Adaptive Star Grammars for Graph Models

Frank Drewes[1], Berthold Hoffmann[2], and Mark Minas[3]

[1] Umeå universitet, Sweden
[2] Universität Bremen, Germany
[3] Universität der Bundeswehr München, Germany

Abstract. Adaptive star grammars generalize well-known graph grammar formalisms based on hyperedge and node replacement while retaining, e.g., parseability and the commutativity and associativity of rule application. In this paper, we study how these grammars can be put to practical use for the definition of graph models. We show how to use adaptive star grammars to specify *program graphs*, models of object-oriented programs that have been devised for investigating refactoring operations. For this, we introduce notational enhancements and one proper extension (application conditions). The program graphs generated by the grammar comprise not only the nested composition of entities, but also scope rules for their declarations. Such properties cannot easily be defined by meta-models like UML class diagrams. In contrast, adaptive star grammars cover several aspects of class diagrams.

1 Introduction

Graphs have always been used for visual communication, in science and beyond. In computer science, graph models gained a new quality since the Unified Modeling Language UML emerged: when models drive the development of object-oriented software, they must be processed by computers, and need to be defined precisely. In this paper we investigate how graph models can be specified by graph grammars. As a case study, we choose program graphs, a language-independent model for object-oriented programs that has been devised for defining and analyzing refactoring operations [15,20]. Program graphs have a particular structure: They are composed in a nested fashion according to a context-free syntax, and contain references from entities to their declarations that respect specific scope rules. We define program graphs by adaptive star grammars [4], an extension of the well-known hyperedge and node replacement grammars [11]. We further extend these grammars to enhance the definition and understanding of rules. One of these extensions, application conditions, increases the generative power of the formalism. Without it, the syntax and scope rules of program graphs could probably not have been defined completely. When we compare the grammar to common model definitions by meta-models, it turns out that the grammar allows to infer the structural information about incidences and multiplicities of edges that is specified, e.g., in UML class diagrams. The syntax and scope rules for program graphs, however, could only be specified by logical predicates, e.g., in the object constraint language OCL of UML.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 442–457, 2008.

In Section 2, we recall the definition of adaptive star grammars and add some extensions that enhance their practical use as a specification language. We introduce program graphs in Section 3, explain their adaptive star grammar, and describe their general properties and parsing. In Section 4, we discuss the relation of our grammatical formalism to object-oriented meta-models. We conclude with a discussion of related and future work in Section 5.

2 Adaptive Star Grammars and Their Extensions

Graph grammars generalize the idea of Chomsky grammars to graphs: A set of graph transformation rules defines how the graphs of the language can be derived by applying them successively to a given start symbol.

Node replacement and hyperedge replacement [11] have been studied most thoroughly as grammars for deriving graph languages. Their rules remove a nonterminal node, and attach a replacement graph to its neighbor nodes. The sort (i.e., the label) and the direction of the edge connecting a neighbor to the nonterminal determine completely how the neighbor is attached to the replacement graph; in hyperedge replacement, the number of neighbors is even fixed for every nonterminal. Both formalisms can specify context-free compositions of graphs in the sense of [3], but fail to define simple languages such as the language of all graphs. Adaptive star grammars [4] overcome these limitations by means of a cloning mechanism that makes its rules more powerful.

2.1 Adaptive Star Grammars

Let us briefly define the concepts needed; for more detailed definitions, see [4].

Graphs. Let $\mathbf{S}$, the set of *sorts*, be the disjoint union of countably infinite disjoint sets $\dot{\mathbf{S}}$ and $\bar{\mathbf{S}}$, which will be used as node and edge labels, resp. For the sake of clarity, subsets S of $\mathbf{S}$ may be specified as pairs $(\dot{S}, \bar{S}) = (S \cap \dot{\mathbf{S}}, S \cap \bar{\mathbf{S}})$.

A *graph* $G = \langle \dot{G}, \bar{G}, s_G, t_G, \dot{\ell}_G, \bar{\ell}_G \rangle$ consists of

- finite sets $\dot{G}$ and $\bar{G}$ of *nodes* and *edges*, resp.,
- *source* and *target functions* $s_G, t_G \colon \bar{G} \to \dot{G}$, and
- functions $\dot{\ell}_G \colon \dot{G} \to \dot{\mathbf{S}}$ and $\bar{\ell}_G \colon \bar{G} \to \bar{\mathbf{S}}$ assigning a sort to each node and edge, resp.

For a node x in G, the subgraph consisting of x and its adjacent nodes and incident edges is denoted by $G(x)$. A *border node of* x is a node in $\dot{G}(x) \setminus \{x\}$. In addition, we use common terminology regarding graphs, such as subgraph, disjoint union and isomorphism, assuming that the reader is familiar with them.

Rules and Replacement. Adaptive star grammars are based upon a simple kind of graph transformation that replaces a subgraph of the form $G(x)$ by another graph. For this, define a *rule* $r = \langle y, R \rangle$ to be a pair consisting of a graph R and a distinguished node $y \in \dot{R}$. We call $R(y)$ and $R \setminus \{y\}$ the left- and right-hand side of r, resp.

Let G be a graph and $x \in \dot{G}$ such that $G(x) \cong_g R(y)$ for some isomorphism g. Then the graph $H = G[x /_g r]$ is obtained from the disjoint union of G and R by identifying $R(y)$ with $G(x)$ according to the isomorphism g and removing x and its incident edges. In the following, we simply write $H = G[x/r]$ instead of $G[x /_g r]$. (But note that x and r do not necessarily determine g uniquely, and we may have $G[x /_g r] \ncong G[x /_h r]$ for $h \neq g$.)

