Electronic Communications of the EASST
Volume 18 (2009)

Proceedings of the
Eighth International Workshop on
Graph Transformation and Visual Modeling Techniques
(GT-VMT 2009)

Aspects for Graph Grammars

Rodrigo Machado, Luciana Foss and Leila Ribeiro

13 pages

Guest Editors: Artur Boronat, Reiko Heckel
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Aspects for Graph Grammars

Rodrigo Machado[1], **Luciana Foss**[1,2] **and Leila Ribeiro**[1]

[1] (rma, leila)@inf.ufrgs.br
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
[2] luciana.foss@ufpel.edu.br
Instituto de Física e Matemática - Departamento de Informática
Universidade Federal de Pelotas
Pelotas, Brazil

Abstract: Aspect-oriented programming (AOP) is an extension to the object oriented paradigm that aims to provide better modularity for code that is usually scattered across an object-oriented system such as logging, authentication and distributed object handling. Aspect weaving is a novel way to compose systems, focusing on the integration of system-wide policies through pattern-action rules. While there are several semantic proposals for representing aspects over source code and programs, aspect weaving for visual models such as graph rewriting systems is still not fully established. In this work, we propose the definition of aspect-oriented graph grammars, an extension to conventional graph grammar where aspects are modeled as transformation rules over the structure of a base graph grammar.

Keywords: aspect-oriented software development, graph grammars, double-pushout approach.

1 Introduction

Aspect-oriented programming (AOP) [KLM+97] is an extension to the object oriented paradigm that aims to provide better modularity for code that is scattered across object-oriented systems, such as logging, authentication and distributed object handling. The main idea of the paradigm is to encapsulate the statements that deal with such situations in a module called aspect. Inside the aspect there are rules (advices) that describe how these statements should be weaved into the base code. Every advice actuates over a specific set (pointcut) of system execution points (join points), executing some action *before*, *after* or *in place of* the join point.

Aspect-orientation can be seen as a kind of *meta-programming* that allows one to describe system-wide behaviors in a compact notation. Since its proposal in the late 90s, the paradigm has been gaining acceptance and development tools. Although there are several proposals [WZL03, JJR06, DDFB06, CL06] to describe the operational effect of aspect weaving over programs, the weaving of aspects over visual models is still not totally established.

The fact that several visual languages can be naturally modeled using graphs makes graph grammars an appealing formalism to define the semantical models for such languages. A graph grammar (GG) [Roz97] is a model in which the state is represented by a graph and system evo-

lution is represented by graph rewriting productions. Several interesting models for computation and software development, such as UML diagrams, have a natural graph-based interpretation, and thus can be naturally modeled by means of GGs [HET08].

In this work we address the issue of crosscutting concerns in graph grammars, and propose the definition of *graph aspects* to modularize their treatment. Our main contribution is the definition of aspect-oriented graph grammars (AOGG), where aspects are represented by a second-order transformation over the productions of a base GG. We also specify how aspects are combined to a base grammar, resulting in a *weaved* graph grammar. By defining formally aspects and aspect weaving over graph grammars, we also provide a semantic interpretation for aspect-oriented concepts over other models that are instances of GGs.

The rest of the text is organized as follows. Initially, in Section 2, we informally present the main concepts of the aspect-oriented paradigm. In Section 3, we review graph grammars and introduce our working example. Then, in Section 4, we discuss how to modify the example in order to implement system-wide policies such as logging. In Section 5, we provide a description of aspects over graph grammars and formally define aspect-oriented graph grammars. Finally, in Section 6, we compare our approach to other proposals, state our final remarks and present future work.

2 Aspect-Oriented Paradigm

The main purpose of using AOP is to spread some behavior automatically over the whole source code (or bytecode) of the application. The fundamental abstractions of the paradigm are the following: *i) join points:* execution points that can be affected by aspects; *ii) pointcuts:* specific sets of join points; *iii) advices:* rules that, given a pointcut, define some behavior to be triggered when the system reaches some of its join points; *iv) inter-type declarations:* extensions to the static structure of the system, which may be needed by the behavior introduced by the advices. *v) aspects:* modules containing all advices and inter-type declarations for dealing with a specific crosscutting concern.

In aspect-oriented programming languages, join points are generally defined as a subset of the system named transitions, like method calls and attribute accesses. Pointcuts are specific sets of join points, specified by means of a *pointcut language*. Advices substitute the join points that match its associated pointcut with some programmed behavior, which can also include the original behavior. The module that combines the aspects over the original base code is called *aspect weaver*. As a simple example of aspect weaving, consider the AspectJ source code depicted in Figure 1 (AspectJ is the most popular AOP extension for the Java programming language). The AspectJ weaver receives both the base code and the aspect code. Then, it applies the advices within the aspect, inserting the commands provided in the advices every time it finds their pointcuts in the base code. In the example of Figure 1, the aspects simply introduces a print command right before the start of the execution of any method without parameters sent to an object of class `A`. Although the result of the combination is shown as a source-code transformation, the AspectJ compiler actually performs byte-code level weaving, i.e. the aspect weaving occurs after the compilation of both base code and aspects.

