Electronic Communications of the EASST
Volume X (2010)

Proceedings of the Ninth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2010)

Defining Models – Meta Models versus Graph Grammars

Berthold Hoffmann, Mark Minas

13 pages

Guest Editors: Jochen Küster, Emilio Tuosto
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Defining Models – Meta Models versus Graph Grammars

Berthold Hoffmann[1], **Mark Minas**[2]

[1]hof@informatik.uni-bremen.de
Universität Bremen und DFKI Bremen, Germany

[2]Mark.Minas@unibw.de
Universität der Bundeswehr München, Germany

Abstract: The precise specification of software models is a major concern in model-driven design of object-oriented software. Metamodelling and graph grammars are apparent choices for such specifications. Metamodelling has several advantages: it is easy to use, and provides procedures that check automatically whether a model is valid or not. However, it is less suited for proving properties of models, or for generating large sets of example models. Graph grammars, in contrast, offer a natural procedure – the derivation process – for generating example models, and they support proofs because they define a graph language inductively. However, not all graph grammars that allow to specify practically relevant models are easily parseable. In this paper, we propose *contextual star grammars* as a graph grammar approach that allows for simple parsing and that is powerful enough for specifying non-trivial software models. This is demonstrated by defining program graphs, a language-independent model of object-oriented programs, with a focus on shape (static structure) rather than behavior.

Keywords: Graph grammar, Meta-model

1 Introduction

The precise specification of software models is a major concern in model-driven design of object-oriented software. Such specifications should support a checking procedure for distinguishing valid from invalid models, they should be well-suited for proofs in order to reason about the specified models, and they should allow for automatically generating model instances that may be used as test cases for computer programs being based on such models. The meta-modeling approach is an apparent choice for such specifications. It allows for precise model definitions and provides checking procedures. However, it is less suited for proofs and for instance generation.

Graph grammars are another natural candidate for specifying software models. They precisely define model languages, they are well-suited for proofs because of their inductive way of defining a graph language, and they offer a natural procedure for automatically generating model instances. Several kinds of graph grammars have been proposed in the literature. In order to allow for the specification of practically relevant models, we need a formalism that is *powerful* so that all properties of models can be captured, and *simple* in order to be practically useful, in particular for *parsing* models in order to determine their validity. However, easy to use graph grammar approaches often fail to completely specify models. As a case study, we consider *pro-*

gram graphs, a language-independent model of object-oriented programs that has been devised for specifying refactoring operations on programs [MEDJ05]. However, neither hyperedge replacement grammars [DHK97], nor the equivalent star grammars [DHJM10, Theorem 2.8], nor node replacement grammars [ER97] are powerful enough for completely specifying program graphs. Even the recently proposed adaptive star grammars [DHJ+06, DHJM10] fail for certain more delicate properties of program graphs. Their rules must be extended by application conditions in order to describe program graphs completely [Hof10].

In this paper, we propose the simpler graph grammar approach of *contextual star grammars*, an extension of star grammars that allows for easy parsing. Plain star rules are extended with positive and negative contexts, which must exist (or must not exist, respectively) in order to apply a star rule. Contexts may specify the existence of paths to certain nodes in the host graph, which may then be linked by the rule application. It turns out that program graphs can be defined by a contextual star grammar. Hence, this graph grammar approach allows for the precise specification of program graphs, i.e., non-trivial software models, supports a natural procedure for generating model instances, is well-suited for proofs, and allows for easy parsing. We contrast this grammar with the definition of program graphs using a conventional meta-model, which is specified by a UML class diagram and logical OCL constraints.

The paper is structured as follows. In Section 2, we recall how object-oriented programs can abstractly be represented as *program graphs*. We define the language of program graphs by a metamodel that consists of a class diagram with additional OCL constraints. Then we introduce star grammars in Section 3, show that they can define *program skeletons*, a sub-structure of program graphs, but fail to define program graphs themselves. We introduce contextual star grammars in Section 4, define program graphs with them, and outline an easy parsing procedure. We discuss these specifications—by metamodels and by contextual star grammars—in Section 5. We conclude with some remarks on related and future work in Section 6.

2 Graphs Representing Object-Oriented Programs

Program graphs have been devised as a language-independent representation of object-oriented code that can be used for studying refactoring operations [MEDJ05]. Therefore, they do not represent the abstract syntax of an object-oriented program, but rather its structural components and their dependencies. For instance, they capture single inheritance of classes and method overriding. Data flow between parameters, attributes, and method invocations represents the structure within method bodies.

Consider the object-oriented program shown in Figure 1 as an example. The program, written in object-oriented pseudo code, consists of class Cell and its subclass ReCell. The superclass has an attribute variable cts and two methods get and set. Subclass ReCell inherits these three features and additionally has an attribute variable backup and a method restore. Moreover, it overrides the method set of its superclass.

