Electronic Communications of the EASST
Volume 39 (2011)

Graph Computation Models
Selected Revised Papers from the
Third International Workshop on
Graph Computation Models (GCM 2010)

Generating Instance Graphs from Class Diagrams with Adaptive Star Grammars

Berthold Hoffmann, Mark Minas

20 pages

Guest Editors: Rachid Echahed, Annegret Habel, Mohamed Mosbah
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Generating Instance Graphs from Class Diagrams with Adaptive Star Grammars

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

[1] hof@informatik.uni-bremen.de
Universität Bremen
Postfach 330 440
28344 Bremen, Germany
and
Fachgebiet Sichere Kognitive Systeme
DFKI Bremen
Enrique-Schmidt-Str.5
28569 Bremen, Germany

[2] Mark.Minas@unibw.de
Universität der Bundeswehr München
85577 Neubiberg, Germany

Abstract: In model-driven software engineering, class diagrams are used to define the structure of object-oriented software and valid object configurations, i.e., what objects may occur in a program and how they are related. Object configurations are essentially graphs, so that class diagrams define graph languages. Class diagrams are declarative, i.e., it is quite easy to check whether a graph is an instance of a class diagram. Graph grammars, on the other hand, define a graph language by derivation and are thus well suited for constructing instance graphs. This paper describes how a class diagram can be translated into a graph grammar that defines the same graph language as the original class diagram. Such a graph grammar may then be used for, e.g., automatically generating valid object configurations as test cases. In contrast to earlier attempts, the presented approach allows to translate class diagrams with arbitrary multiplicities, unique and non-unique associations, composition associations, class generalization, association subsetting and redefinition. This is made possible by using adaptive star grammars, a special kind of graph grammars.

Keywords: Class diagram, Graph grammar, Adaptive star grammar

1 Introduction

The model-driven design of software relies on the precise specification of models, often with diagrams in the *uniform modeling language*, UML. Class diagrams are the sub-language of UML for specifying the structure of object-oriented classes and possible configurations of objects that are instances of these classes. Configurations primarily consist of sets of class instances and links between them. Links are instances of the class diagram's associations. Configurations can be considered as graphs, called *instance graphs* in the following: each object corresponds to a node, and each link corresponds to a directed edge where source and target are used to distinguish the two roles of the (binary) link. The set of all object configurations that are compatible with a class diagram, therefore, corresponds to a language of graphs, i.e., a class diagram specifies a graph language.

It is easy to check whether a given instance graph is compatible with a class diagram. In contrast, constructing sample instance graphs for a class diagram is rather difficult. This, however, is important to generate test cases for a software model. In [EKT09], graph transformation rules have been used to do this. Here we use an adaptive star grammar [DHJM10, DHJ$^+$06, DHM08] for this purpose. The approach described in [EKT09] supports only restricted meta models (e.g., class diagrams with constraints): it does not distinguish *unique* from *non-unique* associations, considers only very restricted association multiplicities, and does not cover composite associations or associations with redefined or subsetted association ends. Adaptive star grammars, as shown in this paper, allow for a fairly straight-forward treatment of all of these concepts. Moreover, adaptive star grammars do not need additional control mechanisms like negative application conditions or prioritizing some rules over others by rule layers as it is necessary in [EKT09].

The rest of the paper is structured as follows. We briefly introduce adaptive star grammars in the next section before we recall some properties of UML class diagrams in Section 3. In Section 4, the main part, the rules generating instance graphs of class diagrams are defined, explained, and illustrated with an example. The conclusions (Section 5) mention some related and future work.

2 Adaptive Star Grammars

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

Node replacement and hyperedge replacement [Eng99] 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 specify context-free compositions of graphs in the sense of [Cou87]; however, they fail to define even simple languages, such as the class of all graphs. *Adaptive star grammars* [DHJM10] overcome these limitations by means of a cloning mechanism that allows the neighbors of a nonterminal to be attached to the replacement graph in an arbitrary fashion.

Let us briefly define the concepts needed; for more detailed definitions, see [DHM08] and for a more thorough discussion [DHJM10].

Graphs. Let the set C of *sorts* be the disjoint union of finite disjoint sets $\dot{C}$ and $\bar{C}$ for labeling nodes and edges, respectively. We distinguish a subset $N \subseteq \dot{C}$ of *nonterminal names*.

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*, respectively, *source* and *target functions* $s_G, t_G \colon \bar{G} \to \dot{G}$, and functions $\dot{\ell}_G \colon \dot{G} \to \dot{C}$ and $\bar{\ell}_G \colon \bar{G} \to \bar{C}$ assigning a sort to each node and edge, respectively. A node $x \in \dot{G}$ is called *nonterminal* if $\dot{\ell}_G(x) \in N$, and *terminal* otherwise.

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 $\dot{G}(x) \setminus \{x\}$. Note that graphs may have several edges with the same source and target; such edges are called *parallel*. Parallel edges with the same label are called *indistinguishable*. In the following, we assume that the reader is familiar with common graph terminology, such as subgraph, disjoint union and isomorphism.

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

Let G be a graph with a node $x \in \dot{G}$ 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.

Multiple Nodes. To make graphs (and rules) adaptive, we distinguish a subset $\ddot{G} \subseteq \dot{G}$ of terminal nodes in a graph G as *multiple nodes*, similar to the set nodes of Progress [SWZ99]. A multiple node x represents any number of ordinary nodes, which are called *clones of* x. (The nodes $\dot{G} \setminus \ddot{G}$ are called *singular*.) In figures, a multiple node is distinguished by drawing it with double lines (see Example 1 below). A graph that does not contain any multiple node is said to be *singular*. Note that nonterminal nodes are always singular.

Cloning. Let G be a graph. A function $\rho \colon \ddot{G} \to \mathbb{N}$ is a *replicator for* G. The graph G^{ρ} is obtained from G by *cloning* each node $x \in \ddot{G}$ according to ρ, by replicating x and its incident edges $\rho(x)$ times. If $\rho(x) = 0$, then x and its incident edges are simply deleted.

Adaptive Star Grammars. Call a graph *simple* if it contains neither adjacent nonterminal nodes nor indistinguishable edges. A graph of the form $G(x)$ is a *star* if it contains neither loops nor indistinguishable edges, x is nonterminal and its border nodes are terminal. An *adaptive star rule over* C is a rule $r = \langle y, R\rangle$, where R is a simple graph with sorts in C, and $R(y)$ is a star. A *clone of* r is a rule $\langle y, R'\rangle$ such that R' is simple, and there is a replicator ρ for R such that R' can be obtained from R^{ρ} 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* (ASG) is a system $\Gamma = \langle C, \mathscr{P}, Z\rangle$, where $\mathscr{P}$ is a finite set of adaptive star rules, and Z is the initial star, which has no multiple border nodes. (All sorts are taken from C.) Given a graph G, we write $G \Longrightarrow_{\mathscr{P}} H$ if $H = G[x\,/\,r]$ for some node $x \in G$ and a clone r of an adaptive star rule in $\mathscr{P}$. The *adaptive star language* generated by Γ is the set of all terminal graphs G that can be derived from Z:

$$\mathscr{L}(\Gamma) = \{G \mid Z \Longrightarrow^{+}_{\mathscr{P}} G \text{ and } \dot{\ell}_G(x) \in \dot{C} \setminus N \text{ for all } x \in \dot{G}\},$$

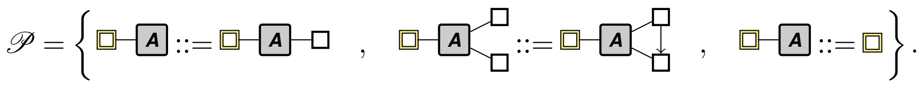
where $\Longrightarrow^{+}_{\mathscr{P}}$ denotes the transitive closure of $\Longrightarrow_{\mathscr{P}}$.