In the following, we assume that $\dot{\mathbf{S}}$ is divided into two infinite disjoint sets $\dot{\mathbf{S}}_{\mathrm{N}}$ and $\dot{\mathbf{S}} \setminus \dot{\mathbf{S}}_{\mathrm{N}}$ of *nonterminals* and *terminals*, resp. For each rule r as above, y will be a nonterminal node, i.e., $\ell_R(y) \in \dot{\mathbf{S}}_{\mathrm{N}}$, and its border nodes will be terminal.

Multiple Nodes. To make rules adaptive, we invent *multiple nodes*, similar to the set nodes of [19]. A multiple node x represents any number of ordinary nodes, the *clones of* x. Formally, multiple nodes are designated by special sorts. For this, we assume that $\dot{\mathbf{S}} \setminus \dot{\mathbf{S}}_{\mathrm{N}}$ is partitioned into a set of *singular* node sorts and a set $\ddot{\mathbf{S}}$ of *multiple* node sorts, where the latter is made up of pairwise distinct copies of the singular node sorts. For a singular node sort a, its copy in $\ddot{\mathbf{S}}$ is denoted by $\ddot{a}$. Likewise, the set of multiple nodes of a graph G is denoted by $\ddot{G}$. In figures, a multiple node of sort $\ddot{a}$ is distinguished by drawing it with a dashed line and a "shade" (see Figure 2 on page 450), but labeled with the singular sort a. A graph that does not contain any multiple node is said to be a singular graph. Note that there are no multiple nonterminal nodes.

Cloning. Let G be a graph. A function $\mu\colon \ddot{G} \to \mathbb{N}$ is a *multiplicity for* G. The graph G^μ is obtained from G by *cloning* each node $x \in \ddot{G}$ according to μ, as follows. If $\dot{\ell}_G(x) = \ddot{a}$, then x and its incident edges are replicated $\mu(x)$ times, where the sort of the copies is changed into a, i.e., clones are always singular. If $\mu(x) = 0$, then x and its incident edges are simply deleted.

Adaptive Star Grammars. Call a graph *well formed* if it contains neither adjacent nonterminal nodes nor indistinguishable edges, i.e., parallel edges with identical sorts. A *star* is a graph of the form $G(x)$ that contains neither loops nor parallel edges, such that x is nonterminal and its border nodes are terminal. An *adaptive star rule over* $S \subseteq \mathbf{S}$ is a rule $r = \langle y, R\rangle$, where R is a well-formed graph with sorts in S, and $R(y)$ is a star. A *clone of* r is a rule $\langle y, R'\rangle$ such that

- R' is well formed, and
- there is a multiplicity μ for R such that R' can be obtained from R^μ by identifying some of the border nodes of y with each other (where, of course, only nodes of the same sort can be identified).

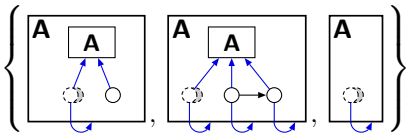
An *adaptive star grammar* is a system $\Gamma = \langle S, \mathcal{P}, Z\rangle$, where $S \subseteq \mathbf{S}$ is a finite set of sorts, $\mathcal{P}$ is a finite set of adaptive star rules, and Z is the initial star (all sorts being taken from S). Given a graph G, we write $G \Longrightarrow_{\mathcal{P}} H$ if $H = G[x /r]$ for some node $x \in G$ and a clone r of an adaptive star rule in $\mathcal{P}$. The *adaptive star language* generated by Γ is the set of all terminal graphs G that can be derived from Z:

$$\mathcal{L}(\Gamma) = \{G \mid Z \Longrightarrow_{\mathcal{P}}^{+} G \text{ and } \dot{\ell}_G(x) \in S \setminus \dot{\mathbf{S}}_{\mathrm{N}} \text{ for all } x \in \dot{G}\},$$

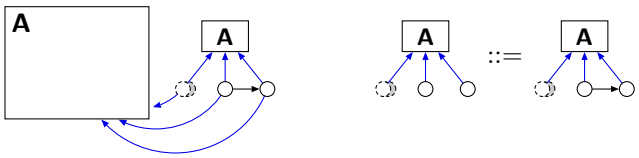
where $\Longrightarrow_{\mathcal{P}}^{+}$ denotes the transitive closure of $\Longrightarrow_{\mathcal{P}}$.

Example 1 (The Language of Unlabeled Graphs). To depict a rule $\langle y, R\rangle$ in a readable and space-efficient manner, we use the following drawing conventions inspired by [13]; see also [11]. The node y designating the left-hand side of the rule is drawn as a large rectangle covering all other nodes of R as well as the edges between them. Edges connecting y with its border nodes attach to the rectangle from the outside. The label of y is written in the upper left corner of the rectangle.

As an example, consider the adaptive star grammar Γ which is given by $\Gamma = ((\{\circ, \ddot{\circ}, \mathbf{A}\}, \{\text{-}\}), R, Z)$, where

$R =$  and $Z =$  .