Figure 1 also shows the corresponding program graph. The graph is actually represented as an object diagram according to the program graphs' model whose class diagram is shown in Figure 2. Note that not all association roles of Figure 2 are shown Figure 1. Only one of the two roles of the associations is shown to avoid clutter. Note also the fat links in Figure 1; they

```
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)
```

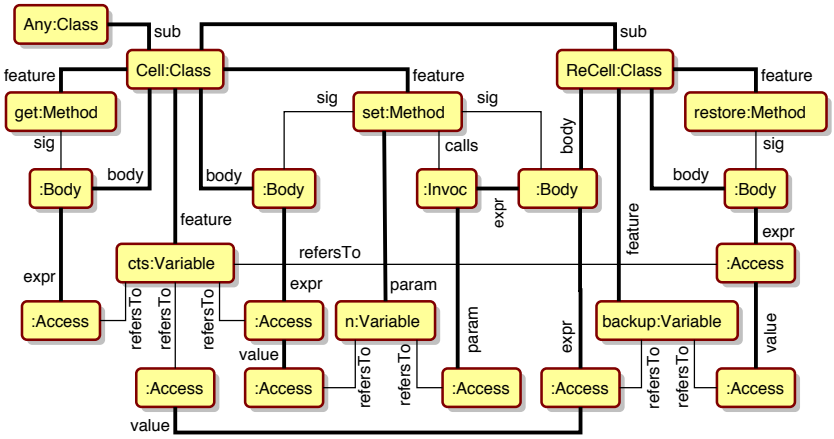

Figure 1: An object-oriented program and its program graph

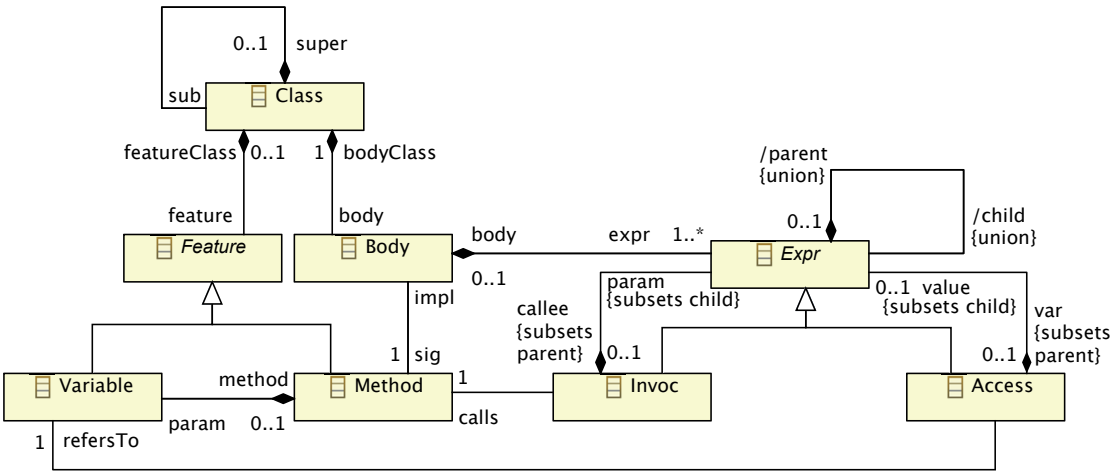

Figure 2: A model $\mathscr{M}$ for program graphs shown as a class diagram

correspond to the composition associations of Figure 2.

Each class is represented by a Class node. Note the universal superclass Any. Each class represents its (protected) attribute variables and (public) methods as features. Method nodes together with Variable nodes as their parameters actually represent method signatures; method bodies are represented separately by Body nodes. If a method is overridden, a new body refers to (we say: *implements*) the signature of the overridden method. Method set is an example: The signature node set:Method is implemented by two Body nodes, one being part of class Cell, the other being part of subclass ReCell. Data flow within method bodies is represented by (abstract) expressions that a body consists of (link expr). Expressions are represented by Access or Invoc nodes, both being subclasses of Expr. Access represents a reference to a variable either using its value, or assigning the value of an expression to it. Invoc nodes represent method invocations with their actual parameters being referred to by param links.

Figure 2 shows a UML class diagram for program graphs. The class diagram represents a model $\mathscr{M}$ of program graphs and also a meta-model because program graphs are models of object-oriented programs, i.e., $\mathscr{M}$ is a model of a modeling language. As usual, missing cardinalities mean 0..*. Also note the child-parent association at class Expr. It is subsetted by the corresponding associations (actually their association ends) for the subclasses Invoc and Access.