Late Cloning. In the terminology of [DHJM10], these definitions make use of *early cloning*, where neither the graphs in derivations, nor the clones of rules contain multiple nodes. However, it requires to "guess" in advance how many clones of a multiple node must be made, which is not always adequate. Especially for constructing derivations, it is better to do cloning as late as possible. In the following, we will use *late cloning*, a corresponding way of constructing derivations, which has been considered in [DHJM10] as well.

A *late replicator* $\ddot{\rho}\colon \ddot{G} \to \mathbb{N} \times \mathbb{N}$ sends multiple nodes to pairs of numbers that represent the numbers of singular and multiple nodes that shall be made of a multiple node, respectively. The graph $G^{\ddot{\rho}}$ contains, for every multiple node $x \in \ddot{G}$ with $\ddot{\rho}(x) = (n,m)$, $n+m$ clones, whereof m are designated as multiple.

In order to apply an adaptive rule $r = \langle y, R \rangle$ to a graph G with late cloning, a combined late replicator $\ddot{\rho}\colon G(x) \cup R(y) \to \mathbb{N} \times \mathbb{N}$ is used to clone both G and r so that $G(x)^{\rho} \cong R'(y)$ where R' is simple and obtained from R^{ρ} by identifying some of the border nodes of y. A step using late cloning is denoted as $G \Rightarrow_{\mathscr{P}} H$ again; H is obtained by applying $\langle y, R' \rangle$ to G^{ρ}. Like early cloning, late cloning does not restrict the language of singular graphs generated by an adaptive star grammar [DHJM10]. [1]

Example 1 (The Language of Unlabeled Graphs) *As an example, consider the adaptive star grammar* Γ *which is given by* $\Gamma = (C, \mathscr{P}, Z)$, *where* $\dot{C} = \{\varepsilon, \boldsymbol{A}\}$, $\bar{C} = \{\lambda\}$, $Z = \boxed{A}$, *and*



A rule $\langle y, R \rangle$ *is drawn as lhs* $::=$ *rhs where lhs* $= R(y)$ *and rhs* $= R \setminus \{y\}$. *The grammar derives arbitrary graphs without loops over the sorts* ε *and* λ, *which are not drawn, 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.*

Let us briefly discuss a major difference between the definition of adaptive star grammars used here and the one in [DHJM10]. The definition of stars and star rules used here is more general since stars are allowed to have parallel (but no indistinguishable) edges which is not allowed in [DHJM10]. Moreover, cloning a star rule $r = \langle y, R \rangle$ prior to application may involve taking a quotient that identifies border nodes of y in R^{ρ} with each other like in [DHM08]. This may sound alarming, as it was shown in [DHJ+06] that quite a similar type of adaptive star grammars can generate all recursively enumerable languages. However, note that the resulting graph R' – after cloning and taking the quotient – must be simple. In particular, R' must not contain indistinguishable edges. The effective rule $\langle y, R' \rangle$, therefore, has simple left-hand and right-hand sides. As a consequence, adaptive star grammars of the sort defined above can be simulated by ordinary ones (i.e., those in [DHJM10]) by using subsets of $\bar{C}$ to label edges in stars (without parallel edges) and turning every rule into a finite number of rules (corresponding to the allowed quotients). The definition used here just simplifies writing star grammars without extending their power.

3 Class Diagrams

Class diagrams are a well-known graphical language of the UML. The focus of this paper is on the specification of graphs without attributes. Hence, we ignore method and attribute specifica-

[1] In that paper, we consider conditions under which a late replicator is *minimal*; this is not necessary here.

tions within classes. They can be easily added if required. Moreover, we ignore associations with cardinality greater than 2.

The class diagrams used in this paper consist of *classes* and (binary) *associations* between them. *Concrete* classes are distinguished from *abstract* classes. The former are drawn as rectangles with white background, the latter with gray background and italic class names. Each association has a name and is drawn as a directed edge in order to distinguish the association end-points.[2] Each end-point of an association is equipped with a *multiplicity* of the form $u..v$ where $u \in \mathbb{N}_0$ is the lower bound and $v \in \mathbb{N} \cup \{*\}$ the upper bound such that $u \leqslant v$ if $v \in \mathbb{N}$. As usual in this context, $*$ stands for "infinity". An association may be declared *unique* (the default) or *non-unique*, indicated by the annotation '{non-unique}'. We distinguish *regular* associations (the default) from *composite* associations. The latter have a black diamond at one of their end-points. The multiplicity at this end-point is either $0..1$ or $1..1$. The class at the end-point with the diamond is called *composite*, the class at the other end-point is called its *part*. Edges representing composite associations are always directed towards the part class, and they may be loops. However, no object can be part of two objects, and no object is (directly or indirectly) part of itself. Moreover, class diagrams may contain *generalization* arrows, which have triangular arrow heads and point from sub-classes to super-classes. Each class may have an arbitrary number of sub-classes and super-classes. However, generalization arrows are not allowed to form cycles. We usually extend the notion of sub-classes and super-classes to all classes that are reachable by chains of generalization arrows (including chains of length 0). We also adopt features of UML 2 and allow association ends to *redefine* or *subset* other association ends. These features are explained in Sections 4.4 and 4.5.

A class diagram specifies graphs by the mechanism of instantiation: A node is an *instance* of a class iff it is labeled with the class name. Note that each node is an instance of exactly one class. A more general concept is defined as follows: A node is called *member* of a class C iff the node is an instance of C or any of its sub-classes. An edge e is an *instance* of an association from a class C_1 to a class C_2 if e is labeled with the association label, and if the source and target nodes are members of C_1 and C_2, respectively. Moreover, each association defines a *multiplicity constraint*: Let a be an association from class C_1 to class C_2 with multiplicities $u..v$ at its source end-point and multiplicity $r..s$ at its target end-point. The multiplicity constraint of a is satisfied iff no member of C_1 has less than r or more than s outgoing a-instances as edges and if no member of C_2 has less than u or more than v incoming a-instances as edges.

The graph language specified by a class diagram consists of all graphs that satisfy the following conditions:

C_1: Each node is an instance of a concrete class, and each edge is an instance of an association.

C_2: The multiplicity constraints of all associations are satisfied.

C_3: No two instances of any unique association are indistinguishable edges.

C_4: The subgraph induced by composite edges (i.e., instances of composite associations) must be a collection of trees.

[2] These directed edges should not be confused with navigation arrows, which solely represent implementation issues in object-oriented programming.

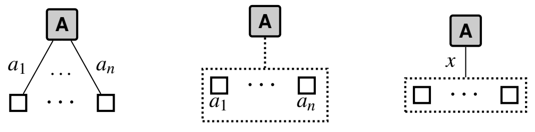
Such graphs are called *instance graphs* in the following. This definition follows the UML specification when nodes are considered as objects and edges as links [OMG].