It derives arbitrary graphs without loops over the "invisible" sorts $\circ$ and -, that is, the class of finite unlabeled graphs. Starting with Z, the first rule makes it possible to add an arbitrary number of border nodes. The second rule adds edges between border nodes, and the third removes the nonterminal node. A more conventional way to draw the second rule can be obtained by "unfolding" it in two steps, like this:



Moving out the right-hand side Separating the sides

Let us briefly discuss the two main differences between the definition of adaptive star grammars used here and the one in [4]. Firstly, using the terminology of [4], we have restricted ourselves to *early cloning*, i.e., cloning produces only singular nodes. As shown in [4], this does not restrict the generative power of adaptive star grammars. Secondly, rules can be applied to subgraphs $G(x)$ that are not stars, but contain parallel edges. Note that, although the left-hand side $R(y)$ of an adaptive star rule $r = \langle y, R\rangle$ is a star, cloning it prior to application may involve taking a quotient that identifies border nodes of y in R^{μ} with each other. This may sound alarming, as it was shown for quite a similar type of adaptive star grammars in [5] that all recursively enumerable languages can be generated. However, note that we restrict this ability to the case where the resulting right-hand side R' is well formed. In particular, R' must not contain indistinguishable edges. As a consequence, adaptive star grammars of the sort defined above can be simulated by ordinary ones (i.e., those in [4]) by using subsets of $\bar{S}$ to label edges in stars and turning every rule into a finite number of rules (corresponding to the allowed quotients).

2.2 Extensions of Adaptive Star Grammars

We now introduce and discuss three extensions of adaptive star grammars that turn out to be useful when writing complex grammars. The first two extensions do not increase the generative power, but are nevertheless important in order to keep the grammars readable, because they lead to a considerable reduction of the number of rules.

Extension 1 (Subsorting). In order to avoid the necessity to draw lots of similar rules, we enrich the set S with additional *abstract sorts* and equip the so-enlarged sets $\dot{S}$ and $\bar{S}$ with *subsort relations*, partial orders denoted by "$\rightarrowtriangle$". For $\ddot{a}, \ddot{b} \in \ddot{\mathbf{S}}$, we require that $\ddot{a} \rightarrowtriangle \ddot{b}$ if and only if $a \rightarrowtriangle b$.

Sorts that are not abstract are *concrete*. For graphs G, H, we write $G \rightarrowtriangle H$ if H and G are equal up to their node and edge sorts and, for all $x \in \dot{G}, e \in \bar{G}$, we have $\dot{\ell}_G(x) \rightarrowtriangle \dot{\ell}_H(x)$ and $\bar{\ell}_G(e) \rightarrowtriangle \bar{\ell}_H(e)$. A graph is concrete if it contains only concrete sorts as labels, and abstract otherwise.

Now, every adaptive star rule $\langle y, R\rangle$ stands for the set of all $\langle y, R'\rangle$ such that R' is a concrete graph with $R \rightarrowtriangle R'$. Clearly, this extension does not make adaptive star grammars more powerful, since the set of all such graphs R' is finite for any given R (as S is finite). However, it can help to avoid a combinatorial explosion of the number of rules that have to be drawn.

Extension 2 (Multiple Subgraphs, Options, and Repetitions). We extend the notion of multiple nodes, as follows. In a rule $r = \langle y, R\rangle$, an induced subgraph M with $y \notin \dot{M}$ may be designated as a *multiple subgraph*. Thus, the multiple subgraph consists of the nodes in $\dot{M}$ and all edges $e \in \bar{R}$ with $s_R(e), t_R(e) \in \dot{M}$. Distinct multiple subgraphs must either be disjoint or properly nested. Graphically, we indicate a multiple subgraph by enclosing it in a dashed box with a dashed shade (see Figure 2). A multiple subgraph may be cloned any number of times prior to rule application. Here, cloning means to choose a multiple subgraph M not contained inside another multiple subgraph, and replacing it with $n \geqslant 0$ isomorphic copies. Each of the copies is connected to the border nodes of M (i.e., the nodes in $\dot{R} \setminus \dot{M}$ which are adjacent to nodes in $\dot{M}$) by copies of the edges that connect M with its border nodes. The cloning procedure is repeated until there are no multiple subgraphs left.

In addition to multiple subgraphs, it is useful to introduce *optional subgraphs*, drawn with dashed borders without a dashed shade, which may have $n \in \{0, 1\}$ clones, and *repeated subgraphs*, drawn with solid borders and a dashed shade, which may have $n \geqslant 1$ clones. If an optional or repeated subgraph consists of a single node, we apply these drawing conventions to the node itself rather than adding a box around it. (Thus, for multiple, optional, and repeated nodes, this is the notation known from PROGRES [19].)

The reader may have noticed that multiple, optional, and repeated nodes and subgraphs lift the use of regular expressions "E^*", "$E^?$", and "E^+" in the extended Backus-Naur form of context-free rules to the graph case, thus avoiding the necessity to use clumsy auxiliary rules. Similar to the case of context-free grammars, the expressive power of adaptive star grammars is not affected. To see

this, note that a multiple subgraph M in a rule r as above may be implemented as follows. We remove M from R, except for the border nodes of y that are contained in M. Singular border nodes of y in M become multiple. Now, we add a new nonterminal x (labeled with a new sort), which is connected to all border nodes of M as well as to the border nodes of y contained in M, by edges labeled with pairwise different sorts. This yields the modified rule $\langle y, R' \rangle$. Finally, two new rules are added to the grammar, which generate any number of copies of the multiple subgraph by iteration. The terminating rule is $\langle x, R'(x) - \dot{M} \rangle$. We leave the straightforward definition of the iteration rule to the reader.

Extension 3 (Application Conditions). The example presented in the next section shows that the power of adaptive star rules suffices to express almost all the important structural properties of program graphs. Nevertheless, in some cases, the subgraph derived by a rule is subject to a condition which we do not know how to capture by adaptive star rules. Therefore, we extend them by allowing required or forbidden terminal subgraphs to be attached to the border nodes in the left-hand side of a rule. In fact, we only need required and forbidden edges between border nodes. We emphasize these edges by underlaying them in grey, and cross them out if they are forbidden.

