Conditional Adaptive Star Grammars

Berthold Hoffmann

Abstract. The precise specification of software models is a major concern in model-driven design of object-oriented software. In this paper, we investigate how program graphs, a language-independent model of object-oriented programs, can be specified precisely, with a focus on static structure rather than behavior. Graph grammars are a natural candidate for specifying the structure of a class of graphs. However, neither star grammars—which are equivalent to the well-known hyperedge replacement grammars—nor the recently proposed adaptive star grammars allow all relevant properties of program graphs to be specified. So we extend adaptive star rules by positive and negative application conditions, and show that the resulting conditional adaptive star grammars are powerful enough to generate program graphs.

1 Introduction

Model-driven design of object-oriented software aims at describing the static structure, dynamic behavior, and gradual evolution of a system in a comprehensive way. Typically, a software model is a collection of graph-like diagrams that is often specified by a meta-model. For instance, the static structure of a system is often defined by class diagrams of the UML. Since graph grammars are another candidate for specifying graph-like structures, we investigate how they can be used to define software models. As a case study, we consider program graphs, a language-independent model of object-oriented programs that has been devised for specifying refactoring operations on programs [14]. Several kinds of graph grammars have been proposed in the literature. Here we need a formalism that is *powerful* so that all properties of models can be captured, and *simple* in order to be practically useful, in particular for *parsing* models in order to determine their consistency. However, neither star grammars (equivalent to the well-known hyperedge replacement grammars [13, 4]), nor node replacement grammars [12] are powerful enough for our purpose. Even the recently proposed adaptive star grammars [6, 5] fail for certain more delicate properties of program graphs. So we define *conditional adaptive star grammars* in this paper. In these grammars, adaptive star rules are extended by positive and negative application conditions. (Informally, application conditions have already been considered in [9, 7].) Conditional adaptive star grammars capture all relevant properties of program graphs.

The paper is structured as follows. In Section 2, we recall how object-oriented programs can abstractly be represented as *program graphs*. Then we introduce star grammars in Section 3, show that they can define *program trees*, a substructure of program graphs, and discuss why they cannot define program graphs themselves. In Section 4, we therfore recall the *adaptive star grammars* devised in [6, 5]. Close inspection reveals that even this formalism fails to capture properties of program graphs. So we extend adaptive star grammars further, by rules with positive and negative application conditions, in Section 5. These *conditional adaptive star grammars*, finally, allow program graphs to be defined completely. We conclude with some remarks on related and future work in Section 6.

2 Graphs Representing Object-Oriented Software

In model-driven software development, software is represented by diagrams, e.g., of UML. Formally, such diagrams can be defined as many-sorted graphs.

Definition 2.1 (Graph). Let $\Sigma = \langle \dot{\Sigma}, \bar{\Sigma} \rangle$ be a pair of disjoint finite sets of *sorts*.

A *many-sorted directed graph over* Σ (*graph*, for short) is a tuple $G = \langle \dot{G}, \bar{G}, s, t, \sigma \rangle$ where $\dot{G}$ is a finite set of *nodes*, $\bar{G}$ is a finite set of *edges*, the functions $s, t \colon \bar{G} \to \dot{G}$ define the *source* and *target* nodes of edges, and the pair $\sigma = \langle \dot{\sigma}, \bar{\sigma} \rangle$ of functions $\dot{\sigma} \colon \dot{G} \to \dot{\Sigma}$ and $\bar{\sigma} \colon \bar{G} \to \bar{\Sigma}$ associate nodes and edges with sorts.

Given graphs G and H, a pair $m = \langle \dot{m}, \bar{m} \rangle$ of functions $\dot{m} \colon \dot{G} \to \dot{H}$ and $\bar{m} \colon \bar{G} \to \bar{H}$ is a *morphism* if it preserves sources, targets and sorts. A morphism m is *surjective* or *injective* if both $\dot{m}$ and $\bar{m}$ have the respective property. If the morphism $m \colon G \to H$ is both injective and surjective, it is an *isomorphism*, and G and H are called *isomorphic*, written $G \cong H$.

In figures of graphs, different sorts of edges are represented by arrows drawn with different widths or dashing, whereas nodes are distinguished by their shape, which may be a box or a circle, and by a label inscribed to that shape.

Program graphs have been devised as a language-independent representation of object-oriented code that can be used for studying refactoring operations [14]. They capture concepts that are common to many object-oriented languages, like single inheritance and method overriding, whereas properties particular to a few languages—like multiple inheritance—are left out.

Example 2.1 (A Program Graph). Figure 1 depicts a program graph for a simple object-oriented program from [1], which is shown in Figure 2. The nodes of a program graph, drawn as circles, represent syntactic entities of a program: classes (C), variables (V), method signatures (M) and bodies (B), and expressions (E). Edges establish relations between entities: "↓" is pronounced "*contains*" , and "⇣" is pronounced "*refers to*".

Nodes of sort C are called "class nodes" or just "classes", and so for the other sorts of nodes. The variables contained in a method signature are called its *parameters*, and we say that a class c' is a *super-class* of a class c if either c' equals c, of if some class contained in c' is a super-class of c. In a similar way, we define a *sub-expression* of a body or expression. If a body b refers to a method signature m, we say that b *implements* m. In expressions, only data flow is represented: a reference to a method represents a *call*; a reference to a

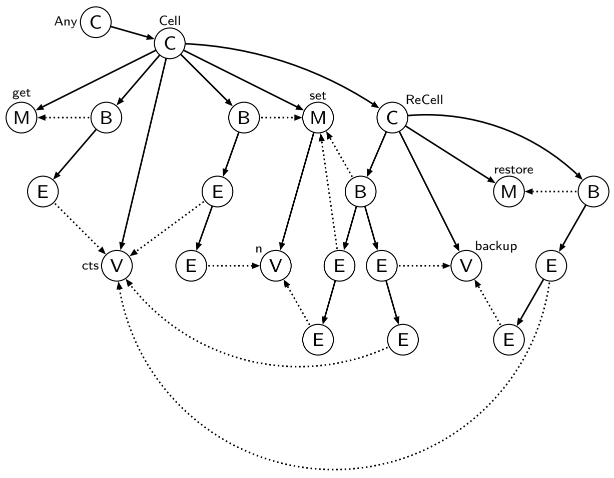

Fig. 1. A program graph

```
class Cell is
  var cts: Any;
  method get() Any is
    return cts;
  method set(var n: Any) is
    cts := n

subclass ReCell of Cell is
  var backup: Any;
  method restore() is
    cts := backup;
  override set(var n: Any) is
    backup := cts;
    super.set(n)
```

Fig. 2. A simple OO program

variable represents an *access* that either *uses* its value, or *assigns* the value of an expression to it.