However, not all instances of the model represented by the class diagram are valid program graphs. Certain syntactic properties, usually called *static semantics* or *consistency conditions*, cannot be expressed by just a class diagram. The class $\mathscr{P}$ of all program graphs is rather defined by the class diagram and additional constraints:

Definition 1 (Program graphs) The class $\mathscr{P}$ of *program graphs* consists of all instances of the model $\mathscr{M}$ in Figure 2 that additionally satisfy the following constraints:

1. There is exactly one root class, i.e., class node without superclass.
2. A Variable node either belongs to a class (link feature) or to a method (link param).
3. An Expr node either belongs to a Body (link body) or to another expression (link parent).
4. A body *b* may implement a method contained in some class *c* if *b* is contained in *c* or in a subclass of *c*.
5. Every class may contain at most one body defining or overriding a particular method *m*.
6. An Access node *e* may refer to a Variable node representing an attribute contained in some class *c* if *e* is a sub-expression of a body that is contained in *c* or some subclass of *c*.
7. An Access node *e* may refer to a parameter of a Method node *m* if *e* is a sub-expression of a body implementing *m*.

```
context Class
def: visible : Set(Feature) =
if super→isEmpty() then
  feature
else
  feature→union(super.visible)
endif

context Expr
def: visible : Set(Feature) =
if body→isEmpty() then
  parent.visible
else
  body.bodyClass.visible
    →union(body.sig.param)
endif
```

```
1) inv uniqueRoot:
  Class.allInstances()
    →select(c | c.super→isEmpty())→size() = 1

2) context Variable inv validVariable:
  featureClass→isEmpty() <> method→isEmpty()

3) context Expr inv validExpr:
  body→isEmpty() <> parent→isEmpty()

4) context Body inv implementsVisibleMethod:
  bodyClass.visible→includes(sig)

5) context Body inv methodImplementedOnce:
  not bodyClass.body→exists(b | b <> self and b.sig = self.sig)

6,7) context Access inv accessesVisibleVariable:
    visible→includes(refersTo)
```

Figure 3: OCL constraints for the program graph model $\mathscr{M}$.

The *Object Constraint Language* OCL of the UML has been defined for formally defining such consistency conditions [Obj06]. Figure 3 shows the OCL constraints for program graphs based on the class diagram in Figure 2. The derived attributes visible of each Class instance contain all features directly defined in the own class together with all visible features of its superclass. These sets, together with all parameters of the implemented method, are propagated to sub-expressions of method bodies. Conditions 1–5 are formalized by constraints uniqueRoot, validVariable, validExpr, implementsVisibleMethod, and methodImplementedOnce, respectively. Constraint accessesVisibleVariable formalizes conditions 6 as well as 7. Numbers in Figure 3 correspond to the ones used above.

Note that conditions 1–3 require each node, except a unique Class node, to be a composite part of exactly one other node. The following observation follows from the fact that compositions cannot form cycles:

Fact 1 *The subgraph $\bar{P}$ of a program graph P induced by the composition edges is a spanning tree of P; the root of $\bar{P}$ is a Class node.*

3 Star Grammars

We first recall many-sorted graphs:

Definition 2 (Graph) Let $\Sigma = \langle \dot{\Sigma}, \bar{\Sigma} \rangle$ be a pair of disjoint finite sets of *sorts*.

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

Given graphs G and H, a pair $m = \langle \dot{m}, \bar{m} \rangle$ of functions $\dot{m}\colon \dot{G} \to \dot{H}$ and $\bar{m}\colon \bar{G} \to \bar{H}$ is a *morphism* if it preserves sources, targets and sorts.

Star grammars are a special case of double pushout (DPO) graph transformation [EEPT06]. By [DHJM10, Theorem 2.8], they are equivalent to hyperedge replacement grammars [DHK97] a well-understood context-free kind of graph grammars.

Definition 3 (Star) We assume that the node sorts contain *nonterminal sorts* $\dot{\Sigma}_{\mathrm{v}} \subseteq \dot{\Sigma}$ so that the *terminal node sorts* are $\dot{\Sigma}_{\mathrm{t}} = \dot{\Sigma} \setminus \dot{\Sigma}_{\mathrm{v}}$.

Consider a (star-like) graph X, with one center node c_X of nonterminal sort $x \in \dot{\Sigma}_{\mathrm{v}}$, and with some border nodes (of terminal sorts from $\dot{\Sigma}_{\mathrm{t}}$) so that the edges of X connect c_X to some of the border nodes. Then X is called an *incomplete star named* x. An incomplete star is called a *star* if each border node is incident with at least one edge. An (incomplete) star is *straight* if every border node is incident with at most one edge. Let $\mathscr{X}$ denote the class of *stars*, $\mathscr{G}(\mathscr{X})$ the graphs with stars, and $\mathscr{G}$ those without stars (where all nodes are labeled by $\dot{\Sigma}_{\mathrm{t}}$). We assume that nodes of nonterminal sort are not adjacent to each other in any graph.