Note that we cannot offer a formal proof for the necessity of application conditions. In particular, the three uses of application conditions in Fig. 2 can perhaps be avoided by a more ingenious grammar. However, we have not been able to come up with such a solution. Finding it or disproving its existence remains a subject for future work.

3 A Grammar for Program Graphs

Program graphs have been devised in [15], as a language-independent representation of object-oriented code for studying refactoring operations. They capture concepts that are common to many object-oriented languages, like single inheritance and method overriding. The simplified graphs defined here represent only the data flow of programs. See [20] for a specification of program graphs that covers their control flow as well.

A Sample Program Graph. Figure 1 shows a program graph for the following rudimentary program:

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

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

The node sorts C, M, K, V, B, E distinguish program entities: *classes*, *method signatures*, *constants*, *variables*, *bodies* (of methods), and *expressions*. In the figure, some nodes are annotated with the names of the entities they represent.

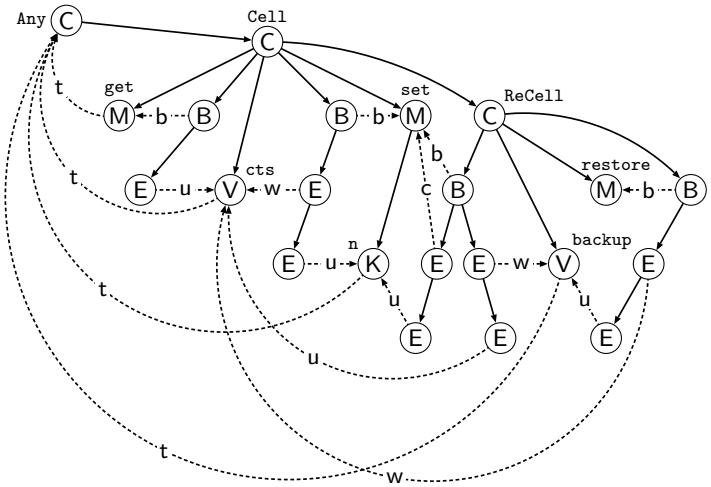


Fig. 1. A program graph

(Note that there are three nodes of sort C, rather than two, the leftmost one representing the predefined root class Any.) Edge sorts distinguish relations between program entities. Arrows drawn as "⟶" represent the *composition* of entities, and arrows of the form "⇢x⇢", with $x \in \{\mathsf{t}, \mathsf{b}, \mathsf{c}, \mathsf{u}, \mathsf{w}\}$, represent *references* of entities to declarations: the *typing* of features by classes, the *binding* of bodies to the signatures of methods, the *call* of methods within expressions, the *use* of constant and variable values in expressions, and the *writing* of values of expressions to variables in assignments.

For instance, the node of sort C representing the class ReCell is composed of four entities. The leftmost node among these four, of sort B, represents the body overriding the body of set in Cell. In turn, the nodes beneath this node represent the expressions (and constants) occurring in the body, as well as the ways in which they call, use, or write to other visible entities.

Program graphs have a particular structure: Their composition follows the syntax of the underlying program texts, whereas their references obey the scope rules for the visibility of declarations in programs. Due to this structural complexity, the generative power of ordinary context-free graph grammars does not suffice to specify the set of all program graphs. Instead, this is now done using an adaptive star grammar, with the extensions introduced in Subsection 2.2.

The Concrete Sorts. The nonterminal node sorts **Prg**, **Hy**, **Cls**, **Fea**, **Sig**, **Bdy**, **Exp**, and **Act** label placeholders for the *program*, a class *hierarchy*, a *class* or *feature* definition, the *signature* or *body* of a method, an *expression*, or *actual parameters*, respectively. In rules, nonterminal nodes are connected to their border nodes by edges of different sort and direction. Every nonterminal node x is incident with one or two *anchor edges*, drawn as "⟵", and may be incident with *scope edges* of the form "⟵v⇢", with $v \in \{\mathsf{g}, \mathsf{h}, \mathsf{i}\}$. Sort and direction of such an edge indicate the *role* of the node at the other end in the derivations of x:

- An anchor edge points to the program entity, represented by a singular node, the components of which are being defined by x. (The only situation in which a nonterminal node has two anchor edges is when a method consisting of a signature and a body is being defined.)
- Ingoing scope edges connect nodes representing declarations within the scope of x; these nodes represent the declarations of constants, variables, methods, and classes that may be referred to in the program entity defined by x.
- Outgoing scope edges connect the nonterminals **Hy**, **Cls**, and **Sig** to the set of declarations that are made visible by the program entities defined by these nonterminals.

The edge sorts g, h, and i classify the visibility of declarations as *global*, *hereditary*, and *internal*, respectively. (In our program graphs, classes are always global, and for the example program above, we assume that all method declarations are global, whereas attribute declarations are internal to a class.) Further edges of sort s connect "selected" nodes of type M to nonterminals **Bdy**, **Exp**, and **Act**, and nodes of type K to nonterminals **Act**. The major purpose of these edges is to keep track of a method signature when deriving a body for it.

The Abstract Sorts and Subsort Relation. In addition to the sorts mentioned above, we use abstract node sorts D, F, and A, together with the following subsort hierarchy on the declarations: A *declaration* D is either a class C, or a *feature* F, which in turn is either a method M, or an *attribute* A, which in turn is either a constant K, or a variable V. Moreover, the scope sorts g, h, and i are subsorts of an abstract scope sort that is just drawn as "$\longrightarrow$". See the sort graphs in Figure 3(a) and (b).