Definition 2.2 (Program Graph). A graph G is a *program graph* if the following conditions are satisfied:

1. In G, nodes and edges are of the sorts $\{\mathsf{C}, \mathsf{V}, \mathsf{M}, \mathsf{B}, \mathsf{E}\}$ and {↓, ⇣}, respectively.
2. Nodes and edges may be incident as shown in Figure 3. In particular,
   (a) a body contains at least one expression, and implements exactly one method signature, and
   (b) an expression either calls exactly one method, or it accesses exactly one variable; in the latter case, it contain at most one expression.
3. The subgraph $\bar{G}$ of G induced by ↓-edges is a spanning tree of G; the root of $\bar{G}$ is a class.
4. An expression may call any method contained in any class.
5. An expression contained in a class c may access any variable contained in any super-class of c.
6. An expression e may access a parameter of a method m if e is a subexpression of a body implementing m.

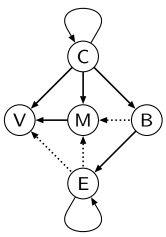

Fig. 3. Incidence of nodes and edges in program graphs

7. A body b contained in some class c may implement any method signature contained in any super-class of c.
8. Every class may contain at most one body implementing a particular method signature m.
9. If an expression e calls a method m, the number of m's parameters must match the number of expressions contained in e.

The class of program graphs is denoted by $\mathcal{P}$.

The graph in in Figure 3 is called a type graph in [10], and a graph schema in Progres [17]; Properties 2.2.1–3 could be described by a class diagram in Uml. Property 2.2.4 defines the visibility of all methods as *public*, and Property 2.2.5 defines the visibility of all variables as *protected*, in the terminology of Java.

The graph-theoretic structure of program graphs is as follows.

Definition 2.3. A rooted, connected, acyclic graph is called a *collapsed tree.*

Fact 2.1. Program graphs are collapsed trees.

Proof Sketch. Acyclicity follows from Property 2.2.2: Cyclic incidences are allowed only for nesting of classes and expressions, but these occur in the underlying the spanning tree so that they do not lead to cycles. Property 2.2.3 implies connectedness; The root class of the spanning tree is the root of the program graph as well, because property 2.2.2 forbids references to classes.

3 Star Grammars

Star grammars are a special case of double pushout (DPO) graph transformation [10], and equivalent to hyperedge replacement grammars [13, 4], a well-understood context-free kind of graph grammars. They are recalled just as a basis for the extensions defined in Sections 4 and 5.

Definition 3.1 (Variable). From now on we assume that the node sorts contain *variable sorts* $\dot{\Sigma}_{\mathrm{v}} \subseteq \dot{\Sigma}$ that define the *terminal node sorts* as $\dot{\Sigma}_{\mathrm{t}} = \dot{\Sigma} \setminus \dot{\Sigma}_{\mathrm{v}}$.

Consider a star-like graph X, with one center node c_X of sort $x \in \dot{\Sigma}_{\mathrm{v}}$, and with some border nodes (of terminal sorts from $\dot{\Sigma}_{\mathrm{t}}$) so that the edges connect c_X to every border node; Then X is called a (*syntactic*) *variable named* x. A variable is *straight* if every border node is incident with exactly one edge.

A graph G is a *graph with variables* if all nodes named with variables are not adjacent to each other.[1] Let $\mathcal{X}$ denote the class of (*syntactic*) *variables*, $\mathcal{G}(\mathcal{X})$ the class of graphs with variables, and $\mathcal{G}$ be the class of graphs without variables (with node sorts from $\dot{\Sigma}_{\mathrm{t}}$).

Definition 3.2 (Star Replacement). A *star rule* is written $L ::= R$, where the *left-hand side* $L \in \mathcal{X}$ is a straight variable and the *replacement* is a graph $R \in \mathcal{G}(\mathcal{X})$ that contains the border nodes of L.

[1] Then all nodes labeled with variable names are centers of stars.

A variable Y in a graph G is a *match* for a star rule $L ::= R$ if there is a surjective morphism $m\colon L \to Y$ where $\bar{m}$ is bijective. Then a *star replacement* yields the graph denoted as $G[Y/_m R]$, which is constructed by adding the nodes $\dot{R} \setminus \dot{L}$ and edges $\bar{R}$ disjointly to G, and by replacing, for every edge in $\bar{R}$, every source or target node $v \in \dot{L}$ by the node $\dot{m}(v)$, and by removing the edges $\bar{Y}$ and center node c_Y.

Let $\mathcal{R}$ be a finite set of star rules. Then we write $G \Rightarrow_{\mathcal{R}} H$ if $H = G[Y/_m R]$ for some $L ::= R \in \mathcal{R}$, some variable Y in G, and some match Y, and denote the reflexive-transitive closure of this relation by $\Rightarrow^*_{\mathcal{R}}$.

Example 3.1 (Star Replacement). Figure 4 shows a star rule $L ::= R$ for an assignment expression. The center nodes of variables are drawn as boxes enclosing the variable name. We shall draw a star rule "blowing up" the center node of L and placing the new nodes and edges of R inside, as can be seen on the right-hand side of Figure 4. A star rule can be represented as it is drawn, as a single *rule graph* wherein a variable is distinguished as the rule's left-hand side. This way, graph operations can be applied to star rules as well. Figure 5 shows a schematic star replacement $G_0 \Rightarrow_{\mathsf{ass}} G_1$ using the rule.

Definition 3.3 (Star Grammar). $\Gamma = \langle \mathcal{G}(\mathcal{X}), \mathcal{X}, \mathcal{R}, Z \rangle$ is a *star grammar* with a *start variable* $Z \in \mathcal{X}$. The *language* of Γ is obtained by exhaustive star replacement with its rules, starting from the start variable:

$$\mathcal{L}(\Gamma) = \{G \in \mathcal{G} \mid Z \Rightarrow^*_{\mathcal{R}} G\}$$

Example 3.2 (Star Grammar for Program Trees). Figure 6 shows the star rules generating the program trees that underlie program graphs. The rules define a star grammar PT according to the following convention: The left-hand side of the first rule indicates the start variable, a variable named **Prg** with a class as a border node in this case. The sorts used in the rules define the sorts of the grammar.

