be added in a straight-forward way. We also restricted ourselves to unconditional rules. Rules with nested application conditions have been added to the theory in [EHL+10, EGH+12]; recently, Hendrik Radke has studied recursive refinement of such conditions [HR10]. We plan to add this concept to our definition in the future.

We have shown that several other kinds of graph rewrite rules can be defined with contextual refinement: rules based on single pushouts, neighborhood-controlled embedding, and variable substitution. Lack of space hindered us to present the definition of other kinds of rules: Adaptive star replacement [DHJM10], a combination of context-free rules with NCE rules, can be modeled along the lines of NCE graph rewriting; rules with variables substituted by adaptive star replacement [DHJ+08] can be modeled along the lines of the substitutive rules considered in Def. 11. The rules of [PH96] where variables can be substituted with arbitrary graphs are a special case of the substitutive rules considered here, as the language of all graphs can be defined with contextual replacement [DHM12, Ex. 1].

Even if graph rewriting with refinement is not to far from plain rewriting, some properties get lost since schemata have infinitely many derivates. So it is not decidable whether schemata may have parallelly (in-) dependent derivates, or not. An advanced property like confluence (based on the joinability of critical pairs) can probably only be considered in restricted situations. This will be the major direction of our future work. We want to provide assistance for users of GRGEN in analyzing the behavior of their specifications.

Acknowledgements: I wish to thank several people: Edgar Jakumeit conceived and implemented recursive refinements for rules in the GRGEN rewriting tool, under the supervision of Rubino Geiß. Katharina Saemann and Ruth Schönbacher identified an obvious error in Fig. 8, and Giorgio Busatto, Annegret Habel, and Hendrik Radke have lent their ears (and brains) to simplify and consolidate early drafts of this paper. Frank Drewes has carefully reviewed two later versions of the paper.

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 V3.6ms1b (2012).

[CU77] C. Cleaveland, R. Uzgalis. *Grammars for Programming Languages*. Elsevier, New York, 1977.

[DHJ+08] F. Drewes, B. Hoffmann, D. Janssens, M. Minas, N. Van Eetvelde. Shaped Generic Graph Transformation. In Schürr et al. (eds.), *Applications of Graph Transformation with Industrial Relevance (AGTIVE'07)*. Lecture Notes in Computer Science 5088, pp. 201–216. Springer, 2008.

[DHJM10] F. Drewes, B. Hoffmann, D. Janssens, M. Minas. Adaptive Star Grammars and Their Languages. *Theoretical Computer Science* 411:3090–3109, 2010.

[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.

[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.

[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. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*. Chapter 1, pp. 1–94. World Scientific, Singapore, 1997.

[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.

[Gei08] R. Geiß. *Graphersetzung mit Anwendungen im Übersetzerbau (in German)*. Dissertation, Universität Karlsruhe, 2008.

[Gö79] H. Göttler. Semantical Descriptions by Two-Level Gaph-Grammars for Quasi-hierarchical Graphs. In Nagl and Schneider (eds.), *Graphs, Data Structures, Algorithms (WG'79)*. Applied Computer Science 13, pp. 207–225. Carl-Hanser Verlag, München-Wien, 1979.

[Hab92] A. Habel. *Hyperedge Replacement: Grammars and Languages*. Lecture Notes in Computer Science 643. Springer, 1992.

[Hes79] W. Hesse. Two-Level Graph Grammars. In Claus et al. (eds.), *Graph Grammars and Their Application to Computer Science and Biology*. Lecture Notes in Computer Science 73, pp. 255–269. Springer, 1979.

[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.

[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).

[HU79] J. E. Hopcroft, J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

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

[Löw93] M. Löwe. Algebraic Approach to Single-Pushout Graph Transformation. *Theoretical Computer Science* 109:181–224, 1993.

[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.

[Ter03] Terese (ed.). *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science 55. Cambridge University Press, Cambridge, UK, 2003.

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