Base code:

```
public class A {
  void a() { ... body of a ... }
  void b() { ... body of b ... }
  void c(int x) { ... body of c ... }
}
```

Aspect:

```
public aspect LogA{
  before() : execution ( * A.*() ) {
     System.out.println("Method without parameters\n");
  }
}
```

Weaved code = Aspect Weaver(Base code, Aspect):

```
public class A {
  void a() { System.out.println("Method without parameters\n");
             ... body of a ... }
  void b() { System.out.println("Method without parameters\n");
             ... body of b ... }
  void c(int x) { ... body of c ... }
}
```

Figure 1: Example of aspect weaving in AspectJ

3 Graph Grammars

A graph grammar (GG) is a visual model to represent systems. In a GG, the states of the system are graphs and the system behavior is defined by an starting graph together with a set of graph rewriting rules. In this section, we recall the basic concepts of GGs, according to the DPO (double-pushout) approach [C+97], and provide the working example to be used in the rest of the paper. We will use *typed graph grammars*, i.e. grammars where all states and rules are typed.

Definition 1 ((Typed) Graph and Graph Morphisms) A *graph* is a tuple $G = \langle V_G, E_G, s^G, t^G \rangle$, where V_G and E_G are sets of vertices and edges, and $s^G, t^G : E_G \to V_G$ are the source and target function. A *(total) graph morphism* $f : G \to G'$ is a pair of functions $(f_V : V_G \to V_{G'}, f_E : E_G \to E_{G'})$ such that $f_V \circ s^G = s^{G'} \circ f_E$ and $f_V \circ t^G = t^{G'} \circ f_E$. The category of graphs and total graph morphisms is called **Graph**. Let $T \in$ **Graph** be a fixed graph, called type graph, a T*-typed graph* G^T is given by a graph G and a (total) graph morphism $t_G : G \to T$. A morphism of T-typed graphs $f : G^T \to G'^T$ is a (total) graph morphism $f : G \to G'$ that satisfies $t_{G'} \circ f = t_G$. A typed graph G^T is called *injective* if the typing morphism t_G is injective. The category of T-typed graphs and T-typed graph morphisms is the comma category **Graph**$\downarrow T$, shortened by T-**Graph**.

Definition 2 (Graph Productions and Graph Grammars) A T*-typed (graph) production* (or graph rule) is a tuple $q : L_q \overset{l_q}{\leftarrowtail} K_q \overset{r_q}{\rightarrowtail} R_q$, where q is the name of the production, L_q, K_q and

R_q are T-typed graph, l_q and r_q are injective morphisms. The class of all T-typed graph production is denoted by T-**Prod**. A T-typed *graph grammar* is a tuple $\mathscr{G} = \langle T, P, \pi, G_0 \rangle$, where T is a type graph, P is a set of production names, π is a function mapping production names to productions in T-**Prod**, and G_0 is a T-typed graph, named the *initial graph*.

Definition 3 (Direct derivation and Derivations) Given a T-typed graph G, a T-typed graph production $q = L_q \stackrel{l}{\leftarrow} K_q \stackrel{r}{\rightarrow} R_q$ and a match (i.e. an injective T-typed graph morphism) $m : L_q \rightarrow G$, a *direct derivation* from G to H using q (based on m) exists if and only if the diagram below can be constructed, where both squares are pushouts in T-**Graph**. In this case the direct derivation is denoted by $\delta : G \stackrel{q,m}{\Rightarrow} H$ or $\delta : G \stackrel{q}{\Rightarrow} H$ if we do not make explicit m.

$$\begin{array}{ccccc} L_q & \stackrel{l}{\longleftarrow} & K_q & \stackrel{r}{\longrightarrow} & R_q \\ {\scriptstyle m}\downarrow & (1) & \downarrow{\scriptstyle k} & (2) & \downarrow{\scriptstyle m^*} \\ G & \underset{l^*}{\longleftarrow} & D & \underset{r^*}{\longrightarrow} & H \end{array}$$

Elements in L_q which are not in the range of l are said to be deleted by q, while elements in R_q which are not in the range of r are said to be created by q. Given a graph grammar $\mathscr{G} = \langle T, P, \pi, G_0 \rangle$, a *derivation* $\rho : G_0 \stackrel{p_1,m_1}{\Rightarrow} G_1 \stackrel{p_2,m_2}{\Rightarrow} G_2 \cdots$ of $\mathscr{G}$ is a finite or infinite list of direct derivations $\delta_i : G_i \stackrel{p_i,m_i}{\Rightarrow} H_i$, where $G_{i+1} = H_i$ and $i \geq 0$. If a derivation $\rho : G_0 \stackrel{p_1,m_1}{\Rightarrow} \cdots \stackrel{p_n,m_n}{\Rightarrow} G_n$ is finite we call G_n the final graph.