4 An Adaptive Star Grammar for Instance Graphs

In the following, we assume an arbitrary, but fixed class diagram. We first describe how this class diagram can be translated into an ASG defining the same graph language. The set of terminal node sorts, hence, must consist of the names of all concrete classes. The set of edge sorts must contain all association names, but also additional edge sorts that will be used for labeling edges between nonterminal and terminal nodes. These edge sorts, but also nonterminal nodes sorts, will be introduced as needed in the following.

The idea of translating a class diagram into an adaptive star grammar defining the same graph language is to start from a graph (called *actual initial graph*, AIG, in the following) that closely resembles the class diagram. We add an initial rule that derives the AIG from the initial star that consists of just a single node with a unique nonterminal label. For each concrete class, the AIG contains a multiple node labeled with the class name. Cloning these nodes in a derivation creates the instances of the corresponding classes. The AIG's other nodes are nonterminal nodes representing the associations of the class diagram: Each *regular* association is represented by a single node with a unique nonterminal label. It is connected to all multiple nodes of those concrete classes that participate in the association, possibly by being a sub-class of a class at an endpoint of the association. A set of rules is responsible for eventually creating all edges representing instances of the association, i.e., links between objects. The rules make sure that conditions C_2 and C_3 (see Sect. 3) are satisfied. The construction is more complicated for *composite* associations since it must make sure that condition C_4 is not violated. Actually, each maximal connected subgraph of the class diagram consisting of composite associations and generalizations only has to be represented by a nonterminal node with a unique label. Rules must be defined that create all possible trees consistent with the class diagram.

These constructions are described in the following. But first, we introduce some notation that makes drawing of rules easier: In rules, corresponding nodes of the left- and right-hand sides are associated by their positions in the drawing. Terminal nodes in rules are always unlabeled; on the left-hand side of a rule, they match terminal nodes with arbitrary labels. Because none of the following rules introduces new terminal nodes on the right-hand side (new terminal nodes are rather created by cloning), there is no need for labeling them. Finally, we use a simplified notation for bundles in graphs as shown in the following illustration: For edge labels $a_1, \dots, a_n$, bundle (a) is drawn as (b), and as (c) if $a_1 = \cdots = a_n = x$.

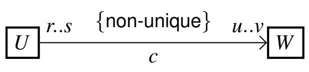


(a) (b) (c)

The next two sub-sections describe the construction for regular associations, the first one for associations declared "non-unique", the second for "unique associations". The presented construction must be repeated for each regular association within the class diagram. Afterwards, we discuss the construction for composition associations, and finally present an example.

4.1 Non-Unique Regular Associations

Consider an association

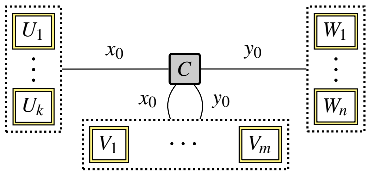


where $r, s, u, v \in \mathbb{N}_0, r \leqslant s, u \leqslant v$

within the class diagram. Classes U and W may be arbitrary classes, U and W may even reference the same class, or U may be a sub-class of W or vice versa. We assume the upper bounds s and v to be finite, i.e., different from "$*$". However, the following construction can be extended easily to the case $s = *$ and/or $v = *$ as we will discuss briefly at the end of this subsection.

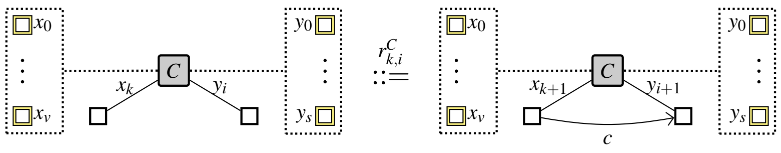
Let $\mathbb{U}$ and $\mathbb{W}$ be the sets of concrete sub-classes of U and W, respectively. Note that $\mathbb{U}$ and $\mathbb{W}$ contain the classes U and W, respectively., if they are concrete. Note also that $\mathbb{U} \cap \mathbb{W}$ contains all concrete classes being sub-classes of both U and W. Each instance of a class in $\mathbb{U} \setminus \mathbb{W}$ has $u..v$ outgoing, but no incoming c-edges, each instance of a class in $\mathbb{W} \setminus \mathbb{U}$ has $r..s$ incoming, but no outgoing c-edges, and each instance of a class in $\mathbb{W} \cap \mathbb{U}$ has $u..v$ outgoing as well as $r..s$ incoming c-edges. Since the association is non-unique, indistinguishable c-edges are permitted. The following construction assures these properties.

Let $\{U_1, \ldots, U_k\} = \mathbb{U} \setminus \mathbb{W}$, $\{W_1, \ldots, W_n\} = \mathbb{W} \setminus \mathbb{U}$, and $\{V_1, \ldots, V_m\} = \mathbb{U} \cap \mathbb{W}$. Note that each of these sets may be empty. Let C be a nonterminal label that is unique for this c-association, and let $\{x_0, x_1, \ldots, x_v\} \cup \{y_0, y_1, \ldots, y_s\}$ be a set of pairwise distinct edge labels. The AIG must contain the following graph as a subgraph:



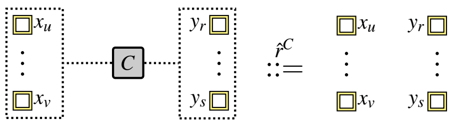
The meaning of the edges labeled x_k and y_i is the following: An x_k-edge connects C with a terminal node that has k outgoing c-edges already, and an y_i-edge connects C with a terminal node that has i incoming c-edges already. Note that terminal nodes with labels in $\mathbb{W} \cap \mathbb{U}$ may have incoming as well as outgoing c-edges; they may even have c-loops where each loop counts as an incoming and as an outgoing edge. The number of incoming and outgoing edges of terminal nodes with labels in $\mathbb{W} \cap \mathbb{U}$ is indicated by two edges labeled x_k and y_i, respectively.

We may add another c-edge between two appropriate nodes where the source node must have $k < v$ outgoing c-edges and the target node $i < s$ incoming c-edges. This is the task of the following set of rules $r^C_{k,i}$, $0 \leqslant k < v$ and $0 \leqslant i < s$, which add a new c-edge between two nodes and increment the "counters" realized by edge labels.



Note that such a rule can also be applied in situations where nodes are connected with two parallel edges labeled x_k and y_i. The corresponding border nodes have to be identified in such cases. A c-loop is added if the two singular border nodes are identified.

We can stop adding c-edges if there is no x_k-edge and no y_i-edge with $k < u$ and $i < r$; we can then remove the nonterminal node C representing the association. This is realized by the following final rule $\hat{r}^C$:

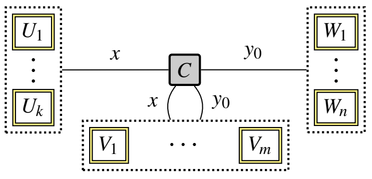


The presented construction uses the fact that v and s are finite numbers and different from $*$. The construction, however, is extended easily to the case $v = *$ and/or $s = *$. Let us assume just $v = *$. The new meaning of an x_u-edge connecting C with a terminal node is that the terminal node has *at least* u outgoing c-edges already. And we change rule $r^C_{u,i}$, so that it no longer creates a new x_{u+1}-edge, but an x_u-edge again. It is clear that this construction now creates any number of outgoing c-edges, but at least u, at appropriate nodes.