Definition 4 (Star Replacement) An *incomplete star rule* is written $r = L ::= R$, where the *left-hand side* $L \in \mathscr{X}$ is a straight incomplete star and the *replacement* (right-hand side) is a graph

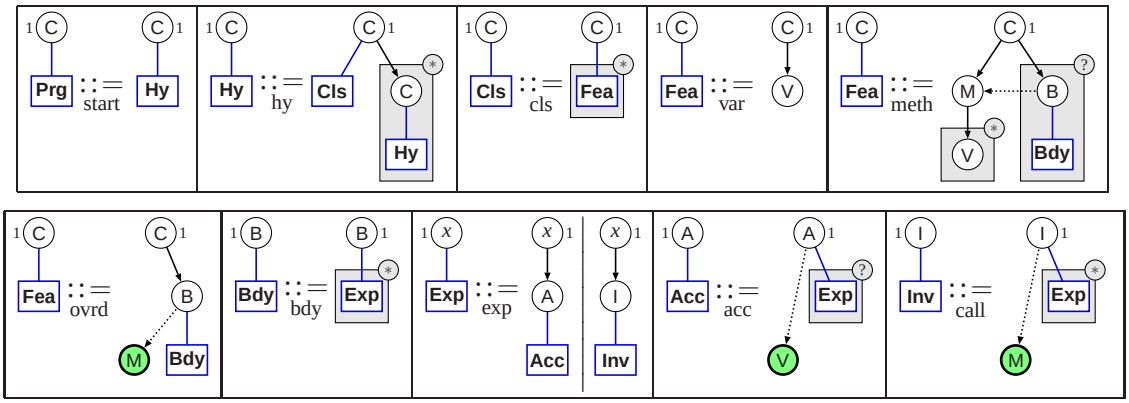

Figure 4: The rules of the star grammar PT

$R \in \mathscr{G}(\mathscr{X})$ that contains the border nodes of L. An incomplete star rule is called a *star rule* if L is a star.

The (incomplete) star rule r *applies* to some graph G if there is a morphism $m\colon L \to G$, yielding a graph H that 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 $\bar{m}(\bar{L})$ and the node $\dot{m}(c_L)$.

We write $G \Rightarrow_r H$ if such a star replacement exists, $G \Rightarrow_{\mathscr{R}} H$ if $G \Rightarrow_r H$ for some r from a finite set $\mathscr{R}$ of (incomplete) star rules, and denote the reflexive-transitive closure of this relation by $\Rightarrow^*_{\mathscr{R}}$.

In the following, we consider star rules only; incomplete star rules will be part of contextual star rules in Section 4.

Example 1 (Star Replacement) *Figure 4 shows a set of star rules. The center nodes of stars are depicted as boxes enclosing the star name. Nodes with terminal symbols are drawn as circles with their sort inscribed.*

The replacements of some rules show examples of abbreviating notation for repetitions in graphs and for optional subgraphs, which we will use frequently in star rules. Shaded boxes with a "∗" in the top-right corner, like those in rules hy, cls, meth, bdy, *and* call, *designate* multiple subgraphs *that may occur n times, $n \geqslant 0$, in the graph, with the same connections to the rest. If the shaded box has a "?" in its top-right corner (in rules* meth *and* acc*), its contained subgraph may occur 1 or 0 times in the graph. Note that this notation is similar to the EBNF notation of context-free string grammars. Likewise, it is just an abbreviating notation; one can always replace rules with this notation by several plain star rules using auxiliary stars and recursion, as long as shaded boxes do not include any of their rules' border nodes.*

Definition 5 (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 2 (Star Grammar for Program Skeletons) *Figure 4 shows the star rules generating the program skeletons that underlie program graphs. The rules define a star grammar* PT*, with the left-hand side of the first rule indicating the start star (named* **Prg***). Terminal node sorts are abbreviations of the class names in Figure 2, e.g.,* C *instead of* Class*. Terminal edge sorts are omitted completely. They can be easily inferred from the sorts of the incident nodes. Rule* exp*, which already is a shorthand for two rules, indicated by the two alternative replacements, uses* x *as border node sort where* x *stands for* B*,* A*, or* I*. Note again that this is just an abbreviating notation.*

Consider rules ovrd*,* acc*, and* call*, which generate method overriding, variable access, and method invocations. The overridden method, the accessed variable, and the invoked method, respectively, are distinguished by drawing them as filled circles with thick lines. In the skeleton rules, they are created anew. In a correct program graph according to Section 2, these distinguished nodes have to be identified ("merged") with the corresponding nodes that have been created elsewhere by rules* var *and* meth*, and represent their declarations. However, identification of nodes cannot be represented by star grammars, but requires contextual star grammars as defined in the next section.*

