Electronic Communications of the EASST
Volume 26 (2010)

Manipulation of Graphs, Algebras and Pictures

Essays Dedicated to Hans-Jörg Kreowski on the Occasion of His 60th Birthday

Conditional Adaptive Star Grammars

Berthold Hoffmann

19 pages

Guest Editors: Frank Drewes, Annegret Habel, Berthold Hoffmann, Detlef Plump
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Conditional Adaptive Star Grammars

Berthold Hoffmann

Fachbereich Mathematik und Informatik, Universität Bremen
and
Forschungsbereich Sichere Kognitive Systeme, DFKI Bremen
Enrique-Schmidt-Straße 5, 28359 Bremen, Germany
hof@informatik.uni-bremen.de

Abstract: The precise specification of software models is a major concern in the model-driven design of object-oriented software. Models are commonly given as graph-like diagrams so that graph grammars are a natural candidate for specifying them. However, context-free graph grammars are not powerful enough to specify all static properties of a model. Even the recently proposed adaptive star grammars cannot capture all properties of object-oriented models. So we extend adaptive star rules by positive and negative application conditions to overcome these deficiencies without sacrificing parsing algorithms. It turns out that conditional adaptive star grammars are powerful enough to generate program graphs, a software model with rather complicated contextual properties.

Keywords: graph grammars; model definition; adaptive star grammar; application condition

1 Introduction

Model-driven design of object-oriented software aims at describing static structure, dynamic behavior, and gradual evolution of systems in a comprehensive way. Typically, a software model is a collection of graph-like diagrams, which is commonly 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. 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 check whether a model is valid, or not. However, neither star grammars (equivalent to the well-known hyperedge replacement grammars [Hab92, DHK97]), nor node replacement grammars [ER97] are powerful enough for our purpose. Even the recently proposed adaptive star grammars [DHJ+06, DHJM09] fail for certain some 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 for adaptive star rules have already been considered in [Eet07, DHM08].) As a case study, we consider a simple variant of program graphs, a language-independent model of object-oriented programs that has been devised for specifying refactoring operations on programs [MEDJ05]. Conditional adaptive star grammars capture all structural properties of these graphs.

The paper is structured as follows. In Section 2, we show how object-oriented programs can abstractly be represented as *program graphs*. Then we recall star grammars in Section 3, show how they define *program trees*, a sub-structure of program graphs, and discuss why they cannot define program graphs themselves. In Section 4, we therefore recall the *adaptive star grammars* devised in [DHJ+06, DHJM09]. Close inspection reveals that even this formalism fails to capture some 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 the UML. Formally, such diagrams can be defined as many-sorted graphs.

Definition 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 $\bar{\sigma}\colon \bar{G} \to \bar{\Sigma}$ *label* 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 drawing arrows in different styles, 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 [MEDJ05]. 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. Here we use a variant that is simplified wrt. [Eet07] in several ways:

1. In method bodies we just represent the *data flow*: use and update of variables, and method calls. The structure of statements and expressions is omitted.

2. We simplify the *visibility rules* for features: all methods are assumed to have global visibility (*public* in Java); variables are assumed to be visible in the declaring class and in its subclasses (*protected* in Java); parameters of a method are visible in its body.

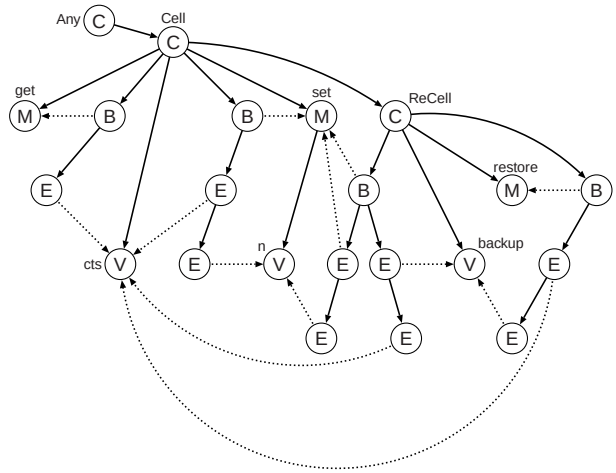
3. We ignore the *typing* of variables, parameters and return values of methods.

Even in this simplified form, program graphs are a good example for a software model. Their admissible shape is given by precise syntactic and contextual rules of object-oriented program-

```
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)
```

(a) A simple OO program



(b) The graph representing the program in Figure 1a

Figure 1: A program graph

ming languages. This makes it easy to check whether a definition of program graphs captures all properties of program graphs.

Example 1 (A Program Graph) *Figure 1a shows a simple object-oriented program from [AC96], for which the program graph is depicted in Figure 1b. The nodes of a program graph, drawn as circles, represent syntactic entities of a program: classes (C), variables (V), method signatures (M), method bodies (B), and expressions (E). Edges establish relations between entities: a solid arrow "↓" is pronounced "*contains*" , and a dashed arrow "⋮↓" 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 method 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 variable represents an* access *that either* uses *its value, or* assigns *the value of an expression to it.*

Definition 2 (Program Graph) A graph G is a *program graph* if it has the following properties:

P_1. Its nodes $\dot{G}$ are labeled with the sorts $\{\mathsf{C},\mathsf{V},\mathsf{M},\mathsf{B},\mathsf{E}\}$, and its edges $\bar{G}$ are labeled with the sorts $\{\downarrow,\vdots\downarrow\}$.

P_2. There is a morphism that maps G to the *incidence graph* G_{inc} shown in Figure 2. In addition, the following conditions hold:

(a) A body contains at least one expression, and it implements exactly one method signature.

(b) An expression e refers to exactly one node, and that node is either a method or a

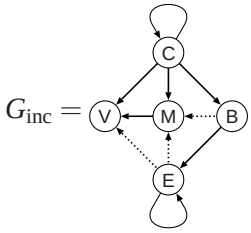


Figure 2: The incidence graph of program graphs

variable. If e refers to a variable v, it contains at most one expression (the value of which shall be assigned to v).

P_3. The subgraph $\bar{G}$ induced by $\downarrow$-edges of G is a spanning tree of G; the root of $\bar{G}$ is a class.