Boxes with dashed lines and/or shades indicate *multiple subgraphs*: a shade indicates that the subgraph may have several copies, which are called *clones* as each of them is connected to the remaining graph in the same way; a dashed line indicates that a subgraph may be present, or missing. (In hy, a hierarchy

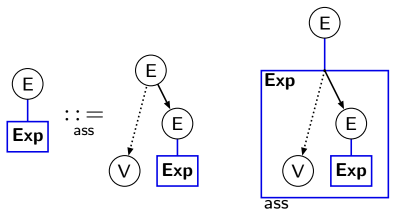

Fig. 4. A star rule and its boxed form

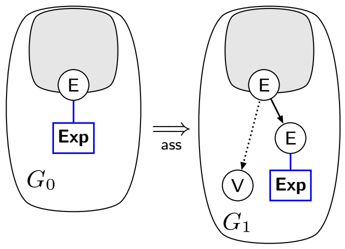

Fig. 5. A star replacement

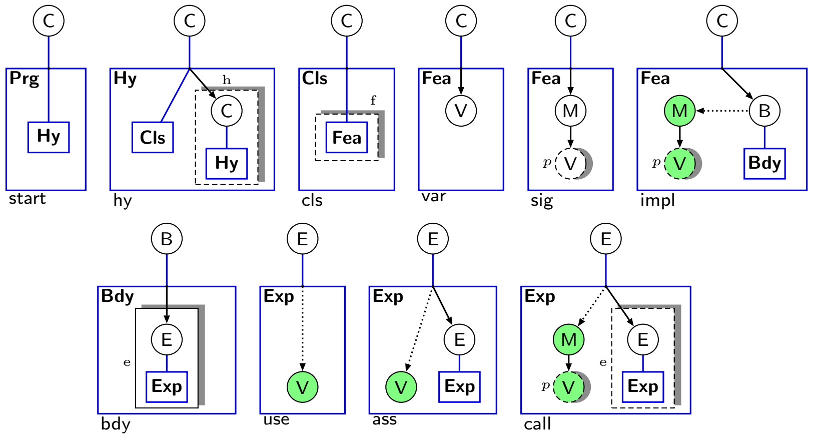

Fig. 6. The rules of the star grammar PT generating program trees

may have $n \geqslant 0$ sub-hierarchies; in bdy, a body contains $n \geqslant 1$ expressions.) Analogously, the dashed and shaded variables in sig, impl, and call indicate multiple nodes that may have $n \geqslant 0$ clones. Note that multiple nodes and subgraphs are just abbreviations, which can be replaced by auxiliary variables that are defined by auxiliary star rules, just as the iteration operators of the extended Backus-Naur Form for context-free word grammars.

Grey nodes designate nodes in the program tree that have to be identified with nodes representing their declarations in order to get a program graph according to Definition 2.2: These are the method and parameters generated in impl and call, and the variables accessed in use and ass.

Inspection of the rules in PT reveals the following.

Fact 3.1. $\mathcal{L}(\mathsf{PT})$ is a language of trees.

The language of PT is closely related to program graphs.

Definition 3.4. The *unraveling* $\hat{G}$ of a collapsed tree G is a tree so that there is a surjective morphism $r \colon \hat{G} \to G$. Let $\hat{\mathcal{P}} = \{\hat{G} \mid G \in \mathcal{P}\}$ denote the unravelings of program graphs.

Fact 3.2. $\hat{\mathcal{P}} \subsetneq \mathcal{L}(\mathsf{PT})$.

Proof Idea. (I): $\hat{\mathcal{P}} \neq \mathcal{L}(\mathsf{PT})$. Inspection of the rules shows that the trees generated with PT satisfy Properties 2.2.1–2.

Also, every graph G in $\mathcal{P}$ satisfies Properties 2.2.1–9. Properties 2.2.1–2 are preserved in the unraveling $\hat{G}$. Properties 2.2.4–8 are irrelevant for $\hat{G}$, as the methods, variables, and parameters referred are copied by the unraveling operation. Property 2.2.9 is unchanged under unraveling

(II): $\hat{\mathcal{P}} \neq \mathcal{L}(\mathsf{PT})$. Rule call may have clones where the formal parameters p have n clones, whereas the actual parameters e have $m \neq n$ clones. However, a

tree generated with such a clone cannot be the unraveling of a program graph, which satisfies Property 2.2.9. (For other rules, one can find similar examples.)

Star grammars are context-free in the sense defined by Courcelle [3]. This suggests that their generative power is limited. Indeed, we have the following

Conjecture 3.1. *There is no star grammar Γ with $\mathcal{L}(\Gamma) = \mathcal{P}$.*

Consider Figure 5 to see why we have this conjecture. The rule ass allows to derive trees of assignments. However, for generating a program graph, the rule should insert a reference to a variable that already exists in graph G_0, and is accessible in the expression. Due to the restricted form of star rules, such a node had to be on the border of ass. Since assignments may set every accessible variable, rule ass must have all these variables on its border nodes so that one of them can be selected in the rule. However, the number of accessible variables depends on the size of the program, and is unbounded. Thus a finite set of star rules cannot suffice to define all legal assignments.

4 Adaptive Star Grammars

Proposition 3.1 gives a clue how the limitation of star grammars can be overcome. We make the left-hand sides of star rules *adaptive* wrt. the numbers of border nodes, as proposed in [6, 5]. Formally, this is defined by cloning.

Definition 4.1 (Singular and Multiple Nodes). We assume that the sorts $\Sigma = \langle \dot{\Sigma}, \bar{\Sigma} \rangle$ are given so that the terminal node sorts $\dot{\Sigma}_\mathrm{t}$ contain a set $\ddot{\Sigma}_\mathrm{t}$ of *multiple* sorts so that every remaining *singular sort* $s \in \dot{\Sigma}_\mathrm{t} \setminus \ddot{\Sigma}_\mathrm{t}$ has a unique multiple sort $\ddot{s} \in \ddot{\Sigma}_\mathrm{t}$, and vice versa.

From now on, $\mathcal{X}$, $\mathcal{G}$ and $\mathcal{G}(\mathcal{X})$ denote classes of graphs with singular sorts only, whereas $\ddot{\mathcal{X}}$, $\ddot{\mathcal{G}}$ and $\ddot{\mathcal{G}}(\ddot{\mathcal{X}})$ denote classes of *adaptive graphs* that may contain multiple sorts as well.

A star rule $L ::= R$ is called *adaptive* if $L \in \ddot{\mathcal{X}}$ and $R \in \ddot{\mathcal{G}}(\ddot{\mathcal{X}})$.

Definition 4.2 (Cloning). Let G be a graph in $\ddot{\mathcal{G}}(\ddot{\mathcal{X}})$ with a multiple node v that is labeled with $\ddot{\ell} \in \ddot{\Sigma}$, and incident with edges $e_1, \ldots, e_n$ ($n \geqslant 0$). Then $G\frac{v}{k}$ denotes the *clone* of G in which v is replaced by $k \geqslant 0$ singular nodes $v_1, \ldots, v_k$ that are labeled with ℓ, where every clone v_i is incident with copies $e_{i,1}, \ldots, e_{i,n}$ of the edges $e_1, \ldots, e_n$ incident with v.

If $r = L ::= R$ is an adaptive star rule with a multiple node v, its clone $r\frac{v}{k}$ is obtained by cloning its rule graph.

Example 4.1 (Adaptive Star Cloning, and Label Specialization). The star rule ass on the left of Figure 7 is adaptive: its variable a is a multiple node, and shall match a set of $n \geqslant 0$ variables in the host graph that are accessible in the expression. On the right-hand side, a schematic view of the clone $\mathsf{ass}\frac{a}{n}$ is given, for $n \geqslant 0$.