PT generates graphs that are closely related to program graphs. Given any graph $G \in \mathscr{L}(\mathsf{PT})$, let G^{T} denote the graph obtained from G by removing all filled nodes and dashed edges from G. Let $\mathscr{L}^{\mathrm{T}}(\mathsf{PT}) := \{G^{\mathrm{T}} \mid G \in \mathscr{L}(\mathsf{PT})\}$ denote the class of all such graphs. The following fact directly follows from the structure of rules in PT:

Fact 2 *$\mathscr{L}^{\mathrm{T}}(PT)$ is a language of trees.*

Obviously, grammar PT creates a single root class and for each class an arbitrary number of subclasses. Each class gets arbitrarily many variables, method declarations, and bodies that consist of an arbitrary number of expressions that are either a variable access or a method invocation, consistent with the class diagram in Figure 2. Apparently, $\mathscr{L}^{\mathrm{T}}(\mathsf{PT})$ is the language of all trees that fit the class diagram when considering just its composition associations and additionally requiring the conditions 1–3 in Definition 1. Based on Fact 1 we can infer:

Fact 3 *The spanning tree $\bar{P}$ of each program graph $P \in \mathscr{P}$ (cf. Fact 1) is a member of $\mathscr{L}^{\mathrm{T}}(PT)$.*

Let $\mathscr{L}^{\mathrm{M}}(\mathsf{PT})$ denote the language of all graphs that can be obtained from a member of $\mathscr{L}(\mathsf{PT})$ by merging each filled node with a corresponding declaration node. Let $P \in \mathscr{P}$ be an arbitrary program graph. Its spanning tree $\bar{P}$ is equal to graph G^{T} of some graph $G \in \mathscr{L}(\mathsf{PT})$ because of Fact 3. P can be obtained by identifying the filled nodes of G with appropriate declaration nodes, i.e., $P \in \mathscr{L}^{\mathrm{M}}(\mathsf{PT})$. On the other hand, it is obvious that each identification of filled nodes with declaration nodes fits the class diagram and conditions 1–3 in Definition 1. Therefore, each graph $G \in \mathscr{L}^{\mathrm{M}}(\mathsf{PT})$ satisfying conditions 4–7, too, is a program graph. We can summarize:

Fact 4 *The set of all graphs $G \in \mathscr{L}^{\mathrm{M}}(PT)$ that satisfy conditions 4–7 in Definition 1 is equal to $\mathscr{P}$.*

Star grammars are context-free in the sense defined by Courcelle [Cou87]. This suggests that

their generative power is limited. Indeed, we have the following

Fact 5 *There is no star grammar* Γ *with* $\mathscr{L}(\Gamma) = \mathscr{P}$.

Proof Sketch *Consider the rule* call *in Figure 4. (The situation is similar for rules* ovrd *and* acc*.) This rule generates a new node for a method (drawn filled, and with thick lines). However, for generating a program graph, the rule should insert a call to a method that already exists in the host graph, and may be called in the expression. Due to the restricted form of star rules, the method had to be on the border of the rule. Since expressions may call every method in the graph, the star rule must have all these methods as its border nodes so that one of them can be selected. However, the number of callable methods depends on the size of the program, and is unbounded. Thus a finite set of star rules does not suffice to define all legal method calls.* □

4 Contextual Star Grammars

The adaptive star grammars defined in [DHJ+06] overcome the deficiencies illustrated in the proof sketch for Fact 5 by allowing the left-hand sides of star rules to adapt to as many border nodes as needed. A further extension, by positive and negative application conditions, extended their power, however with rather complicated rules [Hof10]. In this paper, we therefore consider a different extension of star rules with which program graphs can be completely defined in a simple way. We allow that the application of a star rule depends on its *context* in the host graph. Some *positive* context may be required whereas other, *negative*, context is forbidden if the rule shall apply. Nodes of the positive context may be used by the replacement graph of the rule.

Definition 6 (Contextual Star Rule) A *contextual star rule* r has the form $r = P/N \wr L ::= R$, where $L ::= R$ is an incomplete star rule, and the *positive contexts* P as well as the *negative contexts* N are (decidable) sets of graphs that contain the border nodes of L as subgraph. (Note that r is a star rule if L is a star and P and N are empty sets.)

The contextual star rule r *applies* to some graph G if there is a morphism $m\colon L \to G$ that can be extended to a morphism $C \to G$ for at least one positive context $C \in P$ (if $P \neq \emptyset$) and that cannot be extended to a morphism $C \to G$ for any negative context $C \in N$, yielding a graph H by applying the incomplete star rule $L ::= R$ to G with morphism m.