P_4. If an expression refers to a method m, m must be contained in some class of the graph.

P_5. If an expression e accesses a variable v contained in a class c, e must be a sub-expression of a body b that is contained in a sub-class of c.

P_6. If an expression e accesses a parameter p of a method m, e must be a sub-expression of a body that implements m.

P_7. If a method body b implements a method signature m, b must be contained in a sub-class of the class c containing m.

P_8. For every method signature m, every class contains at most one body implementing m.

P_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 $\mathscr{P}$.

The incidence graph in Figure 2 plays the role that type graphs play in algebraic graph transformation [EEPT06], and that graph schemata play in PROGRES [SWZ99]. Property P_4 defines the visibility of all methods as *public*, and Property P_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 3 A rooted, connected, acyclic graph is called a *collapsed tree*.

Lemma 1 *Program graphs are collapsed trees.*

Proof Sketch. The only (minimal) cycles in the incidence graph G_{inc} in Figure 2 are the two loops on the nodes labeled C and E, respectively. As there is a morphism from G to that incidence graph, this means that all cycles in G consist of containment edges. Hence, by Property P_3, there cannot be any cycles, because these cycles would occur in $\bar{G}$. Property P_3 implies connectedness; The root class of the spanning tree is the root of the program graph as well, because the incidence graph G_{inc} forbids references to classes. □

Program graphs can be specified by models, e.g., by UML class diagrams with logical OCL constraints. The incidence graph Figure 2 corresponds to a simple UML class diagram without subtyping. Properties P_2 (a) and (b) can be expressed as cardinality constraints for that class diagram. Property P_3 can be specified by requiring that "contains"-arcs are compositions, plus an additional OCL constraint assuring that the class hierarchy has a unique root. Properties P_4-P_9 can be specified by structural OCL constraints. For details, see [HM10].

3 Star Grammars

Star grammars are a special case of double pushout (DPO) graph transformation [EEPT06], and equivalent to hyperedge replacement grammars [Hab92, DHK97], a well-understood context-free kind of graph grammars. They are recalled just as a basis for the extensions defined in Section 4 and Section 5.

Definition 4 (Star) From now on we assume that the node sorts contain *nonterminal sorts* $\dot{\Sigma}_n \subseteq \dot{\Sigma}$ that define the *terminal node sorts* as $\dot{\Sigma}_t = \dot{\Sigma} \setminus \dot{\Sigma}_n$.

Consider a star-like graph X, with one center node c_X of sort $x \in \dot{\Sigma}_n$, and with some border nodes (of terminal sorts from $\dot{\Sigma}_t$) so that every border node is adjacent to c_X, and only to c_X. Then X is called a *star named x*. A star is *straight* if every border node is incident with exactly one edge.

A graph G is a *graph with stars* if no nodes named with nonterminals are adjacent to each other.[1] Let $\mathscr{X}$ denote the class of *stars*, $\mathscr{G}(\mathscr{X})$ the class of graphs with stars, and $\mathscr{G}$ be the class of graphs without stars (with node sorts from $\dot{\Sigma}_t$).

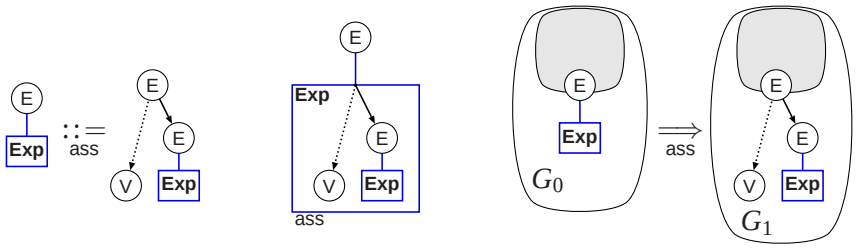
Definition 5 (Star Replacement) A *star rule* is written $L ::= R$, where the *left-hand side* $L \in \mathscr{X}$ is a straight star and the *replacement* is a graph $R \in \mathscr{G}(\mathscr{X})$ that contains the border nodes of L.

A star 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 the center node c_Y.

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

Example 2 (Star Replacement) *Figure 3a shows a star rule $L ::= R$ for an assignment expression. The center nodes of stars are drawn as boxes enclosing their name. We shall draw such a star rule as in Figure 3b, by "blowing up" the box of the center node on its left-hand side, and placing the new nodes and edges of the right-hand side inside this box. A star rule can be represented as it is drawn, as a single* rule graph *wherein one star is distinguished as the rule's left-hand side. This way, graph operations can be applied to star rules as well. Figure 3c shows a schematic star replacement $G_0 \Rightarrow_{\mathsf{ass}} G_1$ using this rule.*

[1] Then all these nodes are centers of stars.



(a) A star rule (b) Boxed form (c) A schematic star replacement

Figure 3: Star replacement

Definition 6 (Star Grammar) $\Gamma = \langle \mathscr{G}(\mathscr{X}), \mathscr{X}, \mathscr{R}, Z \rangle$ is a *star grammar* with a *start star* $Z \in \mathscr{X}$. The *language* of Γ is obtained by exhaustive star replacement with its rules, starting from the start star:

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

Example 3 (Star Grammar for Program Trees) *Figure 4 shows star rules that generate program trees. The rules define a star grammar* PT *according to the following convention: The left-hand side of the first rule indicates the start star, a star named* ***Prg*** *with a class as a border node in this case. The sorts used in the rules define the sorts of the grammar.*

In the rules, we use abbreviations for certain common constructions. Boxes drawn with dashed lines and/or a shade around a subgraph of the right-hand side indicate that a varying number of these subgraphs can be generated: a solid box with a shade indicates that the subgraph may have $n \geqslant 1$ instances, so rule bdy *may generate an arbitrary non-empty set of expressions; a dashed box with a shade indicates that the subgraph may have $n \geqslant 0$ instances, so rule* hy *may generate an arbitrary, possibly empty, set of sub-classes (rules* sig*,* impl*, and* call *show further examples); finally, a dashed box without shade indicates an optional subgraph that may have* 1 *or* 0 *instances,*