The *abstract sort* F of nodes a and a_i is a placeholder for the concrete subsorts V and M. (F stands for for *feature.*) Before applying the clone $\mathsf{ass}\frac{a}{n}$, each of the labels F is specialized either to V or M. As with multiple nodes and subgraphs, a star rule with abstract sorts is just an abbreviation for a set of star rules wherein these abstract sorts are replaced with any combination of concrete sub-sorts.

Definition 4.3 (Adaptive Star Grammar). Let $\Gamma = \langle \ddot{\mathcal{G}}(\ddot{\mathcal{X}}), \ddot{\mathcal{X}}, \mathcal{R}, Z\rangle$ be a star grammar over adaptive variables and graphs. Then Γ is called *adaptive* if $Z \in \mathcal{X}$ (i.e., has no multiple nodes).

Let $\ddot{\mathcal{R}}$ denote the set of all possible clones of a set $\mathcal{R}$ of adaptive star rules. Then Γ generates the language

$$\ddot{\mathcal{L}}(\Gamma) = \{G \in \mathcal{G} \mid Z \Rightarrow^*_{\ddot{\mathcal{R}}} G\}$$

The set of star rules $\ddot{\mathcal{R}}$ generated from a set of adaptive star rules is infinite if at least one of the adaptive star rules contains a multiple node or subgraph. It has been shown in [5] that this gives adaptive star grammars greater generative power than grammars based on hyperedge [13] or node replacement [12], but they are still parseable [6].

Example 4.2 (Adaptive Star Grammar for Program Graphs). The adaptive star rules in Figure 9 define an adaptive star grammar PG that systematically extends the program tree grammar PT of Figure 6.

With two exceptions, the rules of PG just extend those of PT. In PG, rule meth defines a method declaration, which combines a signature sig with an (optional) implementation impl, whereas ovrd defines the overriding of a method in the subclass of the original method definition.

In Figure 8 we show the general form of variables in PG and of the program subgraphs they generate. (In derivations, the multiple nodes d, v, and o of X are cloned.) The sorts of edges indicate the following roles of the border nodes. Node r is the *root* of the program subgraph G_X derived from X. Clones of d are the features *declared* in G_X. Clones of v are the features that are *visible* in G_X. Clones of o are the methods that are *overridable* in G_X. Features may have multiple roles in X and G_X: every feature declared by X is also visible in X, and

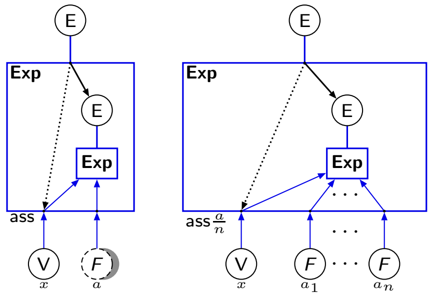

Fig. 7. An adaptive rule and its clones

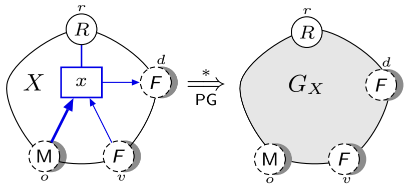

Fig. 8. Variables and derivations in PG

overridable methods are visible as well so that some clones d and v, and some clones of v and o in X may identified. On the left-hand side of rules, the clones of d, v and o in a variable X are be distinct (they are required to be straight) so that they must be identified by matching. The graph G_X is directed and acyclic. Some of its visible border nodes may be isolated. The rest is a collapsed tree with root r.

The rules in Figure 9 extend the rules of Figure 6 by adding border nodes to variables according to the roles explained above. The rules for **Fea** declare a variable or a method (or just override an existing method). The rule cls declares its member variables and methods. A hierarchy declares all methods of its top class and of its sub-hierarchies, makes the variables of the top class visible in the class itself and in the sub-hierarchies, and makes the methods of the top class overridable in the classes of its sub-hierarchies. The rule start makes all methods declared by the program hierarchy visible in it. All rules pass visible features down to the leaves of the program graph. The rules for **Exp** then select visible variables for being used or assigned to, and methods for being called; rule ovrd selects an overridable method signature for overriding it with a body.

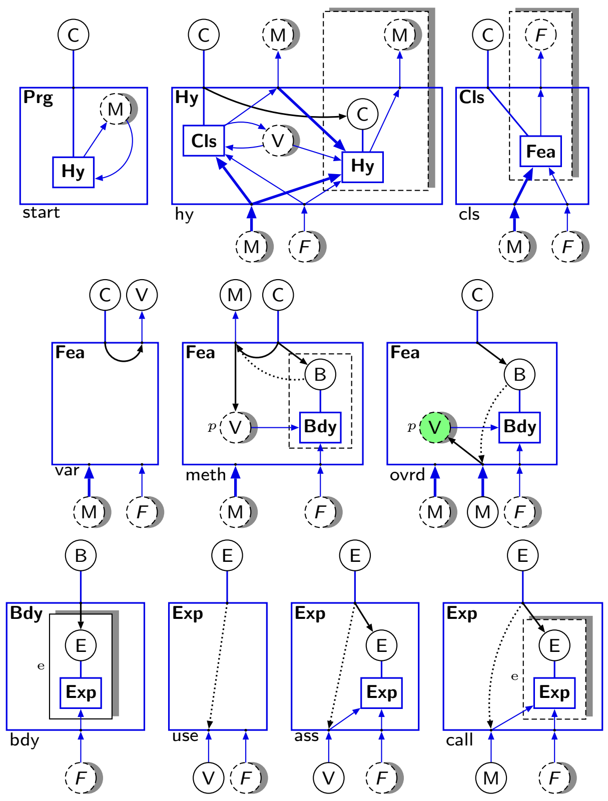

Fig. 9. Rules of the adaptive star grammar PG defining program graphs

Figure 10 shows parts of a derivation of the program graph shown in Figure 1 with PG. We simplify the drawing of edges as follows: A pair of counter-parallel edges “$\rightleftarrows$” is drawn as a single line “$\overline{}$”, and a pair of parallel edges of the form “$\rightrightarrows$” is drawn as a single arrow “$\longrightarrow$”.