Then we write $G \overset{c}{\Longrightarrow}_r H$, $G \overset{c}{\Longrightarrow}_{\mathscr{R}} H$ if $G \overset{c}{\Longrightarrow}_r H$ for some r from a finite set $\mathscr{R}$ of contextual star rules, and denote the reflexive-transitive closure of this relation by $\overset{c}{\Longrightarrow}^*_{\mathscr{R}}$.

Definition 7 (Contextual Star Grammar) $\Gamma = \langle \mathscr{G}(\mathscr{X}), \mathscr{X}, \mathscr{R}, Z \rangle$ is a *contextual star grammar* (CSG) with a *start star* $Z \in \mathscr{X}$ and a finite set $\mathscr{R}$ of contextual star rules. The *language* of Γ is obtained by exhaustive application of its rules, starting from the start star: $\mathscr{L}(\Gamma) = \{G \in \mathscr{G} \mid Z \overset{c}{\Longrightarrow}^*_{\mathscr{R}} G\}$.

In the following, we will either enumerate context graphs of the sets P and N of positive and negative contexts, respectively, or we will specify them by path expressions similar to the PROGRES language [Sch97]. Even more powerful specifications of context graphs are conceivable, e.g., by hyperedge replacement systems, as proposed in [HR10], or by star grammars. But this

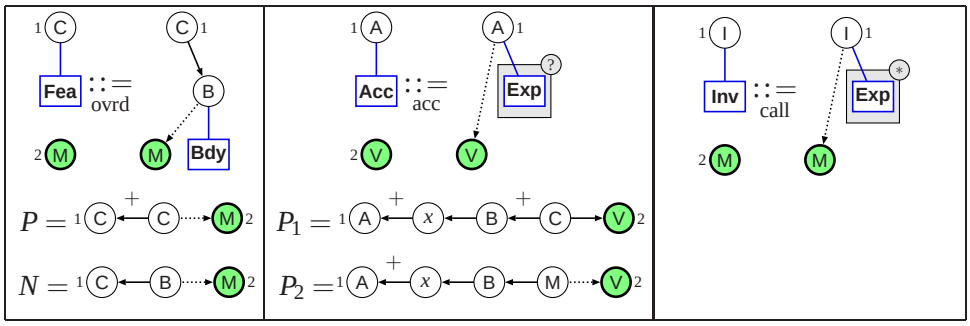

Figure 5: The contextual star rules of the grammar PG

is not necessary for specifying program graphs.

Example 3 (Contextual Star Grammar for Program Graphs) *The CSG for program graphs contains the star rules start, hy, cls, var, meth, bdy, and exp of PT in Figure 4, and the contextual rules in Figure 5. In these rules, specifications of positive and negative contexts are drawn below their incomplete star rules. Small numbers indicate the correspondence between context nodes and border nodes.*

So filled nodes in rules ovrd, acc, and call of PT are turned into context nodes in PG. Path expressions encode conditions 4–7 in Definition 1. A small "+" above an edge represents any path of length $\geqslant 1$. In rule ovrd, the path expression of the positive context P encodes condition 4, i.e., the method declaration must be inherited by the current class, while the negative context encodes condition 5, i.e., the current class must not have a second body for the same method declaration. Rule acc does not have a negative context, but two positive contexts. Rule acc may be applied either with context P_1 or with context P_2. P_1 and P_2 encode conditions 6 and 7, i.e., access to a visible attribute variable and to a visible parameter, respectively. The node labeled x represents a node of any sort. Rule call has empty positive and negative contexts and is applicable if its incomplete star rule applies. However, it still is a contextual star rule because its left-hand side requires the existence of an M-node in the context.

In Figure 6, some steps in the derivation of the program graph in Figure 1 are shown. The first graph represents the class hierarchy of the program graph. The grey bubbles in these graphs abstract from parts of the derived graph that are nor relevant for the derivation steps shown. The rule ovrd can be applied to this graph, where we draw the context node filled, with thick lines, and underlay the path leading to it in grey. We use the same drawing convention for the remaining derivation steps using bdy, call, acc, and again acc.

The following fact is a direct implication of Fact 4:

Fact 6 $\mathscr{L}(PG) = \mathscr{P}$.

Star grammars can be easily transformed into equivalent hyperedge replacement grammars [DHJM10, Theorem 2.8] and vice versa by interpreting stars as hyperedges with nonterminal labels. Hence, parsers for hyperedge replacement grammars like the DIAGEN parser [Min02]