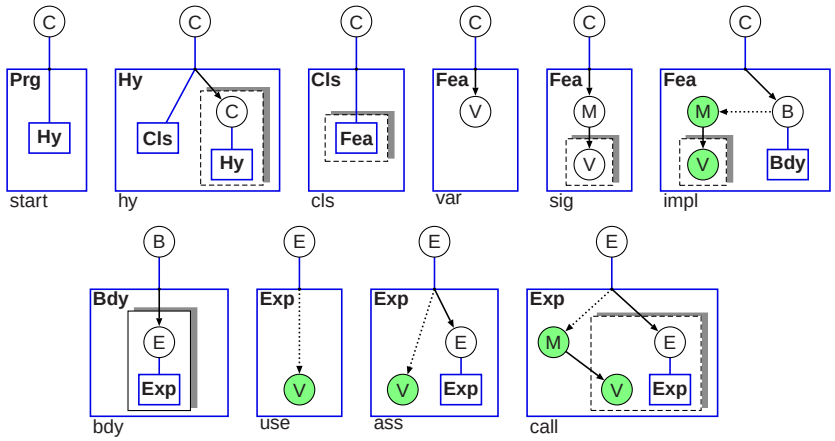


Figure 4: The rules of the star grammar PT generating program trees

so rule meth in Figure 6 on page 10 may derive a method body, or not.

Note that generic subgraphs could be implemented by using auxiliary nonterminals and star rules. In our examples, we just assume that we may use rule instances r^i *of a rule* r *wherein* i *instances of the respective subgraph have been made.*

Green 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: These are the method signatures generated in impl *and* call*, and the variables accessed in* use *and* ass*. (In black-and-white printing, these nodes appear to be grey.)*

Inspection of the rules in PT reveals the following.

Fact 1 *$\mathscr{L}(PT)$ is a language of trees.*

The language of PT is closely related to program graphs.

Definition 7 (Unraveling) Consider a program graph $G \in \mathscr{P}$ and define, for every method node $m \in \dot{G}$ (with $\dot{\sigma}_G(m) = \mathsf{M}$), its *signature tree* $M_G(m)$ as the subgraph of G induced by m and all variable nodes contained in m.

The *unraveling* $\hat{G}$ of G is then obtained by redirecting in G, for every reference edge $e \in \bar{G}$ (with $\bar{\sigma}_G(e) = \vdots$), its target to a new variable node if $\dot{\sigma}_G(t_G(e)) = \mathsf{V}$, and to a fresh copy of the signature tree $M_G(t_G(e))$ if $\dot{\sigma}_G(t_G(e)) = \mathsf{M}$, respectively.

Let $\hat{\mathscr{P}} = \{\hat{G} \mid G \in \mathscr{P}\}$ denote the unravelings of program graphs.

Lemma 2 $\hat{\mathscr{P}} \subsetneq \mathscr{L}(PT)$.

Proof Sketch. ($\hat{\mathscr{P}} \subseteq \mathscr{L}(PT)$). Consider some program graph $G \in \mathscr{P}$. Then its unraveling $\hat{G}$ still has Properties P_1–P_3 of program graphs: No new labels are added so that $\hat{G}$ satisfies Property P_1; the redirection of edges does not change incidences so that Property P_2 is preserved, and the underlying spanning tree $\bar{G}$ is not changed in $\hat{G}$. Moreover, $\hat{G}$ is a tree since unraveling redirects all reference edges to unique new variable nodes and signature trees, respectively. Using these properties, it can be shown by a straight-forward induction over derivations with PT that $\hat{G} \in \mathscr{L}(\mathsf{PT})$.

($\hat{\mathscr{P}} \neq \mathscr{L}(PT)$. Rules impl, use, ass, and call allow to generate implementations and calls of methods, or accesses to variables even if no declaration of a variable or method has been generated in the tree by rules var or sig) Such a tree cannot be the unraveling of a program graph, which must satisfy Properties P_4–P_9. □

Star grammars are context-free in the sense of B. Courcelle [Cou87]. This suggests that their generative power is limited. Indeed, we have the following

Theorem 1 *There is no star grammar Γ with $\mathscr{L}(\Gamma) = \mathscr{P}$.*

Proof Sketch. (By contradiction.) Consider program graphs G_n containing only one class, one method signature, and one body. The method signature contains n parameter nodes $p_1, \ldots, p_n$, and the body contains n expression nodes $e_1, \ldots, e_n$ with $n-1$ sub-expressions $e_{i_1}, \ldots, E_{i_{n-1}}$ each.

Now, consider the following additional requirements:

1. For every e_i, the sub-expressions $e_{i_1},\ldots,e_{i_{n-1}}$ access pairwise distinct parameters in $\{v_1,\ldots,v_n\}$, leaving out exactly one.

2. For every v_i, there is exactly one e_j such that v_i is not accessed by any of its sub-expressions, and for distinct e_j, e_k, these non-accessed parameters are distinct.

Let $\mathscr{P}^2 = \{G_n \mid n < 0\}$ be the class of such program graphs. Clearly, $\mathscr{P}^2 \subseteq \mathscr{P}$. A graph G_n has n^2+n+3 nodes and n^2+n+2 edges. So the size of graphs in $\mathscr{P}^2$ grows quadratically. By [DHJM10, Theorem 2.8], star grammars are equivalent to hyperedge replacement grammars (HR grammars, for short). Thus $\mathscr{P}$ can also be generated by a HR grammar. Moreover, requirements (1) and (2) are easily expressible in first-order logic, and thus also in monadic second order logic. Then, by [Cou90, Theorem 4.4(1)], a HR grammar generating $\mathscr{P}$ can be restricted to a HR grammar generating $\mathscr{P}^2$. This, however, contradicts the linear growth theorem 2.6 in [Hab92] which says that the size of graphs in a HR language grows only linearly. □

4 Adaptive Star Grammars

We make the left-hand sides of star rules *adaptive* wrt. the numbers of border nodes, as proposed in [DHJ+06]. It has been shown in [DHJM09] that this extends the generative power of star grammars. Formally, adaptation is defined by cloning.

