Characterizing Conflicts Between Rule Application and Rule Evolution in Graph Transformation Systems

Rodrigo Machado[1,2](✉), Leila Ribeiro[1,2], and Reiko Heckel[1,2]

[1] Universidade Federal Do Rio Grande Do Sul (UFRGS), Porto Alegre, Brazil
{rma,leila}@inf.ufrgs.br
[2] University of Leicester, Leicester, UK
reiko@mcs.le.ac.uk

Abstract. Systems and models usually evolve with time, triggering the question of how the introduced modifications impact their original behavior. For rule-based models such as graph transformation systems, model evolution may be represented by means of a collection of structural modifications in individual transformation rules. In this work we introduce the notion of inter-level conflict between rule modification and rule application, characterizing the situations where the evolution disables a transition of the original system. We discuss the confluence of the evolution with respect to individual rewritings, and we also propose how the notion of inter-level conflict can be used to help the modeler to foresee the effects of model evolution.

1 Introduction

Computational systems are always evolving. Evolution may be due to correction of errors, optimization, introduction of new features, adaptation to new technologies, languages or platforms, among others. Typically, when one version of a system is delivered, the developers are already working on further versions to come. In such a scenario, it is fundamental to understand how those changes impact the system's original behavior. If we restrict evolution to traceable structural modifications in the description of the system behavior (for instance, rewriting rules) it may be possible to relate these changes with the overall system execution (i.e. the application of those components over the system state), or, at least, be warned of potential implications of some modifications.

Many systems can be modeled by an initial condition and a set of transformation rules. Graph transformation systems (GTS) [2], for instance, are essentially a set of typed graph rewriting rules. The behavior of a GTS is given by iterated application of rules over an initial graph. Due to the simplicity of the concept of graph rewriting, and the availability of modeling and analysis tools such as AGG [11] and Groove [9], GTSs have been used to describe several kinds of model transformations for visual languages (such as the ones from the UML family).

F. Parisi-Presicce and B. Westfechtel (Eds.): ICGT 2015, LNCS 9151, pp. 171–186, 2015.
DOI: 10.1007/978-3-319-21145-9_11

In this work we investigate how structural modifications in GTSs may affect their respective behavior. For instance, augmenting the left-hand side of a given rule has the effect of disabling its application over graphs that do not contain the new requirements. Although this is quite obvious, we have found that deleting some parts of the left-hand side may as well disable some of its rewritings. This is not as obvious as adding a new requirement, and justifies the importance of a method to help the modeler to foresee all situations where changes in rules impact their respective rewritings.

For our discussion we will employ (typed) GTSs under the double-pushout approach (DPO) for graph rewriting [2]. We introduce the notions of *rule evolution* that characterize changes in individual rules, and *inter-level conflicts* that characterize the interference of evolution over a particular graph transformation. We also propose an extension to the *critical pair analysis* technique, whose purpose is to calculate, for a given GTS submitted to evolution, all possible inter-level conflicts. The aim of finding critical pairs is to detect situations in which evolution may not succeed as expected and generate a warning for the modeler regarding the adequacy of the evolution. This kind of static analysis technique is very useful during the modeling stage to avoid the introduction of undesirable behavior.

This paper is organized as follows: in Sect. 2, we present a review of graph transformation systems under the double-pushout approach and introduce our working example. In Sect. 3, we motivate an evolution of the example system and we introduce a formal definition for evolutions. In Sect. 4, we define the notion of inter-level conflict (and inter-level independence) between rule evolution and rule rewriting, presenting some examples of conflicting situations. Questions of confluence between inter-level independent evolutions and rewritings are discussed in Sect. 5, where we prove that we can obtain confluence under specific conditions. In Sect. 6, we review the critical pair analysis algorithm, and propose an extension to capture inter-level conflicts. In Sect. 7, we compare our approach to related work. We conclude in Sect. 8 discussing application scenarios and pointing towards future work.

2 Background

This section reviews the fundamentals of GTSs and present our working example. First, we recall some basic definitions regarding graphs and graph rewriting rules.

A (directed) graph is a tuple $G = (V, E, s, t)$ where V is a set of nodes, E is a set of edges, and s, t are functions that map each edge to its respective source and target node. In the following we refer as graph elements both nodes and edges of a given graph. A graph homomorphism $f : (V_1, E_1, s_1, t_1) \to (V_2, E_2, s_2, t_2)$ is a pair of total functions (f_V, f_E) where $f_V : V_1 \to V_2$, $f_E : E_1 \to E_2$, and, for all $e \in E_1$, we have $f_V \circ s_1(e) = s_2 \circ f_E(e)$ and $f_V \circ t_1(e) = t_2 \circ f_E(e)$. A typed graph $tg : G \to T$ is a graph homomorphism where the elements of T (the type graph) represent types of nodes and edges, and the elements of G (the instance graph) have a type assignment given by the homomorphism mapping. For example, the nodes in graph T (shown in Fig. 2) describe four kinds of nodes: messages (envelops), clients (laptops), servers (tower-style CPUs), and data nodes (sheets of papers). There are five kinds of edges, representing the location of messages over servers and clients, and data over messages, clients and servers.

A morphism between two typed graphs $tg_1 : G_1 \to T$ and $tg_2 : G_2 \to T$ is a graph homomorphism $f : G_1 \to G_2$ between the instance graphs G_1 and G_2 such that $tg_2 \circ f = tg_1$. In the following, we assume all graphs and morphisms are typed over a global typed graph T, and hence, for the sake of brevity, we omit the T-*typed* qualification for rules, matches and rewritings.

