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.

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 is a natural formalism for specifying such systems in an abstract way, ever since this branch of theoretical computer science emerged in the seventies of the last century [8]. Graph rewriting has a well developed theory [4] 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 [9] allows to determine confluence of a set of rules by the analysis of finitely many critical pairs [17], and GROOVE [18] allows to explore the state space of specifications.

This work relates to GRGEN, an efficient graph rewrite generator [1]. Edgar Jakumeit has drastically extended the rules of this tool, by introducing recursive refinement for sub-rules and application conditions [15]. 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 [4] 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 [3]. 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 [14] 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 (“residual”) rewriting strategy, and show the correctness of this translation.

The examples shown 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 models). Many of the basic refactoring operations proposed by Fowler [10] 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 contextual 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 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 [11].

Definition 2.1 (Graph). Let $\Sigma = (\hat{\Sigma}, \check{\Sigma})$ be a pair of finite label sets.

A graph $G = (G, G, \text{att}, \ell)$ consists of two disjoint finite sets $G$ of nodes and $G$ of edges, a function $\text{att}: G \rightarrow G^*$ that attaches sequences of nodes to edges, and of a pair $\ell = (\ell, \check{\ell})$ of labeling functions $\ell: G \rightarrow \hat{\Sigma}$ for nodes and $\check{\ell}: G \rightarrow \check{\Sigma}$ for edges.\(^1\) We will often refer to the component functions of a graph $G$ by $\text{att}_G$ and $\ell_G$.

\(^1\) $A^*$ denotes finite sequences over a set $A$; the empty sequence is denoted by $\varepsilon$. 

---

1. A\(^*\) denotes finite sequences over a set $A$; the empty sequence is denoted by $\varepsilon$. 

---
More on Graph Rewriting With Contextual Refinement

A (graph) morphism $m: G \rightarrow H$ is a pair $m = (\hat{m}, \bar{m})$ of functions $\hat{m}: \hat{G} \rightarrow \hat{H}$ and $\bar{m}: \bar{G} \rightarrow \bar{H}$ that preserve attachments and labels: $\text{att}_H \circ \bar{m} = \hat{m}^* \circ \text{att}_G$, $\ell_H = \ell_G \circ \hat{m}$, and $\bar{\ell}_H = \bar{\ell}_G \circ \bar{m}$. 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 $G \hookrightarrow H$.

**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 [19] 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.

We use the standard definition of graph rewriting [4], and insist on injective matching of rules; this is no restriction, see [12]. We choose an alternative representation of rules proposed in [7] so that the rewriting of rules in Sect. 3 can be easier defined, see also in Appendix A.

**Definition 2.2 (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 $R$. For a function $f: A \rightarrow B$, its extension $f^*: A^* \rightarrow B^*$ to sequences $A^*$ is defined by $f^*(a_1 ... a_n) = f(a_1) \ldots f(a_n)$, for all $a_i \in A$, $1 \leq i \leq n$, $n \geq 0$; $f \circ g$ denotes the composition of functions or morphisms $f$ and $g$. 

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figs/fig1.png}
\caption{Two program graphs}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figs/fig2.png}
\caption{A refactoring rule}
\end{figure}
$R$ in a common body $B$. 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:

$$
\begin{array}{ccc}
G & \xrightarrow{m} & U \\
\downarrow & & \downarrow \\
B & \xrightarrow{m} & H
\end{array}
$$

The diagram exists if the morphism $m: P \to G$ 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 $m(P)$, giving $U$, and (ii) removing the nodes and edges $m(P \setminus R)$ from $U$, yielding $H$ with the *embedding* morphism $\tilde{m}: R \to H$.\(^3\) The construction is unique up to isomorphism, and yields a *rewrite step*, which is denoted as $G \Rightarrow_{r}^m H$.

**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*. However, it only applies if the class has exactly two subclasses, and if the method bodies have the particular shape specified in the pattern.

The graphs in Figure 1 constitute a rewrite step $G \Rightarrow_{pum'}^m H$. 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.)

The general *Pull-up Method* refactoring of Fowler [10] works for classes with any positive number of subclasses, and for method bodies of varying shape and size. This cannot be specified with a plain rule. The general refactoring will be specified in Example 4 further below.