Example 1 (Graph grammar) *Figure 2 shows a graph grammar thats models a client-server scenario. The type graph* `T` *represents the possible kinds of nodes: clients (stylized persons), content servers (cylinders), addresses (pentagons), data (rectangles with sharp angles), signaling messages (rectangles with rounded angles), and connections between clients and servers (circles with the letter* `C`*). There are basically two kinds of interactions in this system: clients can recover information from servers providing an address as parameter, and clients can store information in servers, passing both the address and the desired information as parameters* [1]*. In order to retrieve or store information, the client must first connect with a server that provides the required address. After the connection, the information is exchanged and, finally, the connection is released. The graph productions* `ConnectGet`*,* `TransferGet` *and* `CloseGet` *perform the information retrieval from servers, and the graph productions* `ConnectSend`*,* `TransferSend` *and* `CloseSend` *perform the information update. Inside the rules, the items annotated with small* `D`*'s are the ones being deleted, and the ones with small* `C`*'s are the ones being created. The initial graph of the system consists of two clients and three servers. One of the clients comes with an initial send message for address* `A2` *(the "updater" client), while the other one has two get messages for addresses* `A2` *and* `A3` *(the "reader" client). According to the order in which the productions are applied, the reader client can retrieve information about the address* `A2` *before or after it is updated by the updater client. Also, the reader client can get connected to any server that provides address* `A3`*, retrieving different results according to the server it connects to.*

[1] in this example, it would be necessary to have attributes in order to properly represent addresses and numeric data. Since our main focus is in the crosscutting concerns, for now we left attributes out of the theoretical development.