The class hierarchy is derived in the first row. Exponents of the rules (if present) indicate how many clones are made of the multiple subgraphs and nodes on the right-hand side; for border nodes, it is easy to see how many clones have to be made, and how their labels have to be specialized. Classes Cell and Recell will introduce three and two features, resp.; the methods are visible in both classes, but the variables introduced are only visible in the defining class and

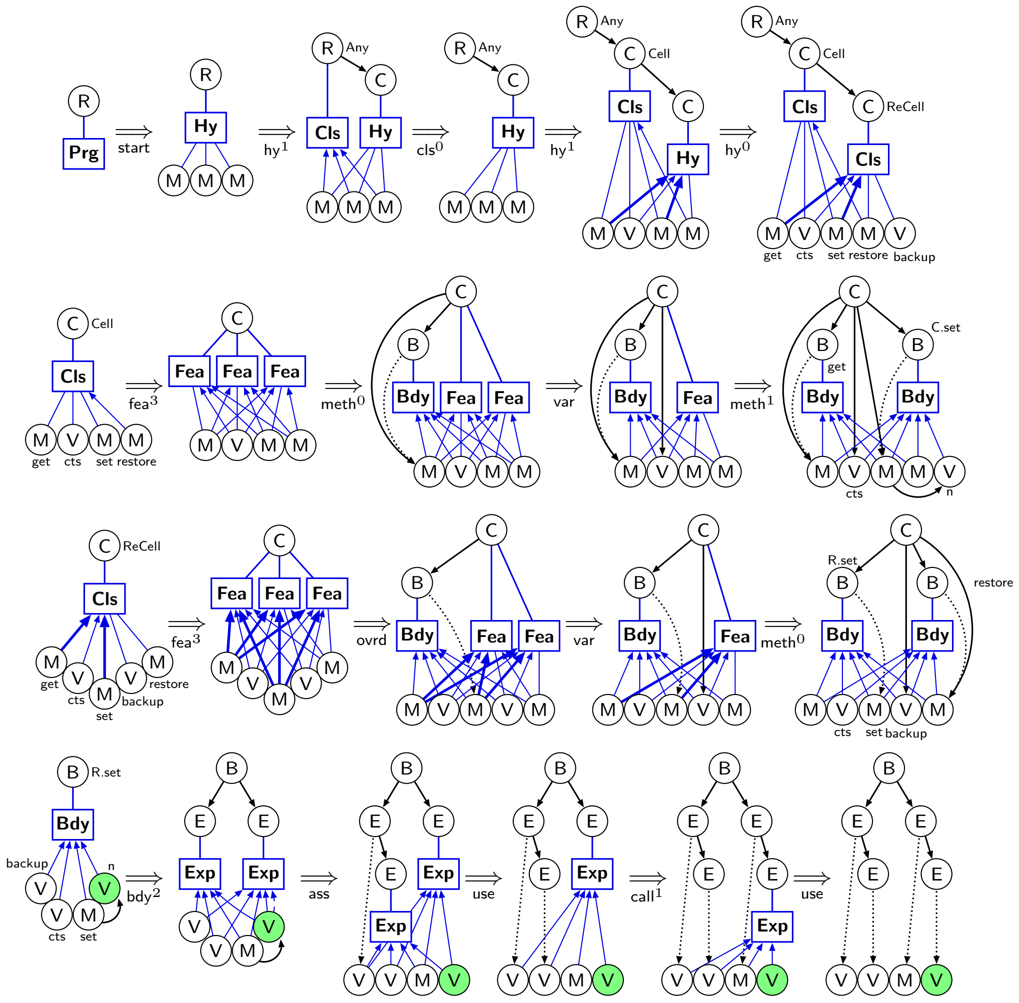

Fig. 10. Deriving the program graph of Figure 1 with PG

in its subclasses so that the variable backup in ReCell will not be visible in Cell. The methods defined in Cell are overridable in ReCell.

The features get, backup, and restore of the class Cell are introduced in the second row, and the features of the class ReCell are derived in the third row: the variable backup and the method restore are introduced, and the method set of Cell is overridden. The last row shows a derivation of the body overriding the method set of class Cell in ReCell.

The derivations in rows one to three can be combined to one big derivation by embedding. However, the start graph of the last row *cannot* be embedded into the final graph of the derivation in the third row. This is because the rule ovrd does not make the parameter n (drawn in grey) of the signature of set visible in the overriding body. The parameter is needed to derive the body, and it should be visible in it. This reveals one of two problems in the grammar, which cannot be overcome with adaptive star grammars.

Theorem 4.1. *A graph G is in $\mathcal{L}(\mathsf{PG})$ iff it satisfies Properties 2.2.(1–5,7).*

Proof Sketch. "$\Rightarrow$": Inspection of the rules (as done in Example 4.2 and Figure 8 above) shows that the border nodes of variables do indeed play the roles given to them. Using these invariants, it can be shown by induction over the structure of rules that every $G \in \mathcal{L}(\mathsf{PG})$ satisfies Properties 2.2.(1–5,7).

"$\Leftarrow$": Let G satisfy Properties 2.2.(1–5,7). We construct a derivation

$$Z = H_0 \underset{\mathsf{PG}}{\Longrightarrow} H_1 \underset{\mathsf{PG}}{\Longrightarrow} \cdots \underset{\mathsf{PG}}{\Longrightarrow} H_n$$

so that there are injective morphism $h_i \colon \bar{H}_i \to G$, where $\bar{H}_i$ is the terminal subgraph of H_i with all nodes that are reachable from the root class via edges of types $\{\downarrow, \dot{\downarrow}\}$, for $0 \leqslant i \leqslant n$.

By Fact 2.1, G is a collapsed tree that has a class, say c, as its root, which is also the root of a spanning tree induced by $\downarrow$-edges (Property 2.2.3). The unique class in Z is then mapped onto c. The number, say s, of direct subclasses of c determines the instance $\mathsf{hy}\frac{h}{s}$ that must be applied to $\hat{c}$ so that the clones of the C-nodes in H_1 can be mapped to the subclasses of c in G so that they can be extended to an injective morphism $h_1 \colon \bar{H}_1 \to G$. (The number of clones for the border nodes in Z and $\mathsf{hy}\frac{h}{s}$ are left open for the moment.) The construction of further graphs $\bar{H}_i$ can be continued in the same way. (See the program graph in Figure 1 as an example.) When a graph $\bar{H}_n$ has been constructed so that h_i is injective and surjective, no variables are left in H. Then all the open multiple border nodes in graphs $H_0, \ldots, H_{n-1}$ can be determined by considering the number of their clones as multiplicity variables, the values of which are determined by the rules used in the derivation. In the rule used in the first step, for instance, the multiplicity of the multiple M-node in Z is given as the sum of the multiplicities of all declared multiple M-nodes in $\mathsf{hy}\frac{h}{s}$, which in turn equals the multiplicity of the multiple F-node in $\mathsf{hy}\frac{h}{s}$. The multiplicity for the overridable M-node in $\mathsf{hy}\frac{h}{s}$ equals 0, and the multiplicity of the multiple C-node in $\mathsf{hy}\frac{h}{s}$ is determined by the rule that has been applied to **Cls**. The process of finding equations for the multiplicities of nodes yields a unique solution, because the equations satisfy the invariants on border nodes. The solutions for the multiple nodes define complete instances for all multiple nodes in the graphs H_i and in the rules of the derivation. Thus H_n is in $\mathcal{L}(\mathsf{PG})$, so that h_i is an isomorphism for $\hat{H}_n = H_n$, and $H_n \cong G$.