Graph rewriting can be used for computations on graphs by applying a set of rules to some input graph as long as possible. Let $\mathcal{R}$ be a set of graph rewrite rules. We write $G \Rightarrow_{\mathcal{R}} H$ if $G \Rightarrow_{r}^m H$ for some match $m$ of a rule $r \in \mathcal{R}$, and

\(^3\) 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.
More on Graph Rewriting With Contextual Refinement

5

denote the transitive-reflexive closure of this relation by $\Rightarrow^*$. A graph $G$ is in normal form wrt. $R$ if there is no graph $H$ so that $G \Rightarrow_R H$. A set $\mathcal{R}$ of graph rewrite rules reduces a graph $G$ to some graph $H$, written $G \Rightarrow^! \mathcal{R} H$, if $G \Rightarrow^* \mathcal{R} H$ and $H$ is in normalform. $\mathcal{R}$ (and $\Rightarrow_R$) is terminating if it does not admit an infinite rewrite sequence $G_0 \Rightarrow_R G_1 \Rightarrow_R \ldots$, and confluent if for all rewrite sequences $H_1 \Rightarrow_R H_2 \Rightarrow_R H_3 \ldots$, there exists a graph $K$ with $H_1 \Rightarrow_R K \Rightarrow_R H_2$.

So, $\mathcal{R}$ defines a partial nondeterministic function from graphs to sets of their normalforms. This function is deterministic if $\mathcal{R}$ is confluent, and total if $\mathcal{R}$ is terminating.

Graph rewrite rules can also be used to derive sets of graphs, which are called languages, as for string grammars. A restricted form of rules has turned out to be useful for that purpose: they replace a variable by gluing a graph to its attached nodes and to some nodes in the context [3].

We assume that the labels contain a set $X \subseteq \Sigma$ of variable names that are used to label placeholders for subgraphs. $X(G) = \{ e \in G \mid \ell_G(e) \in X \}$ is the set of variables of a graph $G$, and $G$ 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 $x \in X$ come with a variable graph $G_x$, 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 $G_{\ell_G(e)}$, for every variable $e \in X(G)$.

Definition 2.3 (Contextual Grammar). A rule $r: (P \rightarrow B \leftarrow R)$ is contextual if the only edge $e$ in its pattern $P$ is a variable, and if $R$ equals $B$ without $e$.

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

The pattern $P$ of a contextual rule $r$ is the disjoint union of a variable graph $G_x$ 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 [11].)

Example 3 (Contextual Rules for Method Bodies). Figure 3 shows contextual rules. Variables are represented as boxes with their variable names inscribed; they are connected with their attached nodes by lines and arrows, ordered from left to right. When drawing contextual rules like those in Fig. 3, we omit the box around the 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 representing the replacement graph.

The set $M = \{ \text{body}_n, \text{use}, \text{call}_n, \text{ass} \}$ of contextual rules derives the data flow of method bodies in program graphs. A method body consists of expressions, which in turn either use the value of a variable, or call a method signature with expressions that are their actual parameters, or assign the value of an expression
Fig. 3. Contextual rules $M$ for method bodies

Fig. 4. Deriving a method body

to it. Actually, $body_n$ and $call_n$ abbreviate sets of (context-free) replicative rules that generate graphs with $n \geq 0$ copies of variables named $Exp$. The rules $body_n$ are context-free; in the rules for $Exp$, variable and signature nodes are contextual.

Figure 4 shows a derivation of a method body with $M$. Note that the body can only be derived if the start graph contains appropriate nodes representing variables and signatures. The missing rules of the complete grammar for program graphs are given in [3]; they do derive appropriate contextual nodes. (The language of program graphs cannot be derived with context-free rules [3].)

As for context-free string grammars, ambiguity is an important issue if the graphs derived by a contextual grammar shall be transformed. This property will be used in Lemma 3.6 further below.

**Definition 2.4 (Ambiguity).** Let $\Gamma = (\Sigma, R, Z)$ be a contextual grammar.

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

Then $\Gamma$ is unambiguous if all rewrite sequences $Z \Rightarrow^*_R G$ for a graph $G$ are equivalent to each other; if some graph $G$ has at least two rewrite sequences that are not equivalent, $\Gamma$ is ambiguous.