4.2 Unique Regular Associations

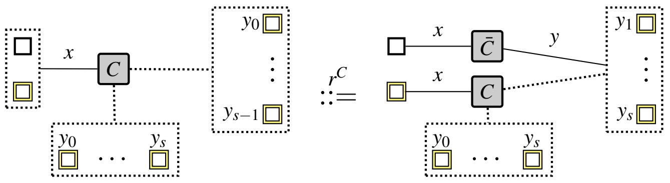
We now consider the slightly different situation with a *unique* instead of a *non-unique* association; the other aspects remain unchanged.

Let C and $\bar{C}$ be two nonterminal labels that are unique for this c-association, and let $\{x, y_0, y_1, \ldots, y_s\}$ be a set of pairwise distinct edge labels. The AIG must contain the following graph as a subgraph:

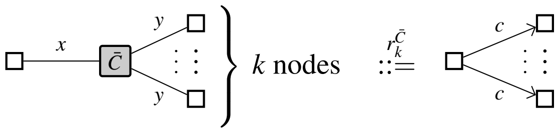


The meaning of y_i-edges is the same as in the previous section, i.e., a terminal node connected to C with a y_i edge means that the terminal node has i incoming c-edges already. However, we do not count outgoing c-edges. Instead, an x-edge between a terminal node and C means that the terminal node has not yet any outgoing c-edge.

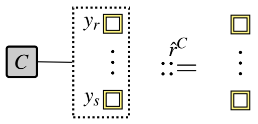
Since indistinguishable c-edges are prohibited, we cannot create c-edges independently from each other like in the previous subsection. Instead, we use rules that create all outgoing c-edges of any singular node in a single derivation step; an x-edge visits those terminal nodes that receive a bundle of outgoing c-edges to pairwise distinct nodes visited by y_i-edges. We define a rule r^C that selects one singular node visited by an x-edge and some of the terminal nodes visited by y_i-edges. The latter nodes will receive new incoming c-edges. Rule r^C, therefore, increments the "counters" realized by y_i-edges:



The rule adds a new nonterminal node labeled with $\bar{C}$. The following set of rules $r_k^{\bar{C}}$, $u \leqslant k \leqslant v$, adds a bundle of c edges from the node visited by the x-edge to the k nodes visited by the y-edges.



Finally, we can remove the C-node as soon as all terminal nodes connected to C with an x-edge have been processed by rule r^C. This is the task of rule $\hat{r}^C$:

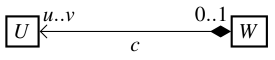


Like in the previous subsection, these rules use the fact that the upper multiplicity bounds v and s are finite and different from *. The case $s = *$ can be realized similarly to the discussion at the end of the previous subsection. As an example, see Sect. 4.7. The case $v = *$ is even simpler: The set of rules $r_k^{\bar{C}}$ is replaced by a single rule $r^{\bar{C}}$ that looks like $r_{u+1}^{\bar{C}}$; however, one of the border nodes visited by an y-edge is turned into a multiple node. Hence, this rule can add bundles with an arbitrary number of c-edges, but at least u.

4.3 Composite Associations

With the constructions of the last two subsections, each class diagram whose associations are all regular can be translated into an adaptive star grammar defining the same graph language. Composite associations are special since subgraphs induced by composite edges must be collections of trees (condition C_4 in Sect. 3). The ASG, hence, must be able to create all such trees. In order to translate this part of a class diagram into an ASG, we have to find those classes whose instances can belong to the same tree. This is done by finding the largest connected subgraphs of the class diagram that contain only composite associations and generalizations. The instance nodes of the classes that belong to the same connected subgraph may, but need not, belong to the same tree. This can be represented in the ASG in the following way: We add a new nonterminal node to the AIG for each of these connected subgraphs, and connect this nonterminal node to each multiple node representing a concrete class within the subgraph. Then we define star rules that create all possible trees consistent with the class diagram.

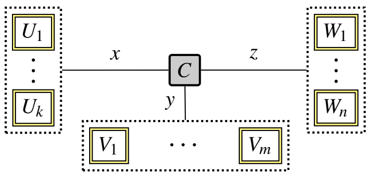
We demonstrate this procedure for just a single composite association as shown here. However, the construction can be generalized easily to more composite associations in the same connected subgraph.



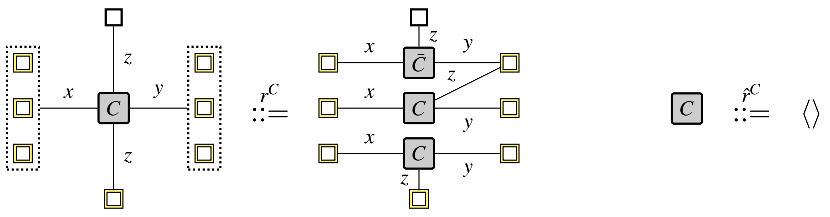
where $u, v \in \mathbb{N}_0, u \leqslant v$

Again, classes U and W may be arbitrary classes, U and W may even reference the same class, or U may be a sub-class of W or vice versa.

Let the sets $\mathbb{U}$, $\mathbb{W}$, $\{U_1,\dots,U_k\} = \mathbb{U} \setminus \mathbb{W}$, $\{W_1,\dots,W_n\} = \mathbb{W} \setminus \mathbb{U}$, and $\{V_1,\dots,V_m\} = \mathbb{U} \cap \mathbb{W}$ be defined as in the beginning of Sect. 4.1. Let $C, \bar{C}$ be two nonterminal labels that are unique for this c-association, and let x, y, z be distinct edge labels. The AIG must contain the following graph as a subgraph:

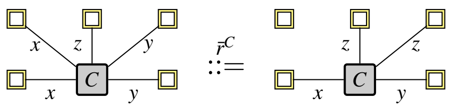


The nonterminal node C stands for all trees with instances of classes in $\mathbb{U} \cup \mathbb{W}$ as nodes connected by c-edges; z-edges visit those terminal nodes (roots) that will "receive" $u..v$ outgoing, but no incoming c-edges. Nodes visited by y-edges (inner nodes) will "receive" $u..v$ outgoing and possibly one incoming c-edge. Finally, nodes visited by x-edges (leaves) will "receive" at most one incoming, but no outgoing c-edge. These properties are assured by the following rules.

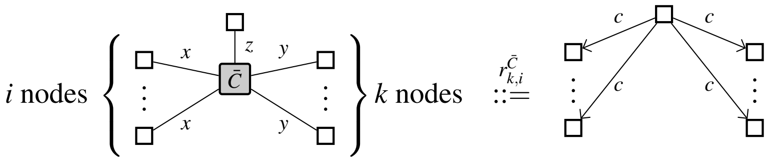


The recursive rule r^C selects a singular root (visited by a z-edge) and recursively proceeds with the rest of the roots by adding a new C-node connected to the multiple node representing the rest of the roots. Moreover, r^C creates a $\bar{C}$-node that will be derived by a rule $r^{\bar{C}}_{k,i}$ (see below) to a bundle of c-edges from the root to its children. Those children that are inner nodes become roots of sub-trees, indicated by the other created C-node. The recursion stops as soon as all nodes have been processed, represented by rule $\hat{r}^C$, which replaces a C-node by the empty graph $\langle\rangle$.

The rules r^C and $\hat{r}^C$ make sure that each member of U receives an incoming c-edge. However, this is actually not required by the association. Rule $\bar{r}^C$, therefore, allows to turn any inner node into a root and any leaf into a singleton tree node. This rule must be dropped if the multiplicity of association c is 1..1 instead of 0..1 at the composite end-point.