The proof constructs a derivation for a given graph. The construction is unique up to isomorphism so that we get the following:

Corollary 4.1. *PG is unambiguous.*

The grammar PG falsely derives some graphs that are not program graphs. In a graph $G \in \mathcal{L}(\mathsf{PG}) \setminus \mathcal{P}$, a class may contain several bodies that override the same method, method bodies that access a wrong number of parameters, and method calls with a wrong number of actual parameters. Let us discuss why adaptive star grammars fail to describe two properties of program graphs.

Property 2.2.6. In rule ovrd, the parameters of the method m being overridden cannot be made visible in its body. Parameters are only visible in the body of their first definition, so they are not among the clones of the F-node in that rule. (See the overriding of the method set in class ReCell discussed in Example 4.2.)

We could pass around all parameters of all methods (not in the role "visible", but in their role as "parameters"). Then, we had to select the parameters of m to be passed on to its body. We thus have to distinguish the parameters of m from those of other visible methods. However, the number of visible methods is unbounded, whereas our supply of edge sorts is finite. So this is not possible. Alternatively, we could generate copies of the formal parameters for every overridden body. But then we must know how many formal parameters m has. Again, this information cannot be made available.

Property 2.2.9. In rule call, the number of actual parameters needs not match the formal parameters of the method being called. As in the previous case, we would need access to the parameters of the method being called, or need to know their number in order to assure parameter correspondence.

These considerations lead to the following

Conjecture 4.1. *There is no adaptive star grammar Γ with $\mathcal{L}(\Gamma) = \mathcal{P}$.*

5 Conditional Adaptive Star Grammars

To overcome the deficiencies of adaptive star grammars, we extend adaptive star rules with *application conditions.* This has been proposed for general (DPO) graph transformation rules in [11], and has been discussed informally for adaptive star grammars in [8].

Definition 5.1 (Conditional Adaptive Star Replacement). An *application condition A* for an adaptive star rule $L ::= R$ is one of the following: *(i)* a *positive condition* C or *(ii)* a *negative condition* $\neg C$ where $C \in \ddot{\mathcal{G}}$, or *(iii)* a *clone condition* $\forall x : A'$ where x is a multiple node in L, and A' is an application condition for L wherein the node x occurs as a singular node that carries the same label as in L. The graphs in an application condition may contain further nodes from L, which carry the same label and are singular or multiple in both A and L. If $A_1, \ldots, A_n$ are application conditions for L, $r = A_1 \wedge \cdots \wedge A_n \parallel L ::= R$ is a *conditional adaptive star rule.* If $n = 0$, the rule r is written without the symbol "$\parallel$", like an unconditional rule.

In a *clone* $\ddot{r} = \ddot{A}_1 \wedge \cdots \wedge \ddot{A}_n \parallel \ddot{L} ::= \ddot{R}$ of a conditional rule r, all multiple nodes have as many clones in $\ddot{A}$ as in $\ddot{L}$. Let $m : \ddot{L} \to Y$ be a match of $\ddot{r}$ with a variable Y in some host graph G. We define recursively over the structure of application conditions in which cases m *satisfies* an application condition A, written $m \vDash A$:

- $m \vDash C$ if m can be extended to C;
- $m \vDash \neg C$ if m cannot be extended to C;
- $m \vDash \forall x : A$ if $m \vDash A(x/x')$ for every clone x' of x, where $A(x/x')$ is obtained from A by renaming x to x'.

If $m \vDash \ddot{A}_i$ for $1 \leqslant i \leqslant n$, the star replacement $G[Y/_m\ddot{R}]$ is a *conditional star replacement*, and we write $G \overset{c}{\Longrightarrow}_{\ddot{r}} H$.

When drawing conditional rules, as in Figure 11, we indicate shared nodes of application conditions and left-hand sides of conditional rules by attaching the same letters to them.

Definition 5.2 (Conditional Adaptive Star Grammar). Let $\mathcal{C}$ be a finite set of conditional adaptive star rules. Then $\Gamma = \langle \ddot{\mathcal{G}}(\ddot{\mathcal{X}}), \ddot{\mathcal{X}}, \mathcal{C}, Z \rangle$ is a *conditional adaptive star grammar* over if $Z \in \mathcal{X}$.

Let $\ddot{\mathcal{C}}$ denote the set of all possible clones of a set $\mathcal{C}$ of conditional adaptive star rules. Then Γ generates the language

$$\ddot{\mathcal{L}}(\Gamma) = \{G \in \mathcal{G} \mid Z \overset{c}{\Longrightarrow}{}^{*}_{\ddot{\mathcal{C}}} G\}$$

Example 5.1. [Conditional Adaptive Star Grammar for Program Graphs] Figure 11 shows the rules of the conditional adaptive star grammar PG_c, which refines the adaptive star grammar PG of Example 4.2 as follows. The variables in PG_c are attached to the border nodes used in PG, and may be attached to two additional sets of nodes, see Figure 12: Outgoing dashed edges $\dashrightarrow$ represent the formal parameters *contained* in variables named **Hy**, **Cls**, and **Fea**, and ingoing dashed edges represent the formal parameters *known* in a variable. The rules make that all formal parameters contained in the features, classes and hierarchies of the program are known to every variable.

In rule call, the positive condition on nodes m and p (where p is multiple as it occurs inside a multiple subgraph) requires that the clones of p are formal parameters of m, and the negative clone condition on nodes m and o forbids, for every other clone o' of o, i.e., for every other parameter known in the program,