### 3 Schema Refinement with Contextual Meta-Rules

The graph rewriting tool grgen [1] supports object-oriented graph models with subtyping and attributes, named and parameterized rewrite rules with negative application conditions, and translates them to code that is highly optimized. Edgar Jakumeit [16,15] has extended the rules drastically, by introducing recursive refinement:

- Rules may contain variables; we call them schemata.
- The substitution of variables can be defined by meta-rules that are based on contextual rules as in Def. 2.3.
- A variable may be attached to nodes in the pattern and the replacement of a rule. Then its substitution refines pattern and the replacement of a schema
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 transform such subgraphs in a single rule application.

We started to study this way of rewriting with contextual refinement in [14]; this work shall be continued in this paper.

We lift morphisms from graphs to rules, for defining the rewriting of rules by meta-rules. For (graph rewrite) rules \( r \) and \( s \), a graph morphism \( m : B_r \to B_s \) on their bodies is a rule morphism, and denoted as \( m : r \to s \), if \( m(P_r) \leftarrow P_s \) and \( m(R_r) \leftrightarrow R_s \). Graph rewrite rules and rule morphisms form a category. This category has pushouts, pullbacks, and unique pushout complements along injective rule morphisms, just as graphs. As with graphs, we write rule inclusions as “\( \hookrightarrow \)”, and let \( \mathcal{R} \) be the kernel of a rule \( r \) wherein all variables are removed.

**Definition 3.1 (Rule Rewriting).** A pair \( \delta : (p \leftrightarrow b \leftrightarrow r) \) of rule inclusions is a rule rewrite rule, or meta-rule for short. With \( \delta_B \) we denote its body rule, which is a graph rewrite rule consisting of the bodies of \( p \), \( b \), and \( r \).

Consider a rule \( s \), a meta-rule \( \delta \) as above, and a rule morphism \( m : p \to s \). The meta-rule \( \delta \) rewrites the source rule \( s \) at \( m \) to the target rule \( t \), written \( s \ \downarrow^\delta \downarrow t \), if there is a pair of pushouts

\[
\begin{array}{ccc}
p & \xrightarrow{m} & b \\
\downarrow & & \downarrow \\
s & \xleftarrow{m} & u \\
\end{array}
\]

The pushouts above exist if the underlying body morphism of \( m \) satisfies the graph gluing condition wrt. the body rule \( \delta_B \) and the body graph \( B_s \).

We use meta-rules with contextual body rules, and apply them to rules that contain variables in their body (but neither in their pattern, nor in their replacement graphs).

**Definition 3.2 (Schema Refinement).** A schema \( s : (P \leftrightarrow B \leftrightarrow R) \) is a graph rewrite rule with \( P \cup R = B \).

Every schema \( s \) is required to be typed in the following sense: every variable name \( x \in X \) comes with a schema \( s_x \) with body \( G_x \) so that for every variable \( e \in X(B) \), the variable subgraph \( B(e) \) is the body of a subschema that is isomorphic to \( s_x \).

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

A less contextual variation \( \delta' \) of a meta-rule \( \delta \) equals \( \delta \) up to the fact that in its body rule \( \delta'_B \), some nodes of \( C_{\delta_B} \) are removed from \( P_{\delta_B} \), but kept in \( R_{\delta_B} \). Let \( \Delta \) be a finite set of meta-rules that is closed under less contextual variations.\(^4\)

Then \( \Downarrow \Delta \) denotes refinement steps with one of its meta-rules, and \( \Downarrow^* \Delta \) denotes

\(^4\) We explain in Example 7 why less contextual variations are needed.
Fig. 5. Pull-up Method: schema

Fig. 6. Replicating meta-rules $\Delta_M = \{\text{body}_{n,i}, \text{use}_i, \text{call}_{n,i}, \text{ass}_i\}$ for the rules $M$ in Fig. 3

repeated refinement, its reflexive-transitive closure. $\Delta(s)$ denotes the refinements of a schema $s$: $(P \hookrightarrow B \hookleftarrow R)$, containing its refinements without variables:

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

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 $H$.