Finally, $\bar{C}$-nodes must be derived to bundles of $u..v$ many c-edges. This is the task of the set of rules $r^{\bar{C}}_{k,i}$ for all $i,k \in \mathbb{N}_0$ so that $u \leqslant i+k \leqslant v$:

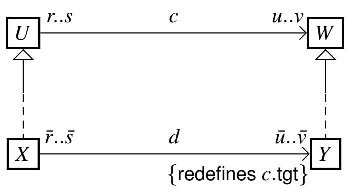


Note that this construction can be easily extended to the situation where $v = *$. The set of rules $r^{\bar{C}}_{k,i}$ must be replaced by the set of rules $r'^{\bar{C}}_{k}$ such that $0 \leqslant k \leqslant u$; $r'^{\bar{C}}_{k}$ looks like $r^{\bar{C}}_{k+1,u-k+1}$, but two of the border nodes, one being visited by an x-edge, the other with an y-edge, are turned into multiple nodes. Hence, these rules can add bundles with an arbitrary number of c-edges, but at least u.

4.4 Class Diagrams with Redefining Associations

Class inheritance, i.e., classes that are specializations of others, is one of the original object-oriented concepts. UML 2 [OMG] has extended class diagrams by the concept of specialization for associations. In this paper, we consider *redefinition* and *subsetting*. The former is described in this section, the latter in the following.

The following class diagram shows an example of a redefined association end.



An end of an association c can be redefined by an end of an association d if d is a specialization of c, i.e., the associated classes must be compatible. This is indicated by the dashed generalization arrows, meaning that X has to be a sub-class[3] of U and Y a sub-class of W. The constraint {redefines c.tgt} marks the redefining association end and declares the redefined association end, here the target end of c. A role name would be used in regular class diagrams. In this paper, we ignore role names and, use src or tgt instead.

The meaning of redefined associations is as follows: Associations with their ends correspond to class properties that describe links from objects to objects at the other association ends. Redefinition of association ends actually restricts what objects may be linked. As specified by association c, each instance of class U and its sub-classes may be linked to instances of class W or its sub-classes. In particular, instances of X (and its sub-classes) may be linked to W-objects by association c. The redefinition of c.tgt means that each W-object being linked from an X-object by association C must actually be a Y-object. These Y-objects are directly accessible through association d; each d-link is a c-link at the same time. Note, however, that the source end of d does not redefine the source end of c in our class diagram. This means that U-objects

[3] As introduced in Section 3, we extend the notion of sub-classes to all classes that are reachable by chains of generalization arrows, including chains of length 0.

which are not X-objects may still link to Y-objects. If the source end of d were annotated with {redefines c.src}, only X-objects would be allowed to be linked to Y objects by association c.

In the following we show how the situation presented in the class diagram shown above can be translated into an ASG such that the links in each instance graph comply with the association end redefinitions. We will consider non-unique associations only. Unique associations or multiple redefined association ends can be translated analogously.

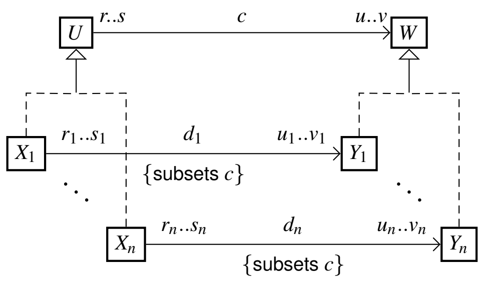
Translation of redefinition semantics to ASGs is actually simple. The scheme for setting up an ASG for the given class diagram is the same as described in Section 4.1. We handle associations c and d as if they were completely independent associations. For c, however, we have to set up the AIG in a more restricted way such that no c-edges can be created between X-nodes and Y-nodes. This can be simply enforced by removing each concrete sub-class of X from set $\mathbb{U}$ first and then by proceeding as described in Section 4.1.

Note that we use edge label d for representing links of association d, which are c-links at the same time. This information is implicit and cannot be represented in instance graphs explicitly.

4.5 Class Diagrams with Association Subsetting

Subsetting of association ends is another extension to associations that has been introduced by UML 2. If an association end may redefine another association end as shown in the previous section, it may also subset it instead. Subsetting association ends do not restrict the subsetted associations, in contrast to redefining association ends. A subsetting association end simply means that each link of the subsetting end must be a link of the corresponding subsetted association end, too. It is easy to see that subsetting one end of an association automatically implies the subsetting of the other association end. Therefore, we rather say that an association subsets another association instead of association ends subsetting other association ends (as done in the UML standard [OMG]).

The following class diagram shows the general situation of an association c which is subsetted by the associations $d_1, d_2, \dots d_n$. The classes at the association ends must be compatible to U and W, respectively, indicated by the dashed generalization arrows.



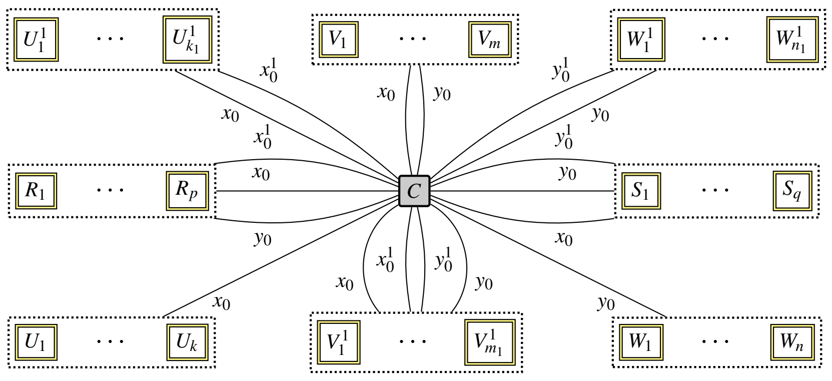
Similar to the previous section, we are going to represent d_t-links by d_t-edges. Subsetting means that each of these d_t-links is a c-link at the same time. Again, this is implicit information that cannot be represented in the instance graph explicitly. Also similar to the previous section, we consider non-unique associations only. Unique associations can be translated analogously.

Since subsetting associations do not restrict subsetted associations, we can create c-edges and d_t-edges almost independently in the ASG. When a d_t-edge is created, however, we must count it as a c-edge, too, since each d_t-link is also a c-link. Counting is necessary if multiplicity is restricted. For instance, if no U-object may be linked to more than two W-objects using association c, i.e., $v = 2$, we must not have an X_1-object with one leaving c-edge and two leaving d_1-edges. Similar to Section 4.1, we use a nonterminal node C for creating all edges labeled $c, d_1, d_2, \dots, d_n$. C is connected with terminal nodes by edges labeled x_k and y_i iff the corresponding terminal node has already k outgoing and i incoming c-edges, respectively. We use edges labeled x_k^t and y_i^t iff the corresponding terminal node has already k outgoing and i incoming d_t-edges, respectively. The AIG has to contain C and one multiple node for each concrete sub-class of U and W. Nodes representing sub-classes of X_t ($t = 1, 2, \dots, n$) are connected with C by x_0^t-edges. Nodes representing sub-classes of W and Y_t ($t = 1, 2, \dots, n$) are connected with C by edges labeled y_0 and y_0^t, respectively. Note that some classes in the class diagram may actually be the same classes or they may inherit from several classes. The corresponding nodes are then connected with C by several edges.