The Rules. The adaptive star rules defining program graphs are shown in Figure 2, and will be explained in the following paragraphs.

Start Rule. The rule start initiates the derivation of a class hierarchy. It applies to the initial star of the grammar, which consists of the nonterminal **Prg** with an anchor node representing the root Any of the class hierarchy. Together with the root node, the clones of the multiple D-node represent all globally visible declarations of the program. All of them are connected to the **Hy**-labeled nonterminal, say x, by in- and outgoing g-edges. Intuitively, this means that they are passed to the rule hy as declarations that are both visible and going to be defined by the derivation of x. Thus, these nodes play two roles in the derivations of x. Note that, in hy, these roles are split as its left-hand side does not contain parallel edges (as required by the definition of adaptive star rules). Thus, quotients will have to be taken in order to apply this rule.

Class Hierarchy. The rule hy defines a class hierarchy, consisting of a root class, plus $n \geqslant 0$ sub-hierarchies. The root classes of the sub-hierarchies are direct subclasses of the root class. The root class defines sets of features of different visibilities: Its internal features are only visible within itself, whereas its hereditary features are also visible to its sub-hierarchies. Its global declarations, together with those of its sub-hierarchies, are the global declarations of the whole hierarchy. The global method declarations are furthermore passed as "hereditary"

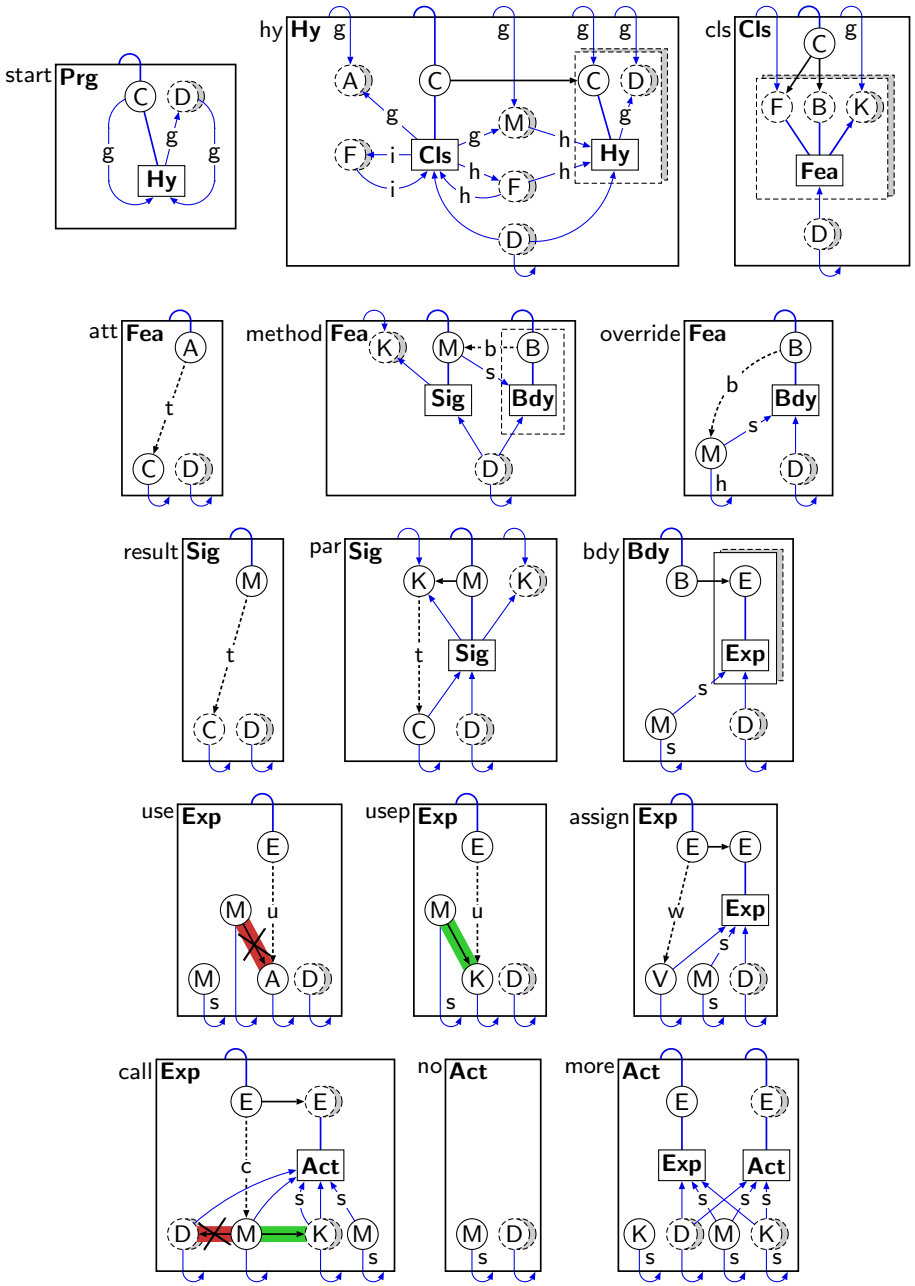


Fig. 2. The rules deriving program graphs

to the sub-hierarchies, because only such methods may be overridden. These methods are thus visible to the sub-hierarchies in two roles, since they are also contained, as "global", in the visible declarations.