Example 4 (Pull-Up Method). Fowler’s refactoring operation Pull-up Method [10] applies to a class $c$ where all direct subclasses contain bodies for the same method signature that are semantically equivalent.\footnote{This condition cannot be decided mechanically; it has to be confirmed by the user when s/he applies the operation, by a \textit{priori} verification or \textit{a-posteriori} testing.} It pulls one of these bodies up to $c$, and removes all others.

The meta-rules $\Delta_M = \{\text{body}_{n,i}, \text{use}_i, \text{call}_{n,i}, \text{ass}_i\}$ for generic meta-variables $\text{Bdy}_i$ and $\text{Exp}_i$ in Fig. 6 replicate method bodies as defined by the contextual
More on Graph Rewriting With Contextual Refinement

rules $M$ in Fig. 3: they remove one method body from a pattern and insert $i \geq 0$ copies of this body in the replacement of a schema. In the less contextual variations $\texttt{use}_i$, $\texttt{call}_n,i$, and $\texttt{ass}_i$ of these meta-rules (which are not shown here) the $S$- and $V$-nodes are no longer contextual. The schema $\texttt{pum}$ in Fig. 5 uses several meta-variables $\texttt{Bdy}_0$ that just remove one method body from a subclass in the pattern, and one variable $\texttt{Bdy}_1$ that moves a method body from one subclass in the pattern to the superclass in the replacement.\(^6\)

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, and (iii) both at $e$ and $v$ if $v$ occurs in both, pattern and replacement. (The last case does not occur in our example.)

The rule $\texttt{pum}'$ in Fig. 2 is a refinement of $\texttt{pum}$ with $\Delta_M$, i.e., $\texttt{pum}' \in \Delta_M(\texttt{pum})$. The upper row in Fig. 10 on page 12 shows a step in the refinement sequence $\texttt{pum} \triangleright^\ast \Delta_M \texttt{pum}'$; it applies the context-free variation $\texttt{ass}_i$ of the meta-rule $\texttt{ass}_i$ in Fig. 6.

A single rewriting step with some refinement of $\texttt{pum}$ copies one method body of arbitrary shape and size, and deletes an arbitrarily number of other bodies, which are also of variable shape and size. This goes beyond the expressiveness of plain rewrite rules, which may only match, delete, and replicate subgraphs of constant size. Note that the application of a refinement $r \in \Delta(s)$, although it is the result of a compound meta-derivation, is a single rewriting step $G \Rightarrow_r H$ on the source graph $G$, similar to a transaction in a data base. Note also that the refinement process is completely rule-based.

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.

The following assumption excludes useless sets of meta-rules.

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

This property is decidable for contextual grammars [3, Corollary 2].

We need a mild condition to show that residual rewriting terminates.

**Definition 3.3 (Pattern-Refining Meta-Rules).** A meta-rule $(p \hookrightarrow b \leftarrow r)$ refines its pattern if $X(R_r) = \emptyset$ or if $P_r \not\sim P_p$. A set $\Delta$ of meta-rules that refine their patterns is called pattern-refining.

**Theorem 3.4.** For a schema $s$ and a set $\Delta$ 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 [14], 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. \(\square\)

\(^6\) The ellipses “…” allows any number $k \geq 0$ subclasses to be matched for removing a body. In Fowler’s operation, no further subclasses should exist. However, this could only be specified with a negative application condition for the schema, in future work.
We now turn to the question whether the patterns of refinements uniquely define the replacement they perform.

**Definition 3.5 (Right-Unique Meta-Rules).** A set \( \mathcal{R} \) of graph rewrite rules is right-unique if different meta-rules \( r_1, r_2 \in \Delta \) have different patterns, i.e., \( P_1 \cong P_2 \) implies that \( r_1 \cong r_2 \).

We have to define an auxiliary notion first. The pattern rule \( \delta_P \) of a meta-rule \( \delta = (p \hookrightarrow b \leftarrow r) \) is a contextual rule obtained from the body rule \( \delta_B \) by removing all nodes and edges in \( B_b \setminus R_b \), and by detaching all variables in \( \delta_B \) from the removed nodes. Let \( \Delta_P \) denote the set of (contextual) pattern rules of a set \( \Delta \) of meta-rules.