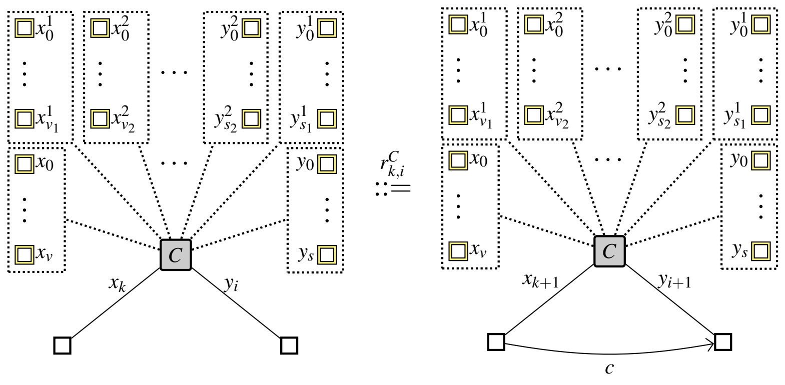
As an example, we choose $n = 1$, i.e., we have one subsetting association only. Similar to Section 4.1, let $\mathbb{U}, \mathbb{W}, \mathbb{X}, \mathbb{Y}$ be the sets of concrete sub-classes of U, W, X_1, Y_1, respectively. Note that $\mathbb{X} \subseteq \mathbb{U}$ and $\mathbb{Y} \subseteq \mathbb{W}$. Note also that $\mathbb{U}, \mathbb{W}, \mathbb{X}, \mathbb{Y}$ contain U, W, X_1, Y_1, respectively, if they are concrete classes. Each concrete sub-class of U or W then falls into exactly one of the following sets:

$$\begin{aligned}
\{U_1, \dots, U_k\} &= (\mathbb{U} \setminus \mathbb{X}) \setminus \mathbb{W} \\
\{W_1, \dots, W_n\} &= (\mathbb{W} \setminus \mathbb{Y}) \setminus \mathbb{U} \\
\{U_1^1, \dots, U_{k_1}^1\} &= \mathbb{X} \setminus \mathbb{W} \\
\{W_1^1, \dots, W_{n_1}^1\} &= \mathbb{Y} \setminus \mathbb{U} \\
\{R_1, \dots, R_p\} &= (\mathbb{W} \setminus \mathbb{Y}) \cap \mathbb{X} \\
\{S_1, \dots, S_q\} &= (\mathbb{U} \setminus \mathbb{X}) \cap \mathbb{Y} \\
\{V_1, \dots, V_m\} &= (\mathbb{U} \setminus \mathbb{X}) \cap (\mathbb{W} \setminus \mathbb{Y}) \\
\{V_1^1, \dots, V_{m_1}^1\} &= \mathbb{X} \cap \mathbb{Y}
\end{aligned}$$

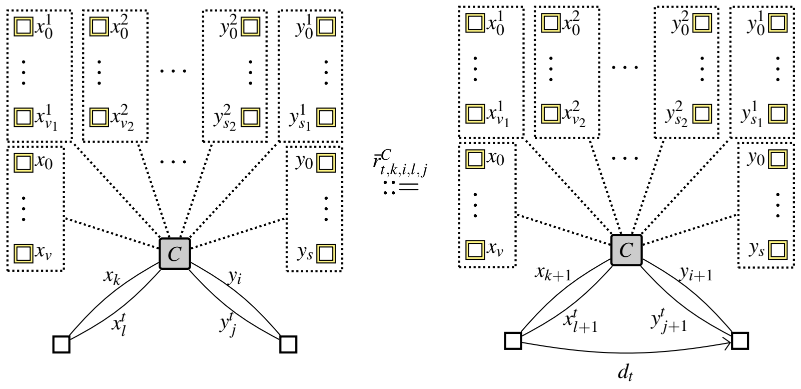
and the AIG, therefore, must contain the following graph as a subgraph:



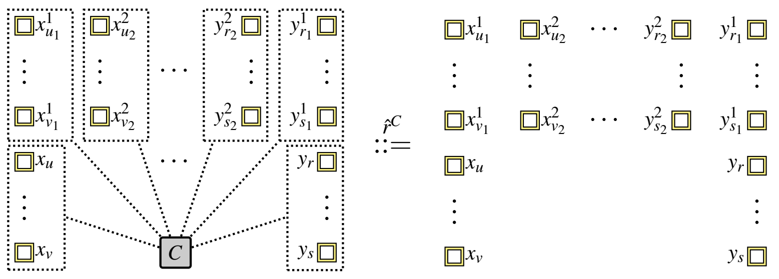
Like in Section 4.1, we may add another c-edge between two appropriate nodes where the source node must have $k < v$ outgoing c-edges and the target node $i < s$ incoming c-edges. This is the task of the following set of rules $r_{k,i}^C$, $0 \leqslant k < v$ and $0 \leqslant i < s$, that add a new c-edge between two nodes and increment the “counters” realized by edge labels.



Analogously, we may add another d_t-edge ($t = 1,2,\ldots,n$) between two appropriate nodes where the source node must have $l < v_t$ outgoing d_t-edges and the target node $j < s_t$ incoming d_t-edges. However, since each d_t-link is also a c-link, we must count this new d_t-edge as a c-edge, too. Therefore, the source node must have $k < v$ outgoing c-edges and the target node $i < s$ incoming c-edges. This is the task of the following set of rules $\bar{r}^C_{t,k,i,l,j}$ ($0 \leqslant k < v, 0 \leqslant i < s, 0 \leqslant l < v_t, 0 \leqslant j < s_t$, and $0 \leqslant t \leqslant n$) that add a new d_t-edge between two nodes and increment the “counters” realized by edge labels.



We can stop adding c-edges and d_t-edges if each “counter” has reached its lower multiplicity bound; we can then remove the nonterminal node C representing the association. This is realized by the following final rule $\hat{r}^C$:

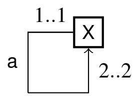


If a class acts as a root of an inheritance hierarchy and must not be instantiated, it is declared as an abstract class. A similar concept exists in UML 2 for associations: Subsetted associations (association c in our class diagram) that must not be instantiated are declared as $\{$union$\}$. Only their subsetting associations (associations d_t, $t = 1, 2, \ldots, n$ in our class diagram) may be instantiated as long as these are not themselves declared as $\{$union$\}$. Union-associations can easily be handled in the construction of the grammar by prohibiting the addition of c-edges, i.e., by dropping the rules $r^C_{k,i}$.

4.6 Class Diagrams with Empty Graph Languages

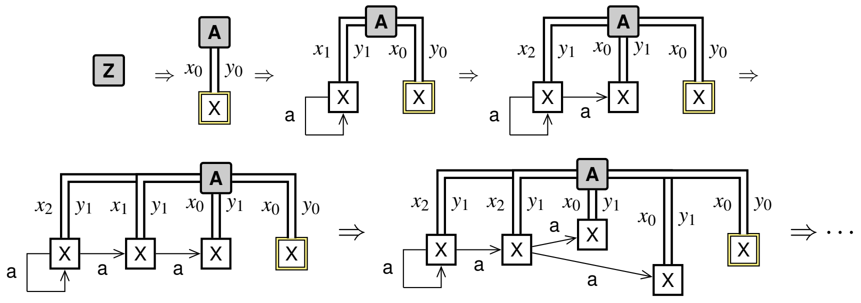
There are class diagrams that cannot be satisfied by any instance graph, i.e., its graph language is empty. This section demonstrates that the ASG constructed from such a class diagram has an empty language, too.