Class Definitions. The rule cls derives a set of features as members of a class. Note that the derived features that may have global, hereditary, or internal visibility, depending on which subsort of the abstract derived scope sort is used. The multiple K-node represents the parameters of all methods in the class; they are passed around as global, so that they can be accessed when their method is used. (See the rules for expressions below, which contain application conditions to make sure that the visibility rules are obeyed.)

Feature Definitions. The rules for **Fea** derive the features that may be defined in a class: either a (constant or variable) attribute (rule att), a new method (rule method), or a method overriding a hereditary method (rule override). The new method is abstract if its optional body is missing. In rules method and override, the M-node being defined or overridden is passed as a "selected" entity to **Bdy** (and further on to **Exp** and **Act**) in order to access the parameters of the signature in expressions.

Method Signatures and Bodies. The rules result and par derive signatures of methods. Methods have an optional result type, and any number of formal parameters. The type of a result or a parameter is the class represented by the target of a t-edge. The rule bdy derives method bodies, each consisting of a set of expressions.

Expressions. The rules use, usep, assign, and call specify four kinds of expressions: the use of a visible attribute value, the use of a parameter of a method within its body, an assignment writing the value of an expression to a variable, and the call of a method with a list of actual parameters. Each of these rules inserts a reference to a declaration that is visible in the expression. The application conditions in use and usep make sure that an attribute is visible only if it is not the (constant) parameter of a method, and that a parameter is visible only within the method it belongs to, i.e., the selected one. In rule call, the application conditions make sure that all parameters of the method to be called are subsumed under the multiple K-node – there are no edges to other nodes in the set of visible entities.

Actual Parameters. The rules no and more generate as many expressions (actual parameters) as there are formal parameters of the method being called. To see this, not that each application of more "releases" one K-node.

Properties of Program Graphs. The rules given in Figure 2 have been designed so that the program graph grammar and the language it derives have the following properties:

– The composition edges induce a syntax tree. This is a spanning tree of the program graph. (Thus, program graphs are connected.)

- The composition and anchor edges of the rules induce *syntactic sub-rules* of the grammar, yielding a hyperedge replacement grammar that generates the syntax trees in a purely context-free manner.[1]
- The outgoing scope edges of a nonterminal n point to declaration nodes that are contained in the syntax tree derived by the syntactic sub-rule of n. These border nodes are uniquely determined by the underlying syntax tree.
- The ingoing scope edges of a nonterminal n connect it to the subset of declaration nodes that are method parameters, or visible in the syntax tree generated by n.
- The rules for a nonterminal n may insert reference edges to the ingoing declaration nodes that are visible. Thus terminal program graphs are syntax trees with references from entities to declarations that are visible to these entities.

The reader may have noticed that the program graph grammar has been designed with attribute grammars [14] in mind, a formalism that has been devised for specifying the contextual analysis and translation of (textual) programming languages. More precisely, the grammar corresponds to an attribute grammar with a semantic basis that has sets of syntax tree nodes as values, and set operations as semantic functions (like set union, and member selection). The anchor edges indicate what would be the underlying syntax of the attribute grammar, and the scope edges specify the values of its (inherited and synthesized) attributes.

Parsing. A parsing algorithm for adaptive star grammars has been developed in [17,4]. However, the rules given above must be transformed in order to make them acceptable for the algorithm:

- Optional, multiple, and repeated nodes and subgraphs can be defined as outlined in Extension 2.
- Empty rules, i.e., rules where the replacement just consists of the border nodes of their distinguished node, are also forbidden. In the program graph grammar, this occurs when the multiple subgraph in rule cls has no clone, or when the optional class node in rule result is missing. Empty rules can also be removed, by a transformation similar to the removal of empty rules from context-free word grammars [4].

As mentioned above, application conditions are a real extension of the formalism. Consequently, it is not clear whether adaptive star grammars with application conditions have a decidable membership problem in general. However, for "attribute-grammar-like" rules like the one presented here, a parsing algorithm can take advantage of the structure of such rules. First, the syntax tree can be parsed. Then the outgoing declaration nodes of the nonterminal stars can be determined, thus defining the ingoing declaration nodes. This, finally, allows to check whether the references are in place. During this process, it is easy to check whether the application conditions are fulfilled.

[1] The auxiliary rules defining the multiple subgraphs in the sub-rules become tree-generating hyperedge replacement rules as well.

4 Adaptive Star Grammars and Meta-models

The static structure of object-oriented software is commonly described by meta-models like UML class diagrams. A class diagram contains specifications of three kinds of properties:

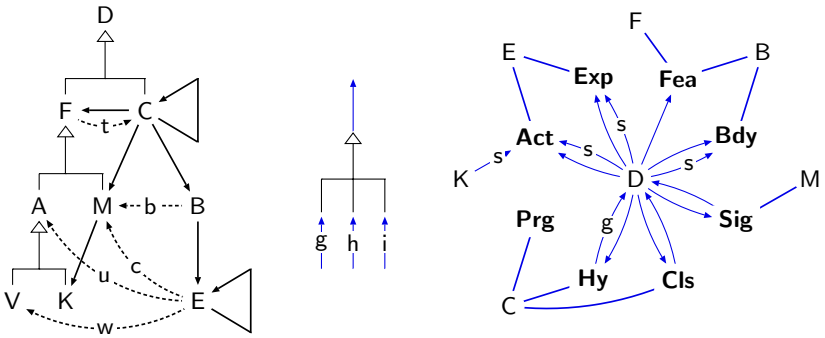
- *Inheritances* describe "is-a" relationships between classes.
- *Incidences* specify which associations may exist between which classes.
- *Multiplicities* define how many associations of some kind may leave and enter some class, thus distinguishing "1:1", "1:n", "n:1", and "n:m" relationships.