**Lemma 3.6 (Right-Uniqueness of Refinements).** A set \( \Delta(s) \) of refinements is right-unique if the pattern grammar \( (\Sigma, \Delta_P, P_s) \) 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 \( \Delta_P \) is unambiguous. This rewriting sequence has a unique extension to a meta-rewrite sequence so that \( r_1 \cong r_2 \). \( \square \)

**Example 5 (Pattern-Refining, Right-Unique Meta-Rules).** The meta-rules \( \Delta_M \) in Fig. 6 are pattern-refining. The contextual rules \( M \) for method bodies in Fig. 3 are unambiguous. They correspond to the pattern rules of the meta-rules \( \Delta_M \) in Fig. 6, so that these are right-unique. (The meta-rules for the encapsulate Field refactoring schema in [14, Ex. 5] are pattern-refining and right-unique as well.)

## 4 Modeling Refinement by Residual Rewriting

As a first step to analyzing further properties of schemata and meta-rules, we translate them into standard graph rewrite rules:

- We turn every schema into an ordinary rule that delays refinement, by adding the meta-variables to its replacement, with all their attached nodes.
- We turn every contextual meta-rule \( \delta = (p \hookrightarrow b \leftarrow r) \) into a graph rewrite rule that refines the delaying rule incrementally, by adding the pattern of \( r \) to that of \( p \), and the variable graphs of \( B_r \) to the replacement \( R_r \).

**Definition 4.1 (Incremental Refinement Rules).** Let \( s: (P \hookrightarrow B \leftarrow R) \) be a schema for meta-rules \( \Delta \).

The delaying rule \( \check{s}: (P \hookrightarrow B \leftarrow R_s) \) of \( s \) has the same pattern \( P \) and body \( B \) as \( s \), and its replacement \( R_s = R \cup \{B(e) \mid e \in X(B)\} \) is obtained by uniting \( R \) with the graphs of all variables in \( B \).

For a meta-rule \( \delta = (p \hookrightarrow b \leftarrow r) \), the incremental rule \( \check{\delta}: (\check{P} \hookrightarrow \check{B} \leftarrow \check{R}) \) has the pattern \( \check{P} = B_p \cup R_r \), a replacement \( \check{R} = R_r \cup \{B_r(e) \mid e \in X(B_r)\} \), and the body \( \check{B} = B_b \). \( \check{\Delta} \) denotes the incremental rules of \( \Delta \).

---

7 The graphs and in \( \Delta_P \) are also typed, but in the type graph \( G_x \) of a variable name \( x \), all nodes that do not belong to the pattern of the schema \( s_x \) are removed.
Example 6 (Incremental Refinement). Figure 7 shows how the schema \( \text{pum} \) for the Pull-up Method refactoring in Fig. 5 is translated into a delaying rule \( \tilde{\text{pum}} \), and how the context-free variation \( \text{ass}_1 \) of the meta-rule \( \text{ass}_1 \) in Fig. 6 is translated into an incremental rule \( \tilde{\text{ass}}_1 \). (In the delayed rule \( \tilde{\text{pum}} \), red arrow and waves (appearing grey in B/W print) indicate edges that do not belong to the replacement.)

If a schema \( s \) is refined with a metarule \( \delta \) to a schema \( t \), the composition \( \tilde{s} \circ_d \tilde{\delta} \) of its delayed and incremental rules (defined in Def. A.1) equals the delayed rule \( \tilde{t} \) (for a particular dependency \( d \)).

Lemma 4.2. Consider a schema \( s = (P \rightarrow B \leftarrow R) \) and a meta-rule \( \delta: (p \rightarrow b \leftarrow r) \).

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 \( d: (R \leftarrow B \rightarrow (B_p \cup R_p)) \) so that \( r^d = \tilde{t} \).