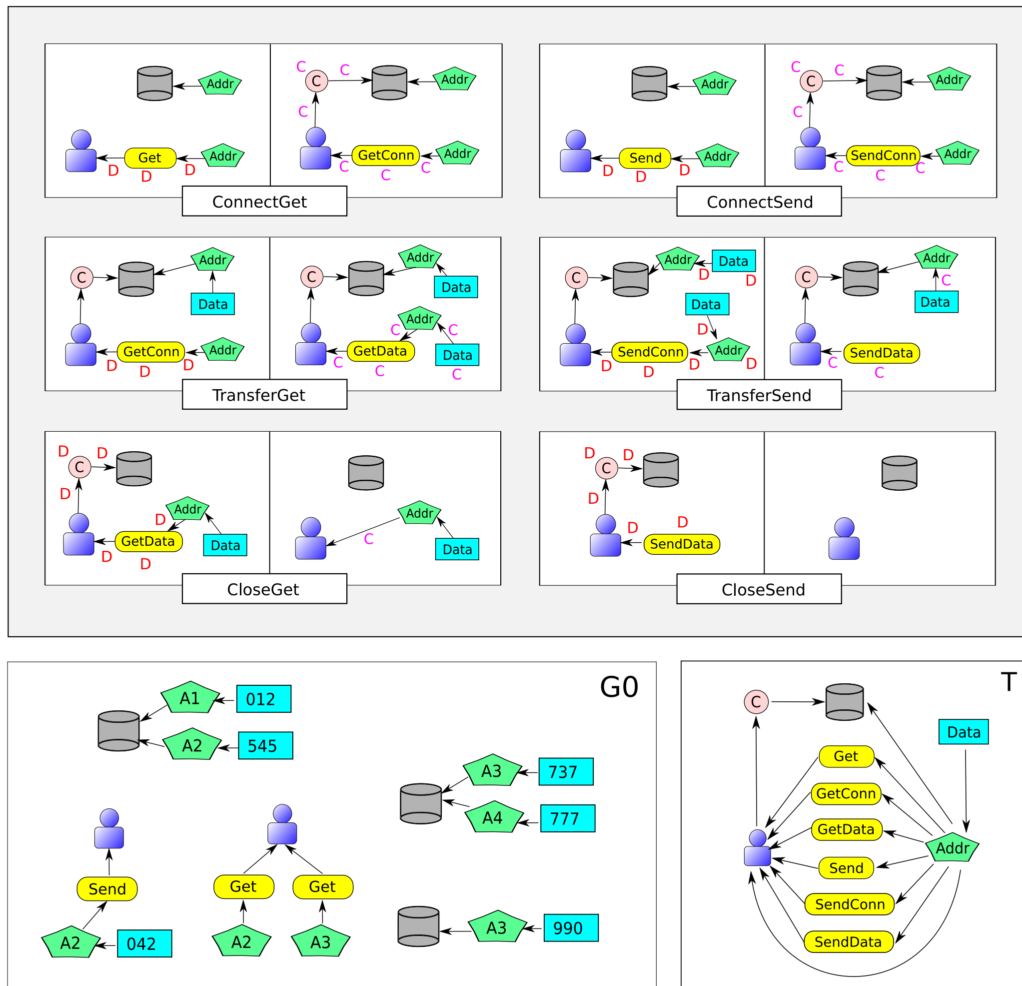


Figure 2: Example of graph grammar for clients and servers

Graph grammars provide a natural and visual way to represent distributed and nondeterministic systems, such as the one shown in Example 1. Distribution is naturally represented by the graph topology. The semantics of graph grammars is based on production applications. If there are matches for more than one production in one state (graph), they may all be applied in parallel, if there are no conflicts. Conflicts exist if two (or more) productions try to delete the same portion of a graph at the same time. In such situation, the choice of which production will be actually applied is non-deterministic.

4 Crosscutting Concerns in Graph Grammars

The main purpose of the aspect-oriented paradigm is to solve the problem of lack of modularity for the code that handles crosscutting concerns. In order to illustrate crosscutting concerns in the

context of graph grammars, we propose two simple modifications to the system of Figure 2: the inclusion of a *logging* object (to log executions) and of a security policy for server access.

4.1 Logging Execution Steps

Suppose we want to register every execution step within the system in order to have access to the execution history. For instance, it is very common that servers store information about the start and the end of each client connection, both for profiling and security reasons. In the context of GGs, this would mean that we have to record each production application, or derivation step. In our example, the changes that have to be performed to introduce such a log object are:

1. the type graph would have to be extended to introduce the new kind(s) of element(s);
2. the initial graph should be populated with initial instances of the new elements (if any). In the case of log, we must introduce one global instance of the log object;
3. all relevant productions must be modified in order to reflect the desired behavior. The left-hand side of every rule should have an additional element (the log register), and some information related to the effect of a production application on this log shall be included.

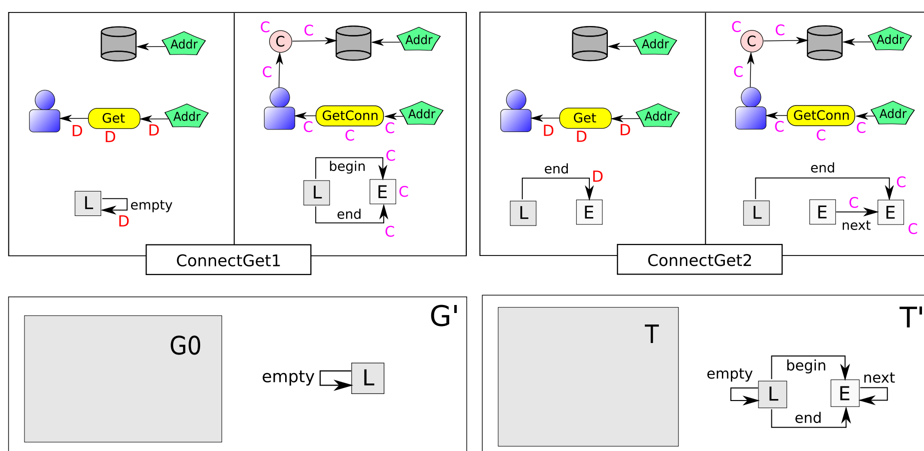


Figure 3: New type graph, initial graph and variations of the original rule `ConnectGet` to implement log.

Figure 3 depicts the required modifications over the GG presented in Example 1 in order to implement a simple log policy. The square node with an `L` represents the global log object. The square node with an `E` represents a log entry, which carries information about the production application. In order to keep the example simple, we omitted this information from log entries – they actually only represent the number of applied productions (this abstraction is fine, since our purpose is not to show how to model logs, but rather how to model transformation of specifications, that is, how one specification is transformed into another by considering an aspect). Log entries are connected to each other in a way that resembles a linked list structure, represented by

the arrows begin, end and next. The empty list is represented by the endoarrow empty. The modification to the initial graph would be only the addition of one empty log object, i.e. one with a unique empty arrow. The greatest impact comes from the modifications in productions, since all of them must be altered to cope with two different situations: when the log list is empty, and when it has at least one element. For instance, the rule ConnectGet must be rewritten as a pair of productions, as shown in Figure 3. This should be done for every production of the original specification, duplicating the total number of productions of the graph grammar. This very small example shows how structural patterns for productions may not scale well in the usual definition of graph grammars.

One interesting effect of this log model concerns the graph grammar execution. Since we have a global log object which is updated by all productions, we lose the possibility of simultaneous application of productions, even if they refer to different client and servers. Thus, this implementation of logging modifies the concurrent semantics of the system, although the sequential semantics is not changed at all.