Definition 8 (Singular and Multiple Nodes) We assume that the sorts $\Sigma = \langle \dot{\Sigma}, \bar{\Sigma} \rangle$ are given so that the terminal node sorts $\dot{\Sigma}_t$ contain a set $\ddot{\Sigma}_t$ of *multiple* sorts so that every remaining *singular sort* $s \in \dot{\Sigma}_t \setminus \ddot{\Sigma}_t$ has a unique multiple sort $\ddot{s} \in \ddot{\Sigma}_t$, and vice versa.

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

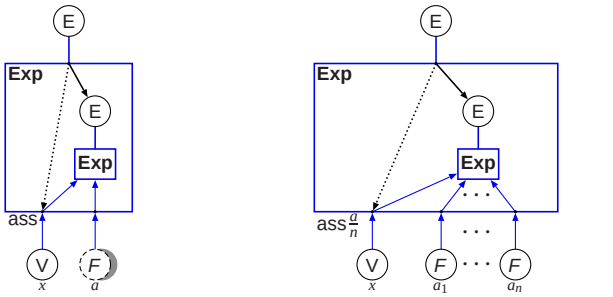
A star rule $L ::= R$ is called *adaptive* if $L \in \ddot{\mathscr{X}}$ and $R \in \ddot{\mathscr{G}}(\ddot{\mathscr{X}})$.

Definition 9 (Cloning) Let G be a graph in $\ddot{\mathscr{G}}(\ddot{\mathscr{X}})$. For a multiple node v that is labeled with $\ddot{\ell} \in \ddot{\Sigma}_t$, and incident with the edges $e_1,\ldots,e_n$ ($n \geqslant 0$), G^v_k denotes the graph in which v is replaced by $k \geqslant 0$ singular nodes $v_1,\ldots,v_k$ that are labeled with ℓ, and every edge e_i is replaced by copies $e_{i,1},\ldots,e_{i,k}$ so that $s_{G'}(e_{i,j}) = s_G(e_i)$, $t_{G'}(e_{i,j}) = t_G(e_i)$, and $\sigma_{G'}(e_{i,j}) = \sigma_G(e_i)$ for $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant k$. A node v_i is called a *clone of* v, and G^v_k is called an *instance* of G.

For a graph $G \in \ddot{\mathscr{G}}(\ddot{\mathscr{X}})$, a function $\mu : \dot{G} \to \mathbb{N}$ is a *multiplicity* if it maps singular nodes to 1. Then G^μ is the instance of G wherein every multiple node v has $\mu(v)$ clones.

Example 4 (Adaptive Star Cloning, and Label Specialization) *The star rule* ass *in Figure 5a is adaptive: its node* a *is multiple, and shall match a set of* $n \geqslant 0$ *singular nodes in the host graph that are accessible in the expression. In Figure 5b, a schematic view of the rule instances* ass^a_n *is given, for* $n \geqslant 0$.

The abstract sort F *of nodes* a *and* a_i *is a placeholder for the concrete sub-sorts* V *and* M. *(*F *stands for* feature.*) Before applying the rule instance* ass^a_n, *each of the labels* F *is specialized either to* V *or* M. *As with generic subgraphs, a star rule with abstract sorts is just an abbreviation*



(a) An adaptive star rule (b) The star rules obtained by cloning

Figure 5: Cloning of adaptive rules

for a set of star rules wherein these abstract sorts are replaced with any combination of their concrete sub-sorts.

Definition 10 (Adaptive Star Grammar) Let $\Gamma = \langle \ddot{\mathscr{G}}(\ddot{\mathscr{X}}), \ddot{\mathscr{X}}, \mathscr{R}, Z \rangle$ be a star grammar over adaptive stars and graphs. Then Γ is called *adaptive* if $Z \in \mathscr{X}$ (i.e., has no multiple nodes).

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

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

The set of star rules $\ddot{\mathscr{R}}$ generated from a set of adaptive star rules is infinite if at least one of the adaptive star rules contains a multiple node. It has been shown in [DHJM09] that this gives adaptive star grammars greater generative power than grammars based on hyperedge [Hab92] or node replacement [ER97], but but they still admit a parsing algorithm [DHJ+06].

Example 5 (Adaptive Star Grammar for Program Graphs) *The adaptive star rules in Figure 6 define an adaptive star grammar PG that systematically extends the program tree grammar PT of Figure 4.*

*As for star rules, we allow generic subgraphs in rules in order to abbreviate repetitions. The adaptive star rule hy has instances hy^i with i instances of the **Hy**-star, and each of them is source of an instance of a multiple M-node. The instance hy^i is then subject to cloning. Again, generic subgraphs could be implemented by auxiliary nonterminals and auxiliary adaptive star rules.*

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 7 we show the general form of stars 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; it is labeled by the* root sort R_x *of x. ($R_{\mathbf{Exp}} = E$, $R_{\mathbf{Bdy}} = B$, and $R_x = C$ for $x \in \{\mathbf{Prg}, \mathbf{Hy}, \mathbf{Cls}, \mathbf{Fea}\}$.) 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*