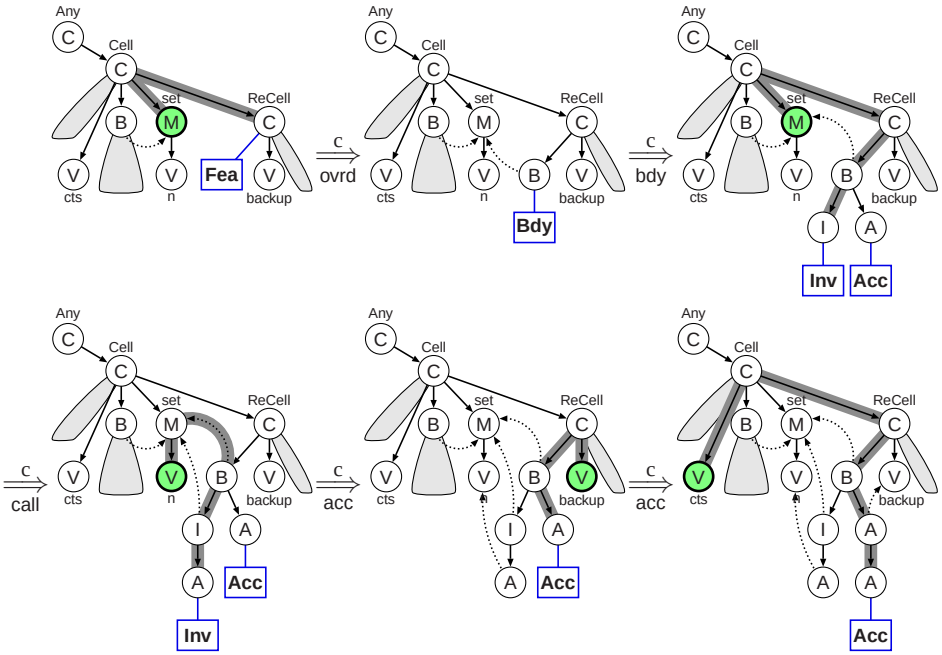

Figure 6: Deriving the program graph of Figure 1 with PG

can be used to parse star grammars. The same parser with only slight extensions can also be used for parsing CSGs. This parser is outlined below.

The parser works on CSGs like a *Cocke-Younger-Kasami* (CYK) parser for string grammars. The CSG has to be in *Chomsky normalform* (CNF) so that each production rule is either terminal or nonterminal. The right-hand side of a terminal rule is a terminal graph with only one edge, the right-hand side of a non-terminal rule is the union of two stars. Similar to string grammars, each CSG that does not produce the empty graph can be transformed into CNF. Given a terminal graph G, the parser creates a derivation for G, if it exists, in two phases. The first phase completely ignores contexts of contextual star rules and creates candidates for derivations. The second phase searches for a derivation by checking these candidates, this time considering contexts.

In the first phase, the parser builds up n sets $L_1, L_2, \ldots, L_n$ where n is the number of edges in G. Each set L_i eventually contains all stars that can be derived to any subgraph of G that contains exactly i edges. Set L_1 is built by first finding each embedding of the right-hand side of each terminal rule and adding the star of the corresponding left-hand side to L_1. If the corresponding rule is a contextual star rule with an incomplete star as left-hand side, only the (complete) star within the incomplete star is added to L_1. The remaining sets are then constructed using nonterminal rules. A nonterminal rule is reversely applied by searching for appropriate stars s and s' in sets S_i and S_j, respectively. If a nonterminal rule (without considering contexts) is applicable, i.e., if the rule's right-hand side is isomorphic to the union of s and s', a new star corresponding to the rule's left-hand side is added to a set S_k. Note that $k = i + j$ since each star in a set S_i can be derived to a subgraph of G with exactly i edges. Each instance of the start star Z in S_n represents a derivation candidate for G.

The second parser phase checks these derivation candidates by testing for each application of a contextual star rule whether required context has already been created and forbidden context has not (yet) been created earlier in the derivation. The parser stops when it finds the first valid derivation, or when it has checked the last derivation candidate without success.

5 Discussion

In the previous sections we have used two different techniques to describe program graphs. We have shown that the specification of program graphs by CSGs is indeed equivalent to their definition using a model together with OCL constraints. Moreover, both approaches allow for automatic checking whether a given graph is a valid program graph. Both specifications are actually even more closely related as the following discussion shows.

The CSG for program graphs consists of (plain) star rules and contextual star rules. As described by Fact 2, the tree structure (made from composition links) of program graphs can be constructed by (plain) star rules. Plain star grammars, however, fail for links refersTo and call, i.e., those program graph edges that represent references to objects that are located "far away" in the program structure. Contextual star rules are needed to add those edges; the far away objects are represented by the rules' context nodes (Figure 5). The positive and negative contexts of those rules play the same role like the OCL constraints for conditions 4–7 in Figure 3. These constraints actually can be considered as an OCL implementation of the rules' context conditions. E.g., constraint methodImplementedOnce for condition 5 in Definition 1 prohibits a second body of the same method within the same class; this exactly corresponds to the negative context N of rule ovrd (Figure 5). The path expressions P, P_1, and P_2 of rules ovrd and acc, respectively, realize conditions 4, 6, and 7. Their OCL realizations make use of the derived attributes visible whose recursive definition exactly reflects the iteration operator "+" in the path expressions.