For an adaptive star grammar like that of program graphs, label subsorting does already specify the inheritances of a meta-model. Incidences can then be determined by constructing a *sort graph* as follows:

- Form the disjoint union of the start graph with all graphs of all rules, where all multiple node labels are turned into singular ones.
- Identify all equally labeled nodes with each other.
- Insert all inheritance edges in that graph. (This requires "meta-edges" between edges.)
- For some edge e labeled ℓ, remove all edges that are parallel to e and labeled by ℓ or a subsort of ℓ.
- If a node may have equally labeled edges to (or from) every direct sub-node of some node n, replace these edges by a single edge to n.

In Figure 3, the sort graph for program graphs is shown in three pieces: the left-hand side shows the subgraph induced by its composition and reference edge sorts, whereas the subgraph on the right-hand side is induced by its anchor and visibility edge sorts. In the middle, we depict the subsort relation on edge sorts. Altogether, the graphs in Figure 3 correspond to *schema graphs* in PROGRES [19], and to *type graphs with inheritance* in algebraic graph transformation [8].

The multiplicities known from UML class diagrams could be inferred by a similar, yet slightly more complicated procedure. For instance, rules start and hy



(a) Terminal sorts (b) Scope edges (c) Nonterminal sorts

Fig. 3. Node and edge sorts in program graphs

imply that composition edges incident with C-nodes have the source multiplicity $0 \leqslant n$ and the target multiplicity $0 \leqslant m \leqslant 1$, i.e., a C-node has at most one ingoing composition edge indicating its supertype, and may have any number of outgoing composition edges that indicate its subtypes. In the notation of Uml class diagrams, this could be specified by annotating the loop edge at the node C in Figure 3 with “$\xrightarrow{*\ \ ?}$”. Most of the composition edges have the multiplicity “$\xrightarrow{*\quad 1}$”, saying that $n \geqslant 0$ edges may leave, and one edge enters a node.[2] Reference edges have the multiplicity “$\xrightarrow{?\quad *}$”, saying that $n \in \{0, 1\}$ of these edges may leave, and $n \geqslant 0$ of these edges may enter a node.

However, even a class diagram with multiplicities is not as precise as the program graph grammar. The scope rules for programs specified in the rules cannot be described by incidences and multiplicities alone. In an object-oriented meta model, they could be specified by logical formulas written in, e.g., Ocl, the logical language of Uml.

Vice versa, adaptive star grammars do not specify attributes for nodes and edges. However, this could be added in a straightforward way; if needed, their values could be determined by attribute functions. For instance, an attribute arity of B-nodes could be initialized to 0 in rules method and override, and increased in rule par.

In summary, one may say that adaptive star grammars provide a generative alternative to the use of meta models combined with logics. It is certainly a matter of debate which one appears more natural and is easier to use. However, as argued in [6], general graph transformation rules can conveniently describe model transformations, such as refactoring operations. This means that software refactoring systems based on graph transformation could make use of adaptive star grammars for specifying the models to be transformed, and of general graph transformation rules to specify the actual transformations. This avoids the need to mix different paradigms, one of them being used for the specification of the models and the other one for the description and implementation of valid transformations.

5 Conclusions

Adaptive star grammars are rather straightforward extensions of hyperedge and node replacement, and still have some properties of context-free grammars, e.g., nondeterministic versions of associativity and confluence. Adaptive star grammars allow to derive graph languages that define rather advanced software models, specifying properties as they are typically needed to represent programming languages.

The correctness of the grammar presented in Section 3 has not been proved formally. Nevertheless, such a proof should be rather straightforward (though technically tedious), exploiting the similarity of this particular example with attribute grammars pointed out earlier. This would formally support our claim that the power of adaptive star grammars is close to what is necessary for being

[2] Composition edges leaving nodes of sort B have the multiplicity “$\xrightarrow{+\ \ 1}$” .

able to specify program graphs and similar structures. However, without application conditions, adaptive star grammars seem to be unable to cope with complex constraints like the correspondence of actual to formal parameters of methods. It is also open whether other constraints, like the type rules for expressions, can be expressed with adaptive star grammars.

In this paper, we have omitted certain features of the program graphs described in [20]. There, method bodies may contain local declarations and control flow. These features can be captured by refining and extending the rule set presented here. The nodes in [20] do also have attributes, for instance strings representing the identifier of a declaration, or numbers representing the arity of a method. Attributes are convenient to have; they cannot be handled by the grammatical formalism itself, but have to be evaluated while applying a rule, somewhat similar to the way in which application conditions are handled.

Certainly, some properties of models should not be described by the grammatical formalism. For instance, type rules could probably be specified by adaptive rules (with application conditions) as well, but only in a rather complicated way. Generally, attribute values and their dependencies, and negative conditions on the structure should rather be defined by predicates, e.g., in OCL.