4.2 Security policy for server access

Another system requirement that is a crosscutting concern is the implementation of *security policies*. Suppose it is important to distinguish between two kinds of users: *content administrators*, the ones that have write and read access to the servers, and ii) *plain users*, who can only read information. Every time a user tries to connnect to a server, its type should be taken into account to decide if the connection should be allowed. A very simple implementation of such policy is depicted in Figure 4, which shows a new type graph and new versions for rules ConnectGet and ConnectSend. The user attribute R represents *read* privilege and W, *write* privilege. Both user marks are preserved by the productions. Unlike the log policy, which affected all the productions, these are the only rules affected by the security policy, since the permissions may be verified only when the connections are being made.

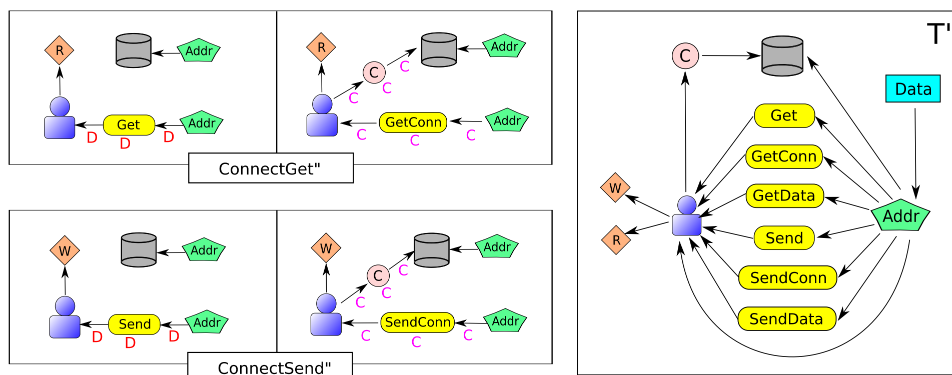


Figure 4: Modified rules and type graph for security policy.

Notice that both the log and the security implementations are modelled as modifications of both the structure (type graph) and the behavior (initial graph and graph productions) of the original GG. If both crosscutting concerns are needed in our specification, the productions may

become excessively complex and difficult to understand, since they may have to treat several crosscutting concerns. One of the original motivations for using visual methods such as GG is its ease of use, and such lack of modularity can difficult its adoption for modeling large systems. In the next sections we introduce aspect-oriented graph grammars (AOGG) as an extension of traditional graph grammars. In AOGGs, the modifications needed to treat every crosscutting concern are encapsulated into an *aspect*, allowing clearer specifications.

5 Aspect-Oriented Graph Grammars

In this section, we describe formally how to define aspects over graph grammars, leading to the definition of aspect-oriented graph grammars (AOGG).

Graph advices may be seen as meta-productions defining how the original graph productions should be modified in order to implement a given crosscutting concern. Therefore, we employ the same mechanism for graph rewriting in order to describe production rewritings. First, we define a notion of how to relate productions (production morphism), that will be used to formally define graph advices.

Definition 4 (Production morphism) Let $p : L_p \overset{l_p}{\leftarrowtail} K_p \overset{r_p}{\rightarrowtail} R_p$ and $q : L_q \overset{l_q}{\leftarrowtail} K_q \overset{r_q}{\rightarrowtail} R_q$ be T-typed graph productions. A *production morphism* $f : p \to q$ is a triple $\langle f_L, f_K, f_R \rangle$ of T-typed graph morphisms between the left-hand side, interface and right-hand side of the productions such that the diagram below commutes. The production morphism $f = \langle f_L, f_K, f_R \rangle$ is *injective* iff all its components are injective. The category of T-typed productions and T-typed production morphisms is denoted T-**MSpan**.

$$\begin{array}{ccccc} L_p & \overset{l_p}{\leftarrowtail} & K_p & \overset{r_p}{\rightarrowtail} & R_p \\ \downarrow f_L & & \downarrow f_K & & \downarrow f_R \\ L_q & \underset{l_q}{\leftarrowtail} & K_q & \underset{r_q}{\rightarrowtail} & R_q \end{array}$$

Definition 5 (Graph advice) A T-typed *graph advice* a is a production over T-typed productions, i.e. it is a monic span $p \leftarrowtail i \rightarrowtail e$ in T-**MSpan**. In terms of T-typed graphs, a graph advice has the structure depicted below, where all squares commute:

$$\begin{array}{ccccc} & & L_i \leftarrowtail K_i \rightarrowtail R_i & & \\ & \swarrow & & \searrow & \\ L_p \leftarrowtail K_p \rightarrowtail R_p & & & & L_e \leftarrowtail K_e \rightarrowtail R_e \end{array}$$

Given a T-typed advice $a : p \leftarrowtail i \rightarrowtail e$, the production p is called the *advice pointcut*, i, the *advice interface*, and e, the *advice effect*.