As an example, consider the following class diagram.



There is no (finite) instance graph compatible with this class diagram because it requires each instance graph node to have an in-degree of 1 and an out-degree of 2 at the same time.

Of course, the ASG constructed from this class diagram has an empty graph language, too. Each derivation starting from the initial star **Z** is infinite. Each derived graph contains a nonterminal node; the terminating rules can never be applied. Such a derivation is shown below; every graph wherein n class nodes satisfy the multiplicity constraints will contain at least n other classes that do not satisfy it.

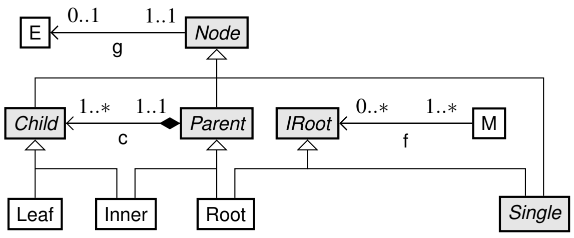


Interestingly, while there is no finite instance of the class diagram, there are, in fact, infinite instances. This fits nicely to the fact that derivation sequences of the ASG are infinite and “generate” these infinite graphs.

4.7 Example

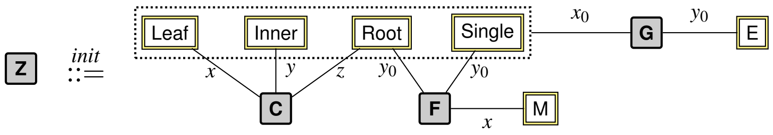
Finally, this section presents the rules constructed from an example class diagram and shows the derivation of an instance graph.

Consider the following class diagram that specifies trees where each tree node may be connected to an E-object. “Manager”-objects of class M are connected to root nodes:

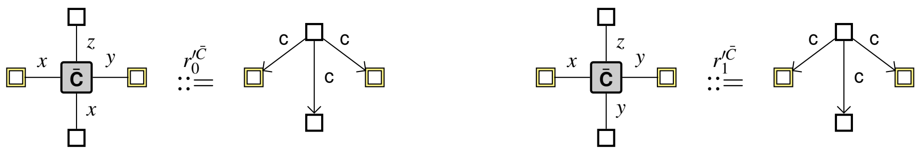


We assume that the association f is unique, but that g is not (which is actually irrelevant for an m-to-1 association). The association c is a composite association.

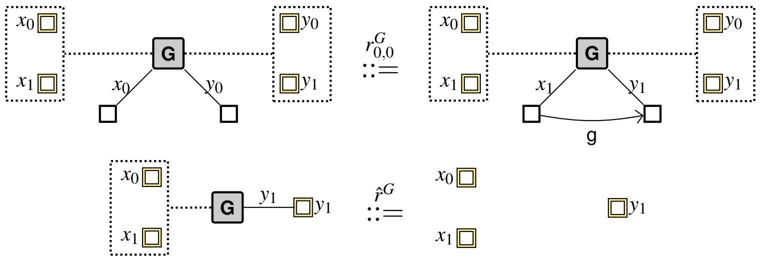
Then the initial rule for the initial star **Z** deriving the AIG looks as follows:



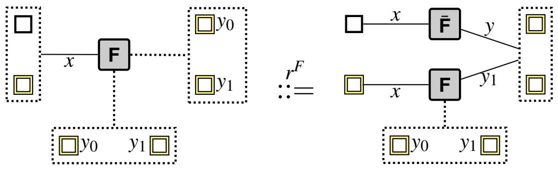
The rules for the composition c are r^C and $\hat{r}^C$ as in Sect. 4.3, but without $\bar{r}^C$ since the lower bound of the association at the composite end-point is 1. Because the upper bound at the part end-point is $*$, we need the following rules for $\bar{C}$:

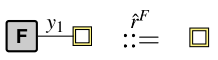


The rules for association g are constructed as shown in Sect. 4.1 with $r = s = v = 1$ and $u = 0$:



The rules for association f are constructed as described in Sect. 4.2 with $r = 1, u = 0, s = v = *$. Because of $s = *$, rule r^F differs from the one shown in Sect. 4.2: an y_1-edge means that the connected terminal node has at least one incoming f-edge. Moreover, as discussed at the end of Sect. 4.2, there is only a single rule $r^{\bar{F}}$ creating 0..$*$ f-edges from an $\bar{F}$-node:





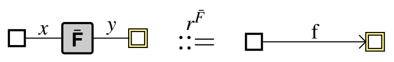


Fig. 1 shows a sample derivation of an instance graph using these rules. Note that node labels have been abbreviated by initial letters of class names. The derivation arrows $d_1, \ldots, d_9$ do not represent single derivation steps, but have the following meaning: d_1 clones the multiple R-node to a singular as well as another multiple one, clones the I- and L-nodes three times each, and applies rule r^C. d_2 clones the topmost I-node to a singular and another multiple one, deletes the R-, L-, and I-nodes at the bottom (by replicating them 0 times), and applies r^C again after cloning the remaining I- and L-nodes three times each. d_3 deletes the 6 lowermost multiple nodes and clones the remaining two multiple L-nodes to singular ones. d_4 applies $\hat{r}^C$ twice, deleting the isolated C-nodes, and then $r_0'^{\bar{C}}$ twice after adapting the rule to the context of the corresponding $\bar{C}$-node. d_5 applies $r^G_{0,0}$, and d_6 applies $\hat{r}^G$, deleting the G-node. d_7 applies r^F; both the R- and the S-node get visited by y_1-edges. d_8 deletes the multiple M-node by replicating it 0 times, and applies $\hat{r}^F$, deleting the F-node. Finally, d_9 clones the S-node twice and applies $r^{\bar{F}}$.

5 Conclusions

We have described how a class diagram can be translated into an adaptive star grammar that defines the same language of instance graphs as the class diagram. The translation process works for all class diagrams with generalizations, and with associations that may be unique, non-unique, or even composite, and may have arbitrary multiplicities. We have explicitly shown the translation process for regular associations with arbitrary multiplicities, have outlined the translation for compositions, and demonstrated the situation for a single composition association. Furthermore, we have shown that newer concepts of class diagrams, like association subsetting and redefinition, can also be translated correctly.

The presented approach closes the gap between class diagrams as a declarative approach for defining instances and graph grammars, which provide a constructive definition. This makes techniques known from graph grammars available to class diagrams as well. For instance, test cases can be constructed automatically by just creating sample derivations. Furthermore, grammars allow for inductive definitions based on their rules and their derivation trees that structure instances of the class diagram hierarchically. Moreover, one may wish to restrict the instances of a class diagram in ways that class diagrams cannot express. The translation into an equivalent grammar may make this possible, because grammars allow for a much finer tuning.

The benefits of using adaptive star grammars come with a moderate descriptive complexity. If association subsetting is not used, the size of each rule (by counting nodes and edges on a rule's left-hand side and right-hand side) depends linearly on the maximum finite multiplicity bound m used in the class diagram, and the number of obtained rules grows linearly with the number of associations and quadratic with m. For class diagrams with association subsetting, the descriptive complexity is worse. Let N_s be the number of associations participating in association subsetting. Then, the size of rules obtained for such associations depends linearly on m and N_s, whereas the number of rules grows linearly with N_s and with the fourth power of m.