So far, grammars, in particular graph grammars, have hardly been considered for defining software models. Graph grammars have been primarily used to define semantics of models or model transformation. Hölscher et al. [12] specify the semantics of some UML diagram types by graph grammars. They specify, among others, state charts and interaction diagrams, i.e., the behavior of object-oriented software. However, graph grammars are used to specify the behavior in terms of the modeling diagrams, and not the syntactic structure of a program as described here. Syntax, in particular abstract syntax, is now usually represented by meta-models together with additional constraints. Typical examples of this approach are based on meta-modeling frameworks MOF [18] or EMF [10]. Moflon [1] allows to specify the abstract syntax of domain specific languages and their implementation using MOF and OCL. Graph transformation systems are used for model transformation and model integration. Tiger [9] uses EMF for specifying the abstract syntax of visual languages by a meta-model. Graph transformations are used to specify editing operations, i.e., behavior of a generated visual editor. While (graph) grammars had been the main approach to specify syntax of (visual) languages (e.g., [16]) in the past, they are now mostly used for behavior specification. Syntax specification by meta-modeling has become the more popular approach, probably since meta-models appear to be easier to build and understand than graph grammars. However, we think that more complex languages, like software models as described in this paper, can be described by graph grammars in a better and more concise manner.

The survey paper [7] has discussed the general scenario of transforming between different graph models. This allows to prove a kind of commutativity between the model transformations and the rules of the graph grammars defining the models concerned. However, it gives no means to check the integrity of

the models themselves, as there is no parsing algorithm for the general graph grammar rules considered there.

In the graph reduction specifications proposed by Bakewell et al. [2], the inverses of the rules have convergent reductions that supply a parsing algorithm. However, they have only be used to specify the shape of data structures involving pointers so far.

Adaptive star grammars are used for shaped generic graph transformation [6]. The rules proposed there may contain variable nodes as placeholders for graphs (and also multiple nodes). Adaptive star grammars are used to specify the legal substitutions for variables. We plan to refine generic transformation so that it its is "shape-preserving": the transformed graphs shall be shaped according to an adaptive star grammar, and rules replace one well-shaped subgraph by another well-shaped graph. This will then allow to define model transformations that may not only be shown to preserve incidence constraints, but the shape of models as well.

In order to use adaptive star grammars in practice, we need a graph systems wherein they can be edited, transformed, and parsers for them can be generated. Such a system is under way. Furthermore, we are interested in the following methodical question: Can grammars be systematically developed from a context-free "kernel" defining spanning trees, as this has been done for program graphs?

Acknowledgments. We want to thank our former colleague Niels Van Eetvelde. Without his work on the first adaptive star grammar for program graphs [20], this paper could not have been written. Furthermore, we thank the anonymous referees for their constructive comments.

References

1. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A standard-compliant metamodeling framework with graph transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 361–375. Springer, Heidelberg (2006)
2. Bakewell, A., Plump, D., Runciman, C.: Specifying pointer structures by graph reduction. Mathematical Structures in Computer Science (to appear, 2008)
3. Courcelle, B.: An axiomatic definition of context-free rewriting and its application to NLC rewriting. Theoretical Computer Science 55, 141–181 (1987)
4. Drewes, F., Hoffmann, B., Janssens, D., Minas, M.: Adaptive star grammars and their languages. Technical Report 2008-01, Departement Wiskunde-Informatica, Universiteit Antwerpen (2008)
5. Drewes, F., Hoffmann, B., Janssens, D., Minas, M., Van Eetvelde, N.: Adaptive star grammars. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 77–91. Springer, Heidelberg (2006)
6. Drewes, F., Hoffmann, B., Janssens, D., Minas, M., Van Eetvelde, N.: Shaped generic graph transformation. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) Applications of Graph Transformation with Industrial Relevance (AGTIVE 2007). LNCS. Springer, Heidelberg (to appear, 2008)

7. Ehrig, H., Ehrig, K.: An overview of formal concepts for model transformations based on typed attributes graph transformation. In: Proc. Graph and Model Transformation Workshop (GraMoT 2005). Electronic Notes in Theoretical Computer Science, vol. 152(4) (2006)
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs on Theoretical Computer Science. Springer, Heidelberg (2006)
9. Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Generation of visual editors as eclipse plug-ins. In: ASE 2005: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pp. 134–143. ACM Press, New York (2005)
10. EMF, Eclipse Modeling Framework web page (2006), `http://www.eclipse.org/emf/`
11. Engelfriet, J.: Context-free graph grammars. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, Beyond Words, vol. 3, pp. 125–213. Springer, Heidelberg (1999)
12. Hölscher, K., Ziemann, P., Gogolla, M.: On translating uml models into graph transformation systems. J. Vis. Lang. Comput. 17(1), 78–105 (2006)
13. Kaul, M.: Syntaxanalyse von Graphen bei Präzedenz–Graph–Grammatiken. Dissertation, Univ. Passau (1985)
14. Knuth, D.E.: Semantics of context-free languages. Math. Sys. Theory 2(2), 127–145 (1968); . Correction: Math. Sys. Theory 5(1), 95-96 (1971)
15. Mens, T., Demeyer, S., Janssens, D.: Formalising behaviour-preserving transformation. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 286–301. Springer, Heidelberg (2002)
16. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. Science of Computer Programming 44(2), 157–180 (2002)
17. Minas, M.: Parsing of adaptive star grammars. Electronic Communications of the European Association of Software Science and Technology 4 (2006), `www.easst.org/eceasst`
18. Object Management Group. Meta Object Facility (MOF) Core Specification, version 2.0 edn., Document - formal/06-01-01 (January 2006)
19. Schürr, A., Winter, A., Zündorf, A.: The PROGRES approach: Language and environment. In: Engels, G., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation, Applications, Languages, and Tools, ch. 13, vol. II, pp. 487–550. World Scientific, Singapore (1999)
20. Van Eetvelde, N.: A Graph Transformation Approach to Refactoring. Doctoral thesis, Universiteit Antwerpen (May 2007)