The contextual star grammar PG (Figure 4 and 5) for program graphs has been created by hand. The discussion, however, has revealed a close relation between OCL constraints and positive as well as negative contexts on the one hand, and between class diagram and contextual star rules without those contexts on the other hand. This close relation may lead to a procedure for translating CSGs and meta models with OCL constraints into each other at least in a semi-automatic way. This line of research is also motivated by work of Ehrig et al. [EKT09]. They create graph transformation rules with graph constraints from meta models with OCL constraints. The graph transformation system is then used to automatically generate model instances of the meta model. However, they only consider very restricted OCL constraints that are not sufficient for the specification of program graphs, and their generated layered graph transformation rules are rather complicated which are less suited for parsing.

6 Conclusions

We have extended star rules by positive and negative context. These contextual star grammars allow to define program graphs precisely, without sacrificing parseability. Program graphs, which represent certain aspects of object-oriented programs, can also be defined by a class diagram together with OCL constraints in a model-based style. However, this approach is less suited

for inductive proofs or for automatic instance generation than the proposed grammar-based approach. A comparison of both specification approaches, however, has shown close relations between both specification approaches which may allow for a semi-automatic translation process between both specification approaches. This will be subject of future work.

Too many kinds of graph grammars are related to contextual star grammars to mention all of them. So we restrict our discussion to those that aim at similar applications. Contextual star rules re-use *path expressions* first devised by M. Nagl [Nag79], which have later been implemented in the PROGRES graph transformation language [Sch97]. Using context conditions and examining their relation to logical graph properties and constraints is not new either. A. Habel and K.-H. Pennemann have shown that nested graph conditions are equivalent to first-order graph formulas [HP09]. These conditions are still too weak to specify program graphs, as they only allow to require or forbid the existence of subgraphs of bounded size. Only in [HR10], A. Habel and H. Radke devised nested graph conditions with variables that allow to express the conditions of CSGs (and more). However, rules and grammars using such conditions are not yet considered in that paper. 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, but are not powerful enough to define models like program graphs. Graph reduction grammars [BPR10] have been proposed to define and check the shape of data structures with pointers. While the form of their rules is not restricted, reduction with them is required to be terminating and confluent, so that random application of rules provides a backtracking-free parsing algorithm. It is still open whether graph reduction grammars suffice to define program graphs.

The nested patterns [HJG08] recently introduced to GRGEN [GBG+06], an efficient graph rewrite tool, allow to express nested graph conditions with variables that are defined by hyperedge replacement systems, and can thus be used to implement contextual star grammars (like that for program graphs) in the future.

Acknowledgements: We thank the reviewers for their constructive criticism that helped to improve our paper.

Bibliography

[BPR10] A. Bakewell, D. Plump, C. Runciman. Specifying Pointer Structures by Graph Reduction. *Mathematical Structures in Computer Science*, 2010. To appear.

[CEM+06] A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, G. Rozenberg (eds.). *3rd Int'l Conference on Graph Transformation (ICGT'06)*. Lecture Notes in Computer Science 4178. Springer, 2006.

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

[DHJ+06] F. Drewes, B. Hoffmann, D. Janssens, M. Minas, N. V. Eetvelde. Adaptive Star Grammars. Pp. 77–91 in [CEM+06].

[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 in [Roz97].

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

[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 in [Roz97].

[GBG+06] R. Geiß, G. V. Batz, D. Grund, S. Hack, A. Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. Pp. 383–397 in [CEM+06]. URL: http://www.grgen.net.

[HJG08] B. Hoffmann, E. Jakumeit, R. Geiß. Graph Rewrite Rules with Structural Recursion. In Mosbah and Habel (eds.), *2nd Intl. Workshop on Graph Computational Models (GCM 2008)*. Pp. 5–16. 2008.

[Hof10] B. Hoffmann. Conditional Adaptive Star Grammars. *Electr. Comm. of the EASST*, 2010. To appear.

[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.

[HR10] A. Habel, H. Radke. Expressiveness of graph conditions with variables. In Ehrig and Ermel (eds.), *International Colloquium on Graph and Model Transformation (GraMoT'10)*. 2010. To appear in Electr. Comm. of the EASST.

[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.

[Nag79] M. Nagl. *Graph-Grammatiken: Theorie, Anwendungen, Implementierungen*. Vieweg-Verlag, Braunschweig, 1979. In German.

[Obj06] Object Management Group. Specification of the Object Constraint Language. http://www.omg.org/spec/OCL/2.0/, 2006.

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

[Sch97] A. Schürr. Programmed Graph Replacement Systems. Chapter 7 in [Roz97].