The only closely related work, to the best of our knowledge, is by K. Ehrig, J. M. Küster, and G. Taentzer [EKT09], which extends the work by R. Bardohl, H. Ehrig, J. de Lara, and

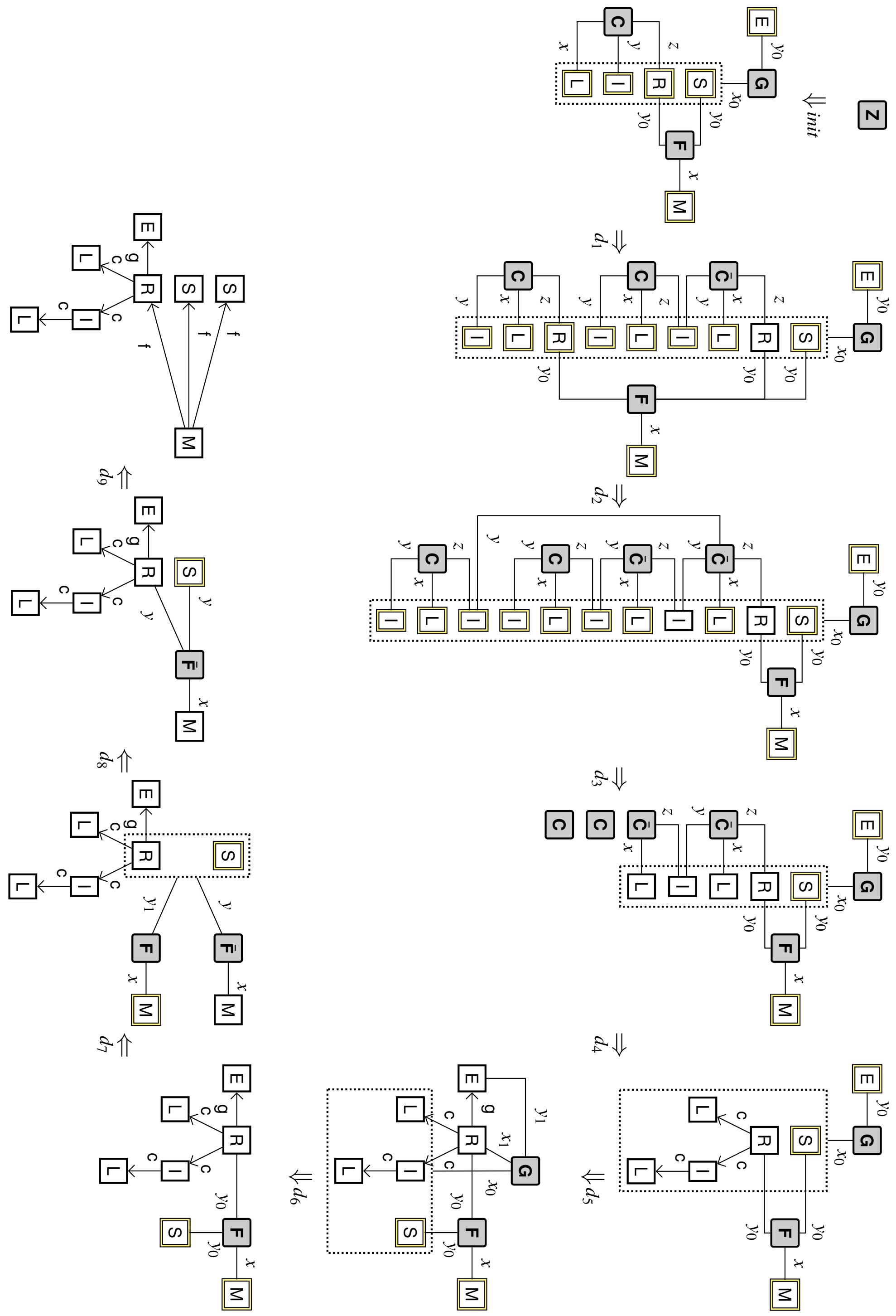


Figure 1: A sample derivation of an instance graph (to be read clockwise)

G. Taentzer [BELT04] considerably. In [EKT09], a meta model is translated into a graph grammar that defines the same language of instance graphs. In that paper, composite associations are not considered, multiplicities are restricted to the simplest cases, and unique and non-unique associations are not distinguished. Moreover, the generated graph grammar makes heavy use of negative application conditions for specifying forbidden context graphs, and uses a layering mechanism to give some rules priority over others. This does not only complicate the grammar, but—more important—also the reasoning about the grammar and the graph language it generates.On the other hand, this kind of grammar allows to treat some simple meta model constraints by translating them into application conditions.

So far, we have not considered how meta model constraints can be translated along with the class diagram into an adaptive star grammar, possibly with application and graph conditions. A related, but not general approach has been described in previous work [HM10] where rules may be applied only if their application conditions are satisfied. In future work, we intend to make use of generalized results on context conditions and their relation to logical graph properties and constraints by A. Habel and K.-H. Pennemann [HP09] and A. Habel and H. Radke [HR10].

Acknowledgement. We would like to thank the anonymous referees for reading the submitted version carefully, and making valuable suggestions that helped to improve the paper.

Bibliography

[BELT04] R. Bardohl, H. Ehrig, J. de Lara, G. Taentzer. Integrating Meta-modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In *Proc. Fundamental Approaches to Software Engineering (FASE'04)*. Lecture Notes in Computer Science 2984, pp. 214–228. Springer-Verlag, 2004. doi:10.1007/978-3-540-24721-0_16

[Cou87] B. Courcelle. An Axiomatic Definition of Context-free Rewriting and its Application to NLC rewriting. *Theoretical Computer Science* 55(2-3):141–181, 1987. doi:10.1016/0304-3975(87)90102-2

[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. doi:10.1007/11841883_7

[DHJM10] F. Drewes, B. Hoffmann, D. Janssens, M. Minas. Adaptive Star Grammars and Their Languages. *Theoretical Computer Science* 411(34-36):3090–3109, 2010. doi:10.1016/j.tcs.2010.04.038

[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. doi:10.1007/978-3-540-87405-8_30

[EKT09] K. Ehrig, J. M. Küster, G. Taentzer. Generating Instance Models from Meta Models. *Software and System Modeling* 8(4):479–500, 2009.
doi:10.1007/s10270-008-0095-y

[Eng99] J. Engelfriet. Context-Free Graph Grammars. In Rozenberg and Salomaa (eds.), *Handbook of Formal Languages*. Volume 3: Beyond Words, chapter 3, pp. 125–213. Springer, 1999.

[HM10] B. Hoffmann, M. Minas. Defining Models – Meta Models versus Graph Grammars. *Elect. Comm. of the EASST* 29, 2010. Proc. 6th Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'10), Paphos, Cyprus.
http://www.easst.org/eceasst/

[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.
doi:10.1017/S0960129508007202

[HR10] A. Habel, H. Radke. Expressiveness of graph conditions with variables. *Elect. Comm. of the EASST* 30, 2010. International Colloquium on Graph and Model Transformation (GraMoT'10).
http://www.easst.org/eceasst/

[OMG] OMG. OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.2. OMG Document Number: formal/2009-02-04.
http://www.omg.org/spec/UML/2.2/Infrastructure

[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.