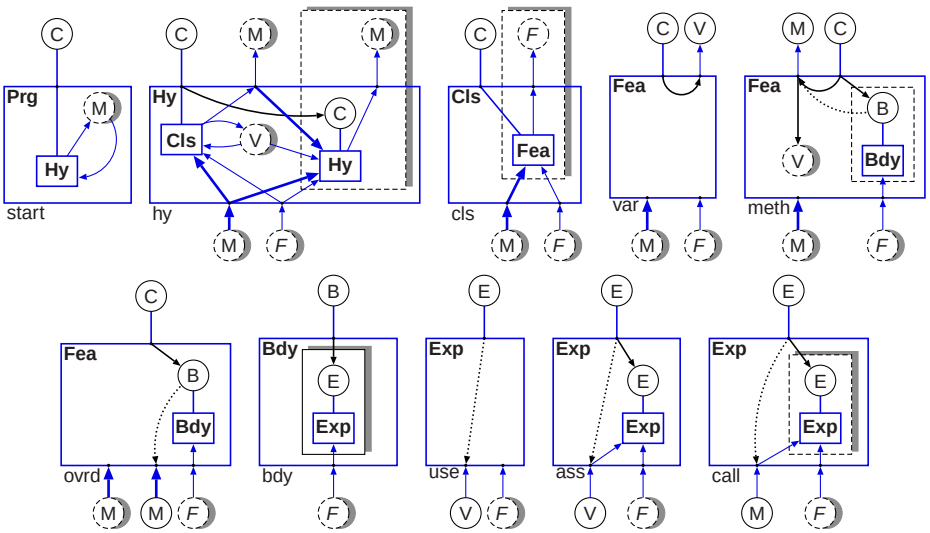


Figure 6: Rules of the adaptive star grammar PG defining program graphs

have several roles in X and G_X: every feature declared by X is also visible in X, and overridable methods are visible as well so that some clones d and v, and some clones of v and o in X may be identified. On the left-hand side of rules, the clones of d, v and o in a star X have to be distinct (as X is 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 6 extend the rules of Figure 4 by adding border nodes to stars 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.*

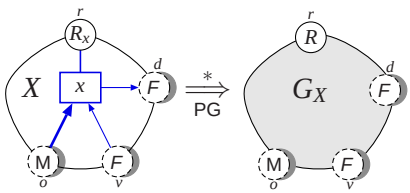


Figure 7: Stars and derivations in PG

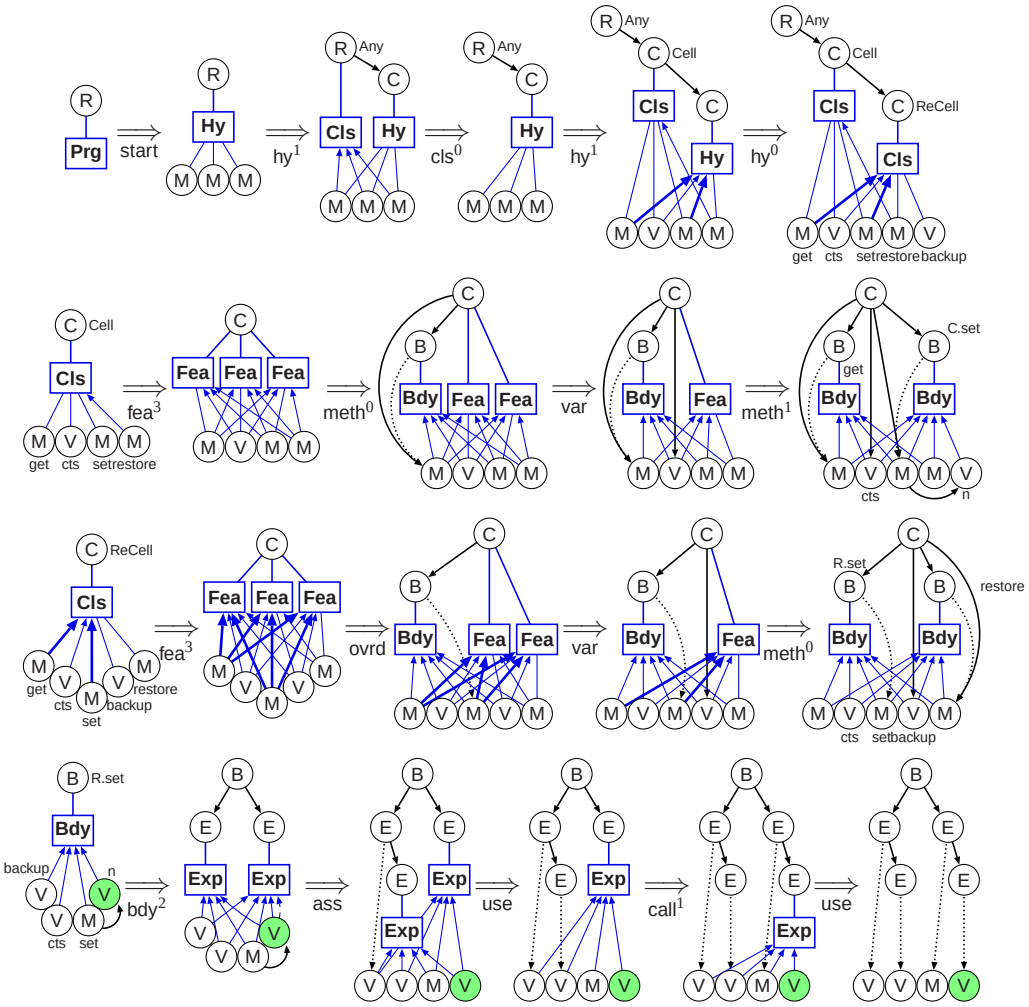


Figure 8: Deriving the program graph of Figure 1b with PG

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 new body.*

Figure 8 shows parts of a derivation of the program graph shown in Figure 1b with PG. We simplify the drawing of edges as follows: A pair of counter-parallel edges "⇄" is drawn as a single line "——", and a pair of parallel edges of the form "⇉" is drawn as a single arrow "⟶".

The class hierarchy is derived in the first row. 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 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 green, or grey, resp.) 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 2 *Every graph G is in $\mathscr{L}(\mathsf{PG})$ satisfies Properties P_1–P_5, and P_7.*

Proof Sketch. Inspection of the rules (as done in Example 5 and Figure 7 above) shows that the border nodes of stars do indeed play the roles given to them. Using these invariants, it can be shown by induction over the structure of rules that every $G \in \mathscr{L}(\mathsf{PG})$ satisfies Properties P_1–P_5, and P_7. □

A graph $G \in \mathscr{L}(\mathsf{PG})$ need not satisfy the remaining properties of program graphs: a class in G may contain several bodies that override the same method, contradicting Property P_8, and a method may be called with any number of actual parameters, contradicting Property P_9. The reverse of this theorem does not hold either. In particular, a program graph $G \in \mathscr{P}$ cannot be derived by PG if it contains an overridden method m that accesses its parameters. In G, all bodies of m may access the parameters of m (by Property P_6), whereas in a graph $G \in \mathscr{L}(\mathsf{PG})$, this is not true for an overridden body of m. For this reason, the last sub-derivation in Figure 8, which overrides the method set, cannot be embedded into a big derivation of the program graph in Figure 1b.