Definition 6 (Graph aspect) Given a graph grammar $\mathscr{G} = \langle T, P, \pi, G_0 \rangle$, we define a *graph aspect* A over $\mathscr{G}$ as a triple $\langle D, t, g \rangle$, where D is a set of T'-typed *graph advices* (see Definition 5), and $t : T \hookrightarrow T'$ and $g : G_0 \hookrightarrow G'_0$ are graph inclusions. The graphs T' and G'_0 are called, respectivelly, the *type graph* and *initial graph* of the aspect A.

Example 2 (Graph aspects) *Figure 5 depicts a graph aspect for the graph grammar of Figure 2, implementing an execution log. The regions* `T` *and* `G0` *refer to the original type graph and initial graph, respectively. The advices* `a1` *and* `a2` *implement the modifications over the original productions as presented in Section 4. The fact that the pointcut is empty makes them match all the original productions, as will be shown. Figure 6 shows a graph aspect implementing a security policy that affects only the* `ConnectGet` *and* `ConnectSend` *productions, since there are matches for the advice pointcuts in those productions.*

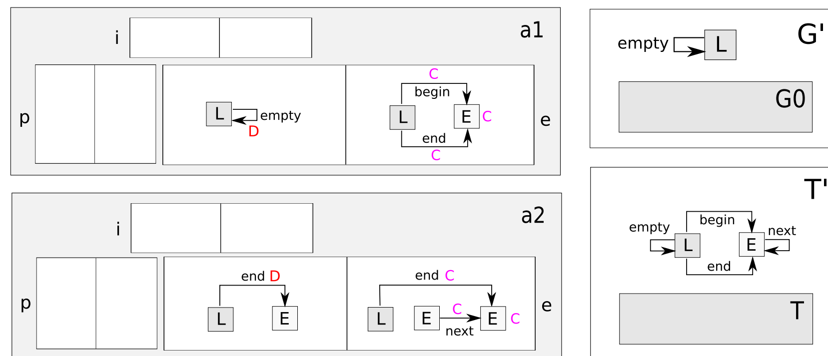


Figure 5: Example of a graph aspect implementing execution log.

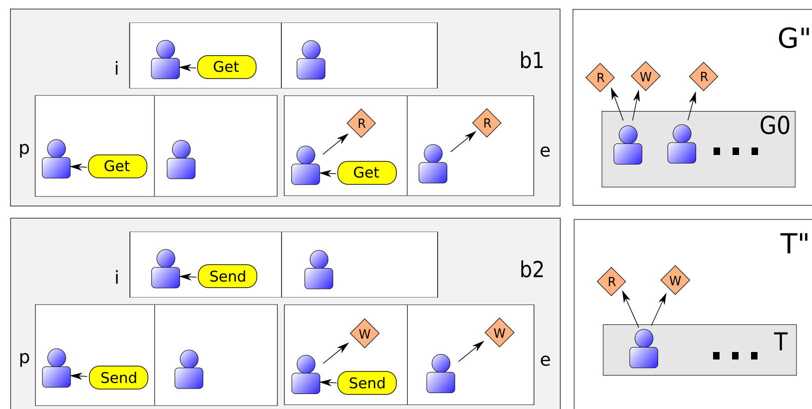


Figure 6: Example of a graph aspect implementing security policy.

Definition 7 (Aspect-oriented graph grammar) An aspect-oriented graph grammar (AOGG) is a pair $\mathscr{A} = \langle \mathscr{G}, \Delta \rangle$, where $\mathscr{G}$ is a graph grammar, and Δ is a (possibly empty) finite sequence $[A_1, A_2, \ldots, A_n]$ of graph aspects over $\mathscr{G}$.

The behavior of an AOGG $\mathscr{A} = \langle \mathscr{G}, \Delta \rangle$ is given by its *weaved graph grammar*, i.e. the graph grammar resulting from the combination of all aspects in Δ over $\mathscr{G}$. We start by defining how a single advice modifies one production (advice weaving), then how an aspect is weaved to a graph grammar (aspect weaving), and finally how one obtains the weaved graph grammar from a given aspect-oriented graph grammar (AOGG weaving).

Definition 8 (Advice weaving) Given a T-typed graph production q, a T-typed graph advice $a : p \leftarrowtail i \rightarrowtail e$ and a production monomorphism $m : p \rightarrowtail q$ (called a *production match*), an *advice weaving* from q to q' using a (based on m) exists if and only if the diagram below can be constructed, where both squares are pushouts in T-**MSpan**. In this case the advice weaving is denoted by $q \overset{a,m}{\Rightarrow} q'$.

$$\begin{array}{ccccc} p & \xleftarrow{l} & i & \xrightarrow{r} & e \\ {\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow \\ q & \longleftarrow & d & \longrightarrow & q' \end{array}$$

An advice can rewrite a production if there is an inclusion of its pointcut in the production. Then, the resulting production is obtained by a double-pushout construction applied componentwise in their left- and right-hand sides and in its interface. Intuitively, the elements that are in the pointcut production (for each graph component L, K and R) but not in the effect production are deleted, and those that are in the effect but not in the pointcut are created.