Proof Sketch. Let \( s, \delta \) be as above, \( t: (P' \rightarrow B' \leftarrow R') \), \( \tilde{s}: (P \rightarrow B \leftarrow R) \) with \( R_\tilde{s} = R \cup \{B(e) \mid e \in X(B)\} \), and \( \tilde{\delta}: (\tilde{P} \rightarrow \tilde{B} \leftarrow \tilde{R}) \) 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. 4.1. Their composition according to the dependency \( d: (R \leftarrow B \rightarrow (B_p \cup R_p)) \) is constructed as in Def. A.1, and shown in Fig. 9.

By this approach, the result \( r^d = \tilde{s} \circ_d \tilde{\delta} \) is constructed as in Def. A.1, and shown in Fig. 9.
Consider the underlying body refinement \( B \Rightarrow_B^m B' \). (See Fig. 8, 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 \( \delta_B \) is contextual, and \( B' = U \setminus \tilde{m}(X(B_p)) \).

It is then easy to show that the body \( B'_d \) equals the body \( B'^d \) of the composed delaying rule, and an easy argument concerning the whereabouts of variables shows that \( t = r^d \).

Example 7 (Schema Refinement and Incremental Rules). Figure 10 illustrates the relation between schema refinement and the composition of their incremental rules established in Lemma 4.2. As already mentioned in Example 4, the upper row shows a step in the refinement sequence \( \text{pum} \bowtie_{\Delta_M} \text{pum}' \) that applies the context-free variation \( \text{ass}_1 \) of the meta-rule \( \text{ass}_1 \) in Fig. 6. This step shows why we need less contextual variations of meta-rules: 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 delaying rule with the corresponding incremental refinement rule \( \text{ass}_1 \), where the dashed box
More on Graph Rewriting With Contextual Refinement

specifies the dependency $d$ for the composition. The composed rule equals the delaying rule for the refined schema.

Using a refined schema has the same effect as applying its delaying 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 delaying rule is applied.

We define the subgraph that is left unchanged in refinement steps and sequences. To ease the following definitions, we assume wlog. that a rewrite step $G \Rightarrow^m H$ with a diagram as in Def. 2.2 is constructed so that the lower horizontal morphisms are inclusions $G \hookrightarrow U \hookleftarrow H$. The track of $G$ in $H$ (via the match $m$ of the rule $r$) is then defined as $\text{tr}^m_r(G) = (G \cap H)$. For a rewrite sequence $d = G_0 \Rightarrow^{m_1} G_1 \Rightarrow^{m_2} \ldots \Rightarrow^{m_n} G_n$, the track of $G$ in $H$ is given by intersecting the tracks of its steps:

$$\text{tr}_d(G) = \text{tr}^{m_1}_{r_1}(G_0) \cap \cdots \cap \text{tr}^{m_n}_{r_n}(G_{n-1})$$

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

**Definition 4.3 (Residual Incremental Refinement).** Consider an incremental refinement sequence $G_0 \Rightarrow^{\delta_1} G_1 \Rightarrow^{\delta_2} \ldots \Rightarrow^{\delta_n} G_n$ with incremental rules $\tilde{\delta}_i$ for meta-rules $\delta_i$: $(p_i \hookrightarrow b_i \hookleftarrow r_i)$ (for $1 \leq i \leq n$).

The step $G_{i-1} \Rightarrow^{\delta_i} G_i$ is residual if $m_i(P_{r_i}) \subseteq \text{tr}^{m_{r_i}}_{r_{i-1}}(G)$. The sequence is residual if every of its steps is residual. Residual steps and sequences are denoted as $\Rightarrow$ and $\Rightarrow^*$, respectively.

**Lemma 4.4.** Consider a schema $s$ for meta-rules $\Delta$ with delaying rule $\tilde{s}$ and incremental rules $\Delta$.

Then a rule $r: (P \hookrightarrow B \hookleftarrow R)$ is a refinement in $\Delta(s)$ if and only if $P \Rightarrow^* s P' \Rightarrow^* \Delta R$.

**Proof.** By induction over the length of meta-derivations, using Lemma 4.2 and the fact that compositions correspond to residual rewrite steps. 