Why is it so difficult to specify Property P_6 with an adaptive star grammar? In rule ovrd, the parameters of the method m being overridden cannot be made visible in its body, as they are not among the clones of the F-node in the rule.

We could pass around all parameters of all methods (not in the role "visible", but in a new role as "parameters"). Then, we had to select the parameters of m because only these should be visible 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 parameters for every overridden body. But then we must know how many parameters m has. Again, this information cannot be made available.

These considerations lead to the following

Conjecture 1 *There is no adaptive star grammar Γ with $\mathscr{L}(\Gamma) = \mathscr{P}$.*

5 Conditional Adaptive Star Grammars

To overcome the deficiencies of adaptive star grammars, we extend adaptive star rules by *application conditions*. This has already been discussed informally in [DHM08].

Definition 11 (Conditional Adaptive Star Replacement) Let $r = L ::= R$ be an adaptive star rule.

A *simple application condition* A for L can be constructed over a graph $C \in \ddot{\mathscr{G}}(\ddot{\mathscr{X}})$ if C is disjoint to L up to some border nodes of L, and if all multiple nodes of C appear in L, with the same sort. Then A may take one of the following forms: (i) if $A = C$, it is a *positive condition*; (ii) if $A = \neg C$, it is a *negative condition*; or, (iii) if $A = \forall_{x_1,\ldots,x_n} \neg C$ ($n > 0$) where $x_1, \ldots, x_n$ are multiple nodes in C, it is a *negative instance condition*.

If $A_1, \ldots, A_n$ are simple application conditions for L, $c = A_1 \wedge \cdots \wedge A_n \parallel L ::= R$ is a *conditional adaptive star rule*. (For $n = 0$, the rule r is written without the symbol "$\parallel$", like an unconditional rule.)

Let L^μ be an instance of the star L an adaptive star rule $r = L ::= R$ (for some multiplicity μ). A match $m\colon L^\mu \to G$ *satisfies* an application condition A, written $m \vDash A$, under one of the conditions below:

- $m \vDash C$ if m can be extended to a morphism $L^\mu \cup C^\mu \to G$;[2]
- $m \vDash \neg C$ if m cannot be extended to $L^\mu \cup C^\mu \to G$;[2]
- $m \vDash \forall_{x_1,\ldots,x_n} \neg C$ if, for every tuple $(y_1, \ldots, y_n)$ of instances of the multiple nodes $x_1, \ldots, x_m$, m cannot be extended to $L^\mu \cup C[x_1/y_1] \ldots [x_n/y_n] \to G$, where $C[x/y]$ is the copy of C wherein the node x (of sort $\ddot{\sigma}$, say) is replaced by a singular node y (of sort σ).

If $m \vDash A_i$ for $1 \leqslant i \leqslant n$, the star replacement $H = G[m(L^\mu)/_m R^\mu]$ is a *conditional star replacement*, and we write $G \overset{c}{\Longrightarrow}_c H$.

Application conditions for general graph transformation rules have been devised in [EH85]. Our application conditions are not nested as those considered in [HP09]. Furthermore they are in conjunctive normal form, and just allow to require the existence or non-existence of subgraphs. This is sufficient for our purpose.[3]

When drawing conditional rules, as in Figure 9, we indicate shared nodes of application conditions and left-hand sides of conditional rules by attaching the same letters to them.

Definition 12 (Conditional Adaptive Star Grammar) Let $\mathscr{C}$ be a finite set of conditional adaptive star rules. Then $\Gamma = \langle \ddot{\mathscr{G}}(\ddot{\mathscr{X}}), \ddot{\mathscr{X}}, \mathscr{C}, Z \rangle$ is a *conditional adaptive star grammar* over adaptive stars and graphs) if $Z \in \mathscr{X}$.

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

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

Example 6 (Conditional Adaptive Star Grammar for Program Graphs) *Figure 9 shows the rules of the conditional adaptive star grammar* PG_c*, which refines the adaptive star grammar* PG *of*

[2] We assume that the instances of multiple nodes in L and C are the same.

[3] The reader may wonder why we consider only negative, all-quantified instance conditions. It is easy to see that $\forall_{x_1,\ldots,x_n} C$ is equivalent to the condition C. Existential conditions $\exists_{x_1,\ldots,x_n} [\neg] C$ can be expressed by adding singular clones for the multiple nodes $x_1, \ldots, x_n$ to L, and requiring $[\neg] C$ just on these clones.