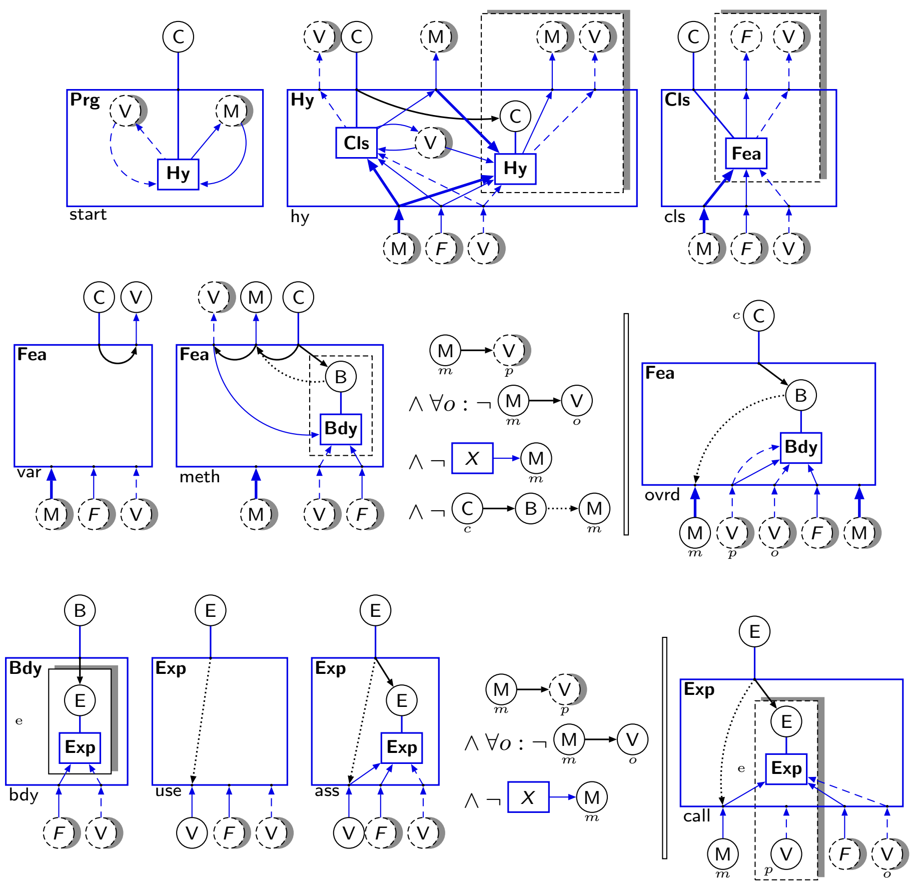

Fig. 11. Rules of the conditional adaptive star grammar $\mathsf{PG_c}$ defining program graphs

that o' is a parameter of m. Thus the nhclones of p are all parameters of m. The remaining condition forbids m to be a declared node of any variable named $X \in \Sigma_v$ to m. This makes sure that the parameters of m have been generated before rule call is applied. Since the multiple subgraph contains the formal parameter p as well as the variable named **Exp** generating the actual parameters, this makes

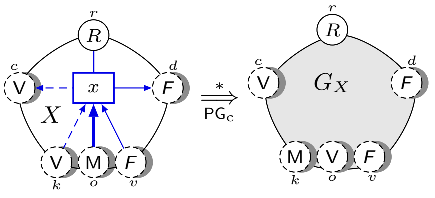

Fig. 12. Variables and derivations in $\mathsf{PG_c}$

sure that call will generate an actual parameter for every formal parameter of m. Thus Property 2.2.9 is respected.

In rule ovrd, the first three application conditions (which equal that of call) make sure that the clones of p are all formal parameters of m. These parameters are not only made known to the overriding body of m, but also made visible to it so that they may be accessed as variables in use and ass. Thus Property 2.2.6 is respected. The fourth application condition makes sure that no other method body contained in the current class c does override the same method m; this guarantees Property 2.2.8.

In Figure 13, we show some steps of a derivation with $\mathsf{PG_c}$ that could eventually derive the program graph in Figure 1. The grey region contains nodes representing the declarations of get, n, backup, and restore. A pair of counter-parallel edges “⇠⇢” is drawn as a single line “- - - -”.

Note that rule meth, which generates the definition of set in class Cell makes the parameter n visible, as a parameter, to the entire program.

When the rule ovrd is applied to the method set, n is made visible as a variable inside its body. The other part of the applicability condition holds as well: Class ReCell does not contain another body overriding set, and no variable has m as a declared border node (but just as a visible border node of **Bdy** and an overridable border node of **Fea**). Note that in class ReCell, the method set cannot be overridden by another body since this would violate the application condition of ovrd. Now the derivation in the last row of Figure 10 can be inserted

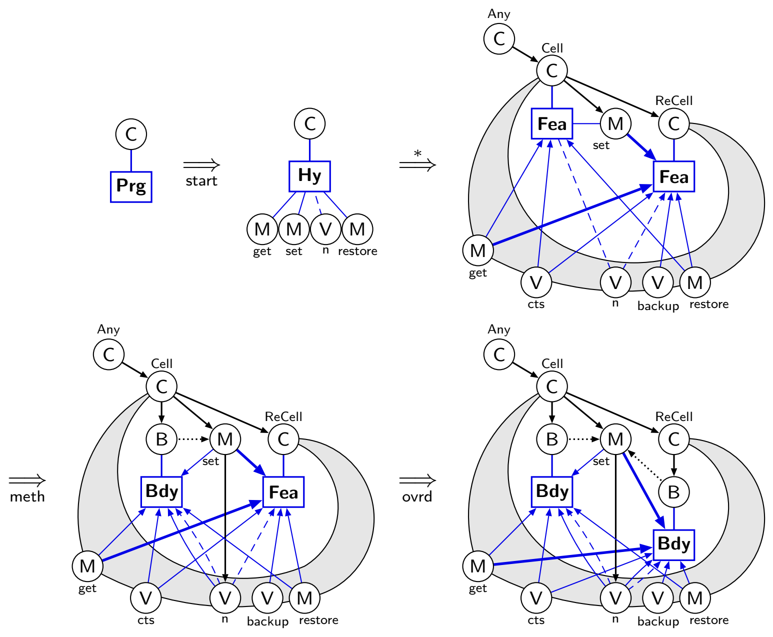

Fig. 13. Deriving the program graph of Figure 1 with $\mathsf{PG_c}$

for the body of set in ReCell because n is present. In that derivation, in the step using rule call, the application condition of $\mathsf{PG_c}$ guarantees that exactly one expression will be generated as an actual parameter since method set has one formal parameter.

Definition 5.3 (Complete Node). Consider a graph $G \in \mathcal{G}(\mathcal{X})$ and a conditional adaptive star grammar Γ.

A node $v \in \dot{G}$ is called *complete* wrt. structural edges if for every derivation $G \overset{\mathrm{c}}{\Longrightarrow}{}^{*}_{\Gamma} H$, v is incident to the same terminal edges in H as it was in G.

Fact 5.1. In graphs derived with $\mathsf{PG_c}$, M-nodes are complete wrt. structural edges if they are not declared nodes of any variables.

Proof Sketch. By inspection of the right-hand sides of the rules for these variables in $\mathsf{PG_c}$, it is clear that structural edges are added only to declared nodes of these rules' left-hand sides.

According to this fact, application conditions over structural edges can safely be checked as soon as the relevant nodes are only visible or overridable border nodes of variables. This is the case for the conditions concerning the parameters of methods.

Thus $\mathsf{PG_c}$ generates the program graph in Figure 1, and will not generate calls with mismatching parameters, nor with methods that are overridden twice in a class.

Theorem 5.1. $\mathcal{L}(PG_c) = \mathcal{P}$.

Proof Sketch. The proof is similar to that of Theorem 4.1.

"$\Rightarrow$": Inspection of the rules (as done in Example 5.1 and Figure 12 above) shows that the border nodes of variables do indeed play the roles given to them. Using these invariants, it can be shown by induction over the structure of rules that every $G \in \mathcal{L}(\mathsf{PG})$ satisfies all Properties 2.2.(1–9) of a program graph.