Definition 9 (Aspect weaving) Let $\mathscr{G} = \langle T, P, \pi, G_0 \rangle$ be a graph grammar, and $A = \langle D, T \overset{t}{\hookrightarrow} T', G_0 \overset{g}{\hookrightarrow} G' \rangle$ and aspect over $\mathscr{G}$. Then the *aspect weaving* of A over $\mathscr{G}$, denoted by $W_{\text{ASP}}(\mathscr{G}, A)$, is a graph grammar $\mathscr{G}' = \langle T', P', \pi', G' \rangle$, where P' and π' are calculated as follows:

1. all T-typed productions $x \in range(\pi)$ are *retyped* for T' by composing their respective typing morphisms with the inclusion t. This generates the set $Q^{T'}$ of T'-typed productions:

$$\begin{array}{ccccc} L_x & \longleftarrow & K_x & \longrightarrow & R_x \\ & {\scriptstyle t_{L_x}}\searrow & \downarrow{\scriptstyle t_{K_x}} & \swarrow{\scriptstyle t_{R_x}} & \\ & & T & \overset{t}{\hookrightarrow} & T' \end{array}$$

with t'_{L_x}, t'_{K_x}, t'_{R_x} mapping L_x, K_x, R_x to T'.

2. the set Q' is defined as the smallest set which the following holds: for all $y \in Q^{T'}$,

   (a) if does not exist an advice $a \in D$ and a match m such that $y \overset{a,m}{\Rightarrow} y'$, then $y \in Q'$

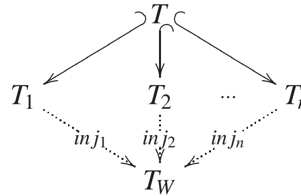
   (b) if $y \overset{a,m}{\Rightarrow} y'$ for some a and m, then $y' \in Q'$

3. The set P' of rule names and the function $\pi' : P' \to Q'$ are chosen arbitrarily, respecting the restriction that π' must be a bijection.

The application of an aspect weaving in a AOGG generates a GG consisting of the type and initial graph of the aspect and all productions obtained by applying all advices based on all possible matches over all productions of the AOGG. The productions that are not updated by any advice are kept in the resulting GG.

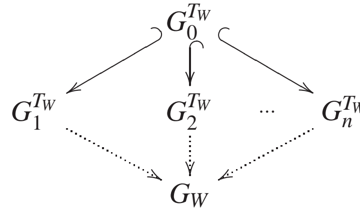
Definition 10 (AOGG weaving) Let $\mathscr{A} = \langle \mathscr{G}, \Delta \rangle$ be an AOGG, such that $\mathscr{G} = \langle T, P, \pi, G_0 \rangle$ and $\Delta = [\langle D_i, T \overset{t_i}{\hookrightarrow} T_i, G_0 \overset{g_i}{\hookrightarrow} G_i \rangle \mid 1 \leq i \leq n]$. The weaved graph grammar $\mathscr{G}_W^\Delta$ is calculated as follows:

1. the type graph T_W of $\mathscr{G}_W^\Delta$ is the object of the colimit (in **Graph**) of the diagrams containing all type graph inclusions in Δ, as shown in the diagram below.



2. in order to relate the initial graphs and productions in all aspects, we need to retype the original graph grammar $\mathscr{G}$ and all aspects in Δ by composing all their typing morphisms with the respective injections over T_W. This generates the retyped AOGG $\langle\langle T_W, P^{T_W}, \pi^{T_W}, G_0^{T_W} \rangle, \Delta^{T_W} \rangle$, where all type graph inclusions $t_i^{T_W} : T_W \hookrightarrow T_W$ (in all aspects) become identities.

3. the initial graph G_W of $\mathscr{G}_W^\Delta$ is the object of the colimit (in T_W-**Graph**) of the diagram containing all T_W-retyped initial graph inclusions in Δ^{T_W}, as shown in the diagram below.



4. the graph grammar $\mathscr{G}_W^\Delta$ is obtained as the result of $W_{\text{AOGG}}(\langle T_W, P^{T_W}, \pi^{T_W}, G_W \rangle, \Delta^{T_W})$. The operation W_{AOGG} is defined inductively, combing all aspects in Δ^{T_W} according to the order they appear.

$$W_{\text{AOGG}}(\mathscr{G}', []) = \mathscr{G}'$$
$$W_{\text{AOGG}}(\mathscr{G}', [A_1, A_2, \ldots A_n]) = W_{\text{AOGG}}(W_{\text{ASP}}(\mathscr{G}', A_1), [A_2, \ldots, A_n])$$

Finally, the AOGG weaving is done by applying all aspects in order of occurrence: the first aspect is applied over the original grammar and the subsequent ones are applied over the grammar resulting of the previous aspect weaving.