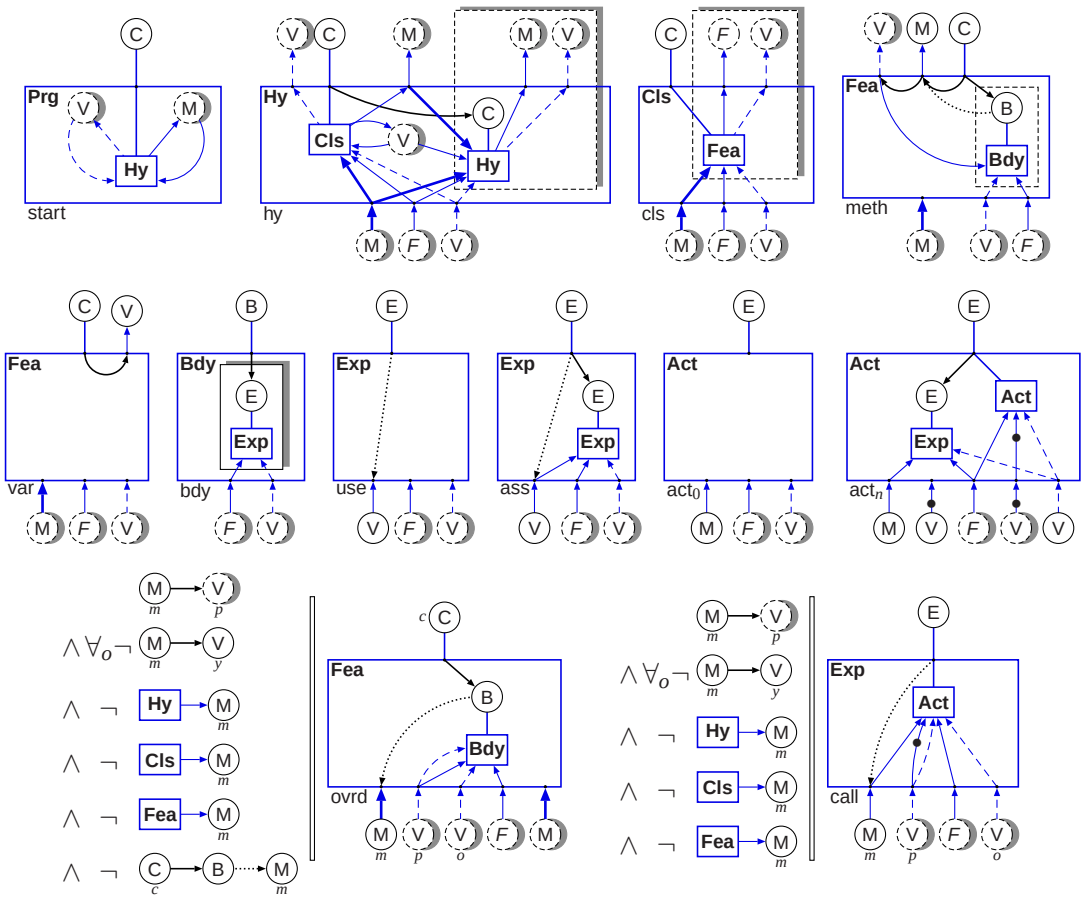


Figure 9: Rules of the conditional adaptive star grammar PG_c defining program graphs

Example 5 as follows. All stars in PG_c are attached to the border nodes used in PG, and may be attached to two additional sets of nodes, see Figure 10: Outgoing dashed edges ⇢ represent the parameters contained *in stars named* ***Hy***, ***Cls***, *and* ***Fea***, *and ingoing dashed edges represent the parameters* known *in a star. The rules make that all parameters contained in the features, classes and hierarchies of the program are known to every star.*

In rule call, the positive condition on nodes m and p requires that the clones of p are parameters of m, and the negative instance condition on node o forbids every other parameter known in the

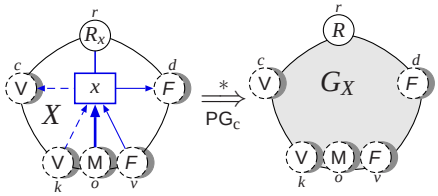


Figure 10: Stars and derivations in PG_c

program to be a parameter of m. Thus the clones of p are all parameters of m. The remaining three conditions forbid m to be a declared node of any star named ***Hy***, ***Cls***, *or* ***Fea***. *This makes sure that all parameters of m have already been generated (in the rules for* ***Fea***) *before rule* call *can be applied. Rule* call *generates a new nonterminal* ***Act*** *to which the parameters of m are connected by an edge* →•→. *In the rules for* ***Act***, *these edges are used to "count" the number of parameters while generating the corresponding actual arguments (by* ***Exp***). *Thus Property* P_9 *is respected.*

In rule ovrd, *the first five application conditions (which equal that of* call) *make sure that the clones of p are all parameters of m. These parameters are not only become known (as parameters) 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* P_6 *is respected. The sixth application condition makes sure that no other method body contained in the current class c does override the same method m; this guarantees Property* P_8.

In Figure 11, we show some steps of a derivation with PG_c *that could eventually derive the program graph in Figure 1b. 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 star 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*

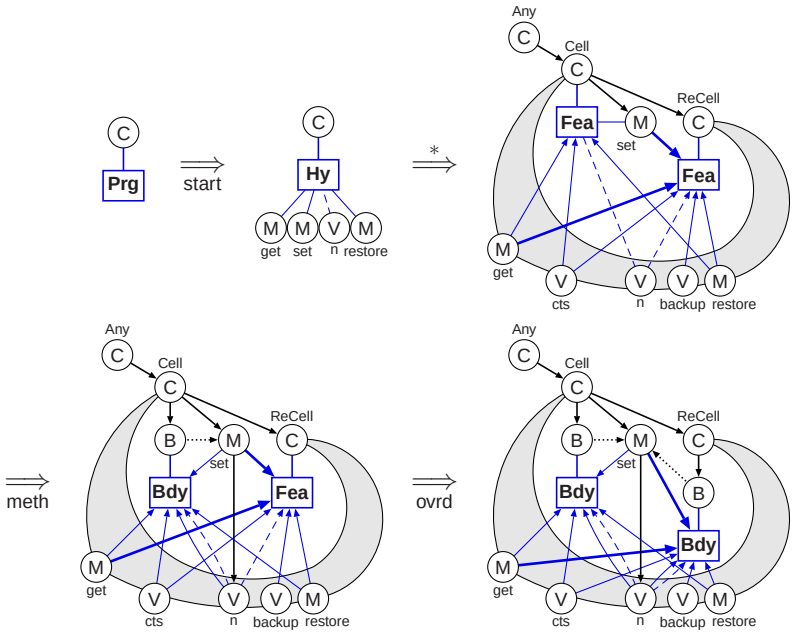


Figure 11: Deriving the program graph of Figure 1b with PG_c

*be overridden by another body since this would violate the application condition of ovrd. Now the derivation in the last row of Figure 8 can be inserted for the body of set in ReCell because n is present. In that derivation, in the step using rule call, the application condition of PG_c guarantees that exactly one expression will be generated as an actual parameter (by the rules of **Act**) since method set has one parameter.*

Definition 13 (Complete Node) Consider a graph $G \in \mathscr{G}(\mathscr{X})$ and a conditional adaptive star grammar Γ. An edge is *terminal in G* if it is not part of a star.

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

Lemma 3 *In graphs derived with PG_c, M-nodes are complete wrt. terminal edges if they are not declared border nodes of any stars named **Hy**, **Cls**, or **Fea**.*

Proof Sketch. By inspection of the right-hand sides of the rules for these stars 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 stars. This is the case for the conditions concerning the parameters of methods.