"$\Leftarrow$": Given a program graph $G \in \mathcal{P}$, we can construct a derivation according to the underlying structure (with edges of type $\downarrow$) first, before we determine the clones for border nodes according to the equations on the multiplicity variables. At last, it can be verified that the conditional rules ovrd and call satisfy their application conditions.

Application conditions do not sacrifice parseability of adaptive star grammars. Because, checking a condition, which consists of a positive and negative terminal graph, is always decidable. In contrast to simple adaptive star rules, the matches of conditional adaptive star rules in a graph may have critical overlaps. The application condition of one rule may contradict the application application condition of another rule. Consider, e.g., the node ReCell in the rightmost graph in the top row of Figure 13. The rule ovrd matches every **Fea** node in reCell. However, if the match includes the same method (get or set), then the application of the rule to one feature would disable the other application, due to the negative application concerning unique implementation. The critical pair analysis for graph transformation rules (as implemented in the AGG-system) applies to conditional graph transformation rules; it might be used to analyze conflicts in conditional adaptive star rules if we can extend it to multiple nodes.

6 Conclusions

In this paper, we have attempted to define the well-known class of program graphs [14] by graph grammars. This seems to be impossible with star grammars and even adaptive star grammars [6, 5], whereas it can be done with the conditional adaptive star grammars that have been introduced informally in [9, 8]. A richer class of program graphs, featuring more general visibility rules, contextual rules for abstract methods and classes, control flow in method bodies, and static typing of variables and methods has been specified in [9] by conditional adaptive star grammars as well.

There are too many kinds of graph grammars to relate conditional adaptive star grammars to all of them. So we restrict our discussion to approaches that aim at a similar application. Context-embedding rules [15] extend hyperedge-replacement grammars by rules that add a single edge to an arbitrary graph pattern. They are used to define and parse diagram languages and are not powerful enough to define models like program graphs. Graph reduction grammars [2] have been proposed to define and check the shape of data structures with pointers. The form of their rules is not restricted, but reductions with the inverse rules are required to be terminating and confluent, providing a backtracking-free parsing algorithm. It is an open question whether graph reduction grammars suffice to define program graphs.

A lot of work has to be done until we get a graph grammar mechanism that is useful for defining software models. Yet another problem is to convince software engineers that it is a practical benefit for their daily work!

First of all, graph grammars should be compared with the conventional software models, like UML diagrams. For instance, can such a model be derived from a grammar? Can at least parts of a model be obtained “automatically”? There is some indication that a class diagram specifying Properties 2.2.(1–3) of program graphs can be inferred from the rules of a (conditional) adaptive star grammar.

Even if conditional adaptive star grammars are powerful enough, their rules tend to be rather complicated, both to write and to read. So a more general challenge would be to come up with yet another graph grammar formalism that is easier to use, but enjoys many of the formal properties of (adaptive) star rules.

The proof of conjectures 3.1 and 4.1 poses the theoretical challenge to disprove membership in a class of graph languages. While there are at some concepts for star languages (e.g., the pumping lemma for the equivalent hyperedge replacement languages [13, 4]), nothing is known for (conditional) adaptive star languages.

Acknowledgments. I thank my favorite co-authors for their constructive reviews of this paper.

Special Thanks. *Danke, Hans-Jörg!* For your long-lasting support in form of scientific (and other) discussions, opportunities to visit conferences, never-ending supply of co-authors from your PhD students, and—last but not least—for fruit and cake in meetings!

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer, New York, 1996.
2. Adam Bakewell, Detlef Plump, and Colin Runciman. Specifying pointer structures by graph reduction. *Mathematical Structures in Computer Science*, 2009. Accepted for publication.
3. Bruno Courcelle. An axiomatic definition of context-free rewriting and its application to NLC rewriting. *Theoretical Computer Science*, 55:141–181, 1987.
4. Frank Drewes, Annegret Habel, and Hans-Jörg Kreowski. Hyperedge replacement graph grammars. In Rozenberg [16], chapter 2, pages 95–162.
5. Frank Drewes, Berthold Hoffmann, Dirk Janssens, and Mark Minas. Adaptive star grammars and their languages. Technical Report 2008-01, Departement Wiskunde-Informatica, Universiteit Antwerpen, 2008.
6. Frank Drewes, Berthold Hoffmann, Dirk Janssens, Mark Minas, and Niels Van Eetvelde. Adaptive star grammars. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *3rd Int'l Conference on Graph Transformation (ICGT'06)*, number 4178 in Lecture Notes in Computer Science, pages 77–91. Springer, 2006.
7. Frank Drewes, Berthold Hoffmann, Dirk Janssens, Mark Minas, and Niels Van Eetvelde. Shaped generic graph transformation. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformation with Industrial Relevance (AGTIVE'07)*, number 5088 in Lecture Notes in Computer Science, pages 201–216. Springer, 2008.
8. Frank Drewes, Berthold Hoffmann, and Mark Minas. Adaptive star grammars for graph models. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *4th International Conference on Graph Transformation (ICGT'08)*, number 5214 in Lecture Notes in Computer Science, pages 201–216. Springer, 2008.
9. Niels Van Eetvelde. *A Graph Transformation Approach to Refactoring.* Doctoral thesis, Universiteit Antwerpen, May 2007.
10. Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation.* EATCS Monographs on Theoretical Computer Science. Springer, 2006.
11. Hartmut Ehrig and Annegret Habel. Graph grammars with application conditions. In Grzegorz Rozenberg and Arto Salomaa, editors, *The Book of L*, pages 87–100. Springer, Berlin, 1986.
12. Joost Engelfriet and Grzegorz Rozenberg. Node replacement graph grammars. In Rozenberg [16], chapter 1, pages 1–94.

13. Annegret Habel. *Hyperedge Replacement: Grammars and Languages.* Number 643 in Lecture Notes in Computer Science. Springer, 1992.
14. Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.
15. Mark Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(2):157–180, 2002.
16. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations.* World Scientific, Singapore, 1997.
17. Andy Schürr, Andreas Winter, and Albert Zündorf. The PROGRES approach: Language and environment. In Gregor Engels, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. II: Applications, Languages, and Tools*, chapter 13, pages 487–550. World Scientific, Singapore, 1999.

Dr. Berthold Hoffmann
Fachbereich 3 – Informatik
Universität Bremen
D-28334 Bremen (Germany)
hof@informatik.uni-bremen.de
http://www.informatik.uni-bremen.de/~hof

Berthold Hoffmann became a colleague of Hans-Jörg Kreowski in 1978 at TU Berlin. After both of them moved to the University of Bremen in 1982, he took part in research activities of Hans-Jörg's group, particularly in the EC Working Groups COMPUGRAPH, APPLIGRAPH, and SEGRAVIS.