**Theorem 4.5.** Consider a schema $s$ with meta-rules $\Delta$ as above. Then, for graphs $G$, $H$, and $K$, $G \Rightarrow_{\Delta(s)} H$ if and only if $G \Rightarrow^* \Delta K \Rightarrow^* \Delta H$.

**Proof.** Combine Lemma 4.4 with the embedding theorem [4, Sect. 6.2].

5 Conclusions

In this paper we have defined how the refinement of schemata of plain graph rewrite rules according to contextual meta-rules can be translated to standard
rules that perform the refinement incrementally. We have also investigated conditions under which the refinement behaves well, i.e., terminates, and yields unique refinements.

Our ultimate goal is to analyze confluence of systems of schemata and meta-rules with the critical pair lemma [17]. The negative result shown in [14, Thm. 3] indicates that considerable restrictions have to be made to reach this aim. A possible way could be to restrict the rewriting with refinements to graphs that are shaped, e.g., according to contextual grammars like the program graphs shown in this paper.

Until now, we have not considered attributed graphs and subtyping. As they are included in the foundation [4], we expect that this can be added in a rather orthogonal way. We also restricted ourselves to unconditional rules. Rules with nested application conditions have been added to the theory in [6,5]; recently, Hendrik Radke has studied recursive refinement of such conditions [13]. We plan to add these concepts to our definition in the future.

Acknowledgments. I am indebted to Annegret Habel and Rachid Echahed for their patience and their encouragement.

References

More on Graph Rewriting With Contextual Refinement


A Double-Pushout Graph Rewriting

The standard theory of graph rewriting is based on so-called spans of (injective) graph morphisms [4], where a rule consists of two morphisms from a common interface $I$ to a pattern $P$ and a replacement $R$. An alternative proposed in [7] 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:

```
\begin{array}{c}
\vdash P \triangleleft I \triangleleft R \\
\vdash P \triangleright B \triangleright R \\
m \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
G \triangleleft C \triangleleft H \\
\end{array}
```

Intuitively, rewrites are constructed via a match morphism $m: P \to G$ in a source graph $G$; for a span rule $\vdash$, 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 cospan rule \( \hat{r} \), the new items \( B \setminus P \) are added first, yielding the united graph \( U \) before the obsolete pattern items \( B \setminus P \) are removed. The constructions work if the matches \( m \) satisfy certain gluing conditions.

The main result of [4] says that \( \hat{r} \) is the pushout of \( \tilde{r} \), 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 \( \tilde{r} \) together with a rule \( \hat{r} \).

The following definition and theorem adapt well-known concepts of [4] to our notion of rules.

**Definition A.1 (Sequential Rules Composition).** Let \( r_1 : (P_1 \leftrightarrow B_1 \leftrightarrow R_1) \) and \( r_2 : (P_2 \leftrightarrow B_2 \leftrightarrow R_2) \) be rules, and consider a graph \( D \) with a pair \( d : (R_1 \leftrightarrow D \rightarrow P_2) \) of injective morphisms.

1. Then \( d \) is a sequential dependency of \( r_1 \) and \( r_2 \) if \( D \neq \langle \rangle \) (which implies that \( D \neq \langle \rangle \)).
2. The sequential composition \( r_1 \circ_d r_2 : (P^d \leftrightarrow B^d \leftrightarrow R^d) \) of \( r_1 \) and \( r_2 \) along \( d \) is the rule constructed as in the commutative diagram of Fig. 11, where all squares are pullbacks.
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 \( P_2 \rightarrow H \).

**Proposition A.2.** Let \( r_1 \) and \( r_2 \) be rules with a dependency \( d \) and a sequential composition \( r_d^d \) as in Def. A.1.

Then there exist \( d \)-related rewrite steps \( G \Rightarrow r_1 H \Rightarrow r_2 K \) if and only if \( G \Rightarrow r_d^d K \).

**Proof.** Straightforward use of the corresponding result for “span rules” [4, Thm. 5.23] and of the duality to “cospan rules” [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., \( A \rightarrow B \rightarrow D = B \rightarrow C \rightarrow D \), and universal, i.e., for every pair of morphisms \( B \rightarrow A' \leftarrow C \) 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 [4, Def. 2.2]

---

**Fig. 11.** Sequential composition of graph rewrite rules