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

Theorem 3 $\mathscr{L}(PG_c) = \mathscr{P}$.

Proof Sketch. The idea is similar to that of Theorem 2.

"$\subseteq$": Inspection of the rules (as done in Example 6 and Figure 10 above) shows that the border nodes of stars do indeed play the roles given to them. Using these invariants, it can be shown by induction over the structure of rules that every $G \in \mathscr{L}(\mathsf{PG})$ satisfies all Properties (P_1–P_9) of a program graph.

"$\supseteq$": Given a program graph $G \in \mathscr{P}$, we can construct a derivation according to the underlying structure (with edges of type ↓) 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. □

Given a matching of a rule, its application condition is decidable so that there is a chance to combine application conditions with the existing parsing algorithm for adaptive star grammars [DHJM10, Section 6]. 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 condition of another rule. Consider, e.g., the node ReCell in the rightmost graph in the top row of Figure 11. 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 sixth application condition concerning unique implementation. The critical pair analysis for graph transformation rules applies to conditional graph transformation rules; it might be used to analyze conflicts in conditional adaptive

star rules if we can extend the analysis procedure to multiple nodes.

6 Conclusions

With this paper, we continue our search for a powerful, parseable, and readable kind of graph grammars for object-oriented software models. We succeeded in defining the well-known class of program graphs [MEDJ05] by conditional adaptive star grammars. This cannot be done with star grammars (by Theorem 1), and seems to be impossible with adaptive star grammars [DHJ+06, DHJM09].

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 earlier been specified in [Eet07]. Most of these properties can be specified easily with conditional adaptive star grammars. The typing of features, represented by edges from variables and methods to the class defining their type, may be more difficult. For, type compatibility of method calls, for instance, requires to check whether the type of the actual parameter is a subtype of the type of the actual parameter. This requires to check whether there is a path of arbitrary length betwen these types. It is not clear whether this can be specified by application conditions as they are.

Readers may ask themselves: Are there other representations of object-oriented programs as graphs that would be easier to generate, by simpler kinds of grammars? Now, program graphs are a rather straight-forward representation of programs: the hierarchical structure of the program is represented by a spanning tree; different occurrences of entities like methods and variables are identified so that they are represented once. This resembles standard representations of programs as abstract syntax trees and attributed trees that are known from compiler construction [ALSU07], and make it easy to access and modify all information associated with an entity.

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 [Min02] 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 [BPR09] 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! This will only be possible if graph grammars have practical advantages wrt. 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 P_1–P_3 of program graphs can be inferred from the rules of a (conditional) adaptive star grammar. A real advantage of grammars, which are a constructive mechanism, is that they do not only allow to check the validity of a model (by parsing), but also allow to generate sample instances of a model, e.g., for

testing [EKT09].

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. It may turn out that contextual star grammars [HM10] are easier to understand.

The proof of Conjecture 1 poses the theoretical challenge to disprove membership in a class of graph languages. Whereas some results for star languages (e.g., the pumping lemma for the equivalent hyperedge replacement languages [Hab92, DHK97]) helped to prove Theorem 1, 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!* Without Your long-lasting support in form of scientific (and other) discussions, opportunities to visit conferences, and rich supply of co-authors from Your group, much of my reasearch had never happened!

Bibliography

[AC96] M. Abadi, L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, New York, 1996.

[ALSU07] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Pearson/Addison-Wesley, Boston, Massachusetts, 2nd edition, 2007.

[BPR09] A. Bakewell, D. Plump, C. Runciman. Specifying Pointer Structures by Graph Reduction. *Mathematical Structures in Computer Science*, 2009. Accepted for publication.

[Cou87] B. Courcelle. An Axiomatic Definition of Context-free Rewriting and its Application to NLC rewriting. *Theoretical Computer Science* 55:141–181, 1987.

[Cou90] B. Courcelle. Graph Rewriting: An Algebraic and Logical Approach. In Leeuwen (ed.), *Handbook of Theoretical Computer Science*. Volume B, pp. 193–242. Elsevier, Amsterdam, 1990.

[DHJ+06] F. Drewes, B. Hoffmann, D. Janssens, M. Minas, N. V. Eetvelde. Adaptive Star Grammars. In Corradini et al. (eds.), *3rd Int'l Conference on Graph Transformation (ICGT'06)*. Lecture Notes in Computer Science 4178, pp. 77–91. Springer, 2006.

[DHJM09] F. Drewes, B. Hoffmann, D. Janssens, M. Minas. Adaptive Star Grammars and Their Languages. *Theoretical Computer Science*, p. 41, 2009. Accepted for publication.

[DHJM10] F. Drewes, B. Hoffmann, D. Janssens, M. Minas. Adaptive Star Grammars and Their Languages. *Theoretical Computer Science*, 2010. Accepted for publication.

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

[DHM08] F. Drewes, B. Hoffmann, M. Minas. Adaptive Star Grammars for Graph Models. In Ehrig et al. (eds.), *4th International Conference on Graph Transformation (ICGT'08)*. Lecture Notes in Computer Science 5214, pp. 201–216. Springer, 2008.

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

[Eet07] N. V. Eetvelde. *A Graph Transformation Approach to Refactoring*. Doctoral thesis, Universiteit Antwerpen, May 2007.

[EH85] H. Ehrig, A. Habel. Graph Grammars with Application Conditions. In Rozenberg and Salomaa (eds.), *The Book of L*. Pp. 87–100. Springer, Berlin, 1985.

[EKT09] K. Ehrig, J. M. Küster, G. Taentzer. Generating Instance Models from Meta Models. *Software and System Modeling* 8(4):479–500, 2009.

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

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

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

[HP09] A. Habel, K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science* 19(2):245–296, 2009.

[MEDJ05] T. Mens, N. V. Eetvelde, S. Demeyer, D. Janssens. Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution: Research and Practice* 17(4):247–276, 2005.

[Min02] M. Minas. Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation. *Science of Computer Programming* 44(2):157–180, 2002.

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

[SWZ99] A. Schürr, A. Winter, A. Zündorf. The PROGRES 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 13, pp. 487–550. World Scientific, Singapore, 1999.