The AOGG weaving model has the following characteristics: *i) positive pointcut match:* the pointcut matching is given by a single production monomorphism; *ii) non-reentrant weaving:*

our weaving model combines one advice and one rule at most once for every possible match; *iii) deterministic aspect combination:* by using a sequence instead of a set, we enforce a canonical ordering for the aspects in a AOGG. Although these properties allow us to easily express the aspects for our example, they also may not be the most expressive choices. In aspect-oriented languages, usually there is a rather complex *expression language* for defining pointcuts, which also includes negative expressions such as "all methods whose return type is *not* void". Without negative matches, we can not differ created elements from preserved elements in the pointcut, since we can not test their absence from the interface of the production. It would also be of interest to define how pointcuts should be composed in our graph-based setting. Concerning the non-reentrant weaving, this brings both advantages and drawbacks. The positive effect is that non-deleting advices (the ones where the left-hand side is isomorphic to the interface) pose no problem to the weaving, since they will never start a non-terminating rewriting. On the other hand, it may not suffice to describe more complex aspects.

6 Concluding Remarks

One of the first connections between graph rewriting systems and aspect-oriented programming was made in [AL99], where graph rewriting mechanism was proposed has a tool for describing aspects over graph based models. Some proposals, such as the MATA framework [WJ07], follow this principle, characterizing aspect weaving as a special kind of model transformation. Most of these works do not extend the theory of graph rewriting for aspects, employing it as a language for specifying diagram transformations. As far as we know, there is no other formal approach for defining aspects and aspect weaving over graph grammars.

In this work, we addressed the problem of the lack of modularity for crosscutting concerns in graph-grammars, and claimed that aspects for graph grammars are an interesting approach for the modularisation of such requirements. We provided a formal definition for aspects over graph grammars, leading to the definition of aspect-oriented graph grammars (AOGG). We also defined the aspect weaving process that combines all the aspects over the base grammar in an AOGG, resulting in a (weaved) graph grammar. We showed by means of an example how the use of AOGGs may reduce the size of GG-based specifications that must deal with crosscutting concerns.

Our approach differs from others that relate aspects and graph rewriting systems mainly because it propose the definition of aspects over graph grammars, and not graph grammars as rewriting models for aspects. On the technical side, there is still room for improvements on the proposed theory, such as extending the pointcut matching model and defining composition operations for pointcuts and advices. It would be interesting to confirm that this theory holds for other kinds of graph rewriting models, such as attributed graph grammars or even in the single-pushout approach. The way AOGG is defined models static weaving of systems. Thus, it remains an open question if there is a way to define dynamic modifications in rules during the system execution. Other topics of investigation include the study of aspect interference over the execution of the base grammar and the possible conflicts between aspects.

Acknowledgements: The authors would like to thank the anonymous referees for their helpful

comments and suggestions. This work was partially supported by CNPq – Conselho Nacional de Desenvolvimento Científico e Tecnológico.

Bibliography

[AL99] U. Aßmann, A. Ludwig. Aspect Weaving with Graph Rewriting. In Czarnecki and Eisenecker (eds.), *GCSE*. LNCS 1799, pp. 24–36. Springer, 1999.

[C+97] A. Corradini et al. Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 1, chapter 3, pp. 163–245. World Scientific, River Edge, February 1997.

[CL06] C. Clifton, G. T. Leavens. MiniMAO$_1$: an imperative core language for studying aspect-oriented reasonings. *Sci. Comput. Program.* 63(3):321–374, 2006.

[DDFB06] S. D. Djoko, R. Douence, P. Fradet, D. L. Botlan. CASB: Common Aspect Semantics Base,. Technical report, Research Report, Network of Excellence in AOSD (AOSD-Europe, August 2006, no D54)., 2006.

[HET08] F. Hermann, H. Ehrig, G. Taentzer. A Typed Attributed Graph Grammar with Inheritance for the Abstract Syntax of UML Class and Sequence Diagrams. *ENTCS* 211:261–269, 2008.

[JJR06] R. Jagadeesan, A. Jeffrey, J. Riely. Typed parametric polymorphism for aspects. *Sci. Comput. Program.* 63(3):267–296, 2006.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, J. Irwin. Aspect-oriented programming. In *Eropean Conference on Object-Oriented Programming, ECOOP'97*. LNCS 1241, pp. 220–242. Springer, Finland, June 1997.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 1. World Scientific, River Edge, February 1997.

[WJ07] J. Whittle, P. K. Jayaraman. MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation. In Giese (ed.), *MoDELS Workshops*. LNCS 5002, pp. 16–27. Springer, 2007.

[WZL03] D. Walker, S. Zdancewic, J. Ligatti. A Theory of Aspects. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*. Pp. 127–139. ACM Press, New York, NY, USA, 2003.