Under the double-pushout approach to graph transformation, a graph rule is a span $p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ (a pair of typed graph morphisms l and r with the same source) where L, K and R are graphs and both l and r have *injective* mappings. Within p, the left-hand side graph L represents a pattern to be found in order to apply the rule, the interface graph K represents the elements which are maintained by the rule application. The right-hand side graph R presents new nodes and edges to be added by the rule rewriting. An element of L which does not have a pre-image in K along l is said to be *deleted by* p and an element of R which does not have a pre-image in K is said to be *created by* p.

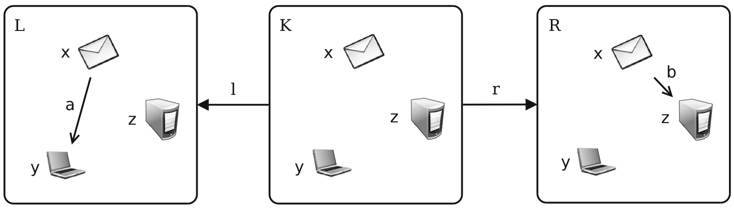

Fig. 1. Graph transformation rule.

Example 1. Figure 1 shows a graph rule which deletes the edge a, creates the edge b and preserves the three nodes x, y and z (both l and r are inclusions). Since edges are used to specify the location of messages, this production represents the act of sending a message from a computer to a server.

A match for rule $p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ over a graph G is simply an homomorphism $m : L \to G$. The effect of modifying a graph G into a graph G' by means of a graph rule p and match m is a graph rewriting, which we denote $G \overset{p,m}{\Longrightarrow} G'$. Informally, the rewriting consists of deleting the image along m of the elements deleted by p, which results in an intermediate graph D. Then, we add to D the elements created by p. In the literature of graph transformation, this double step may be compactly described as the existence of a *double-pushout diagram* in the category of typed graphs involving G, m, p and G', as shown below (for more details, see [2]):

$$\begin{tikzcd}
R \arrow[ddd] & & & R' \arrow[lll] \arrow[rrr] & & & R'' \arrow[ddd] & & \\
& K \arrow[ul, "r"] \arrow[dr, "l"'] \arrow[ddd] & & & K' \arrow[lll] \arrow[rrr] \arrow[ul] \arrow[dr] & & & K'' \arrow[ul, "r''"'] \arrow[dr, "l''"] \arrow[ddd] & \\
& & L \arrow[ddd, "m_0"] & & & L' \arrow[lll, "f_L"] \arrow[rrr, "g_L"] & & & L'' \arrow[ddd, "m_0''"] \\
H \arrow[rrrrrr, dash, "\cong"] & & & & & & H'' & & \\
& D \arrow[ul] \arrow[dr] \arrow[rrrrrr, dash, "\cong"] & & & & & & D'' \arrow[ul] \arrow[dr] & \\
& & G \arrow[rrrrrr, dash, "="] & & & & & & G
\end{tikzcd}$$

The double-pushout approach to graph rewriting imposes two conditions over the match, which have to be satisfied in order for the rewriting to occur: *(i)* an element deleted by the rule may not be identified in the match with any other element of the graph (identification condition); *(ii)* a node may not be deleted if there are incident arrows over it which are outside of the match (dangling condition). Whenever these two conditions (called gluing conditions) are satisfied for a given match, the rewriting is possible.

A graph transformation system is a tuple $\mathcal{G} = (T, P, \pi)$ where T is a type graph, P is a set of rule names, and π is a function that associates to each rule name a particular T-typed graph transformation rule. In the following, whenever we refer to a rewriting of p, we mean actually a rewriting of rule $\pi(p)$.

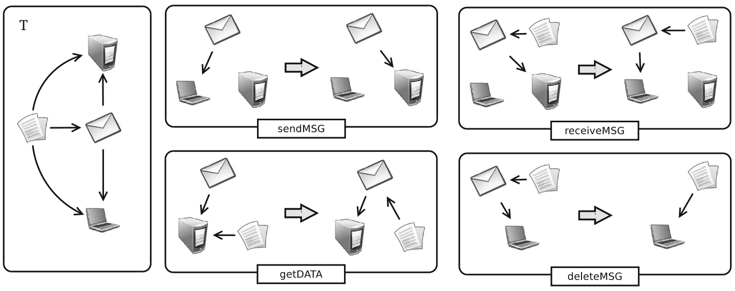

Fig. 2. Graph transformation system for clients and servers.

Example 2. Figure 2 shows a graph transformation system thats models a client-server scenario. There are four kinds of transitions in this system: clients sending a message to servers (sendMSG), obtaining data elements from the server (getDATA), servers returning the messages to the clients (receiveMSG) and clients obtaining data from returned messages (deleteMSG). The visual depiction of each rule omits the interface graph, which we implicitly take to be the intersection of the left-hand side and right-hand side graphs.

Given an initial graph G_0, a derivation of $\mathcal{G}$ from G_0 consists of a sequence of graph rewritings $G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} G_2 \overset{p_3,m_3}{\Longrightarrow} \dots$ where $p_i \in P$ for all $i \in \mathbb{N}$.

Example 3. Figure 3 presents a derivation of the shown graph transformation system over an initial situation consisting of a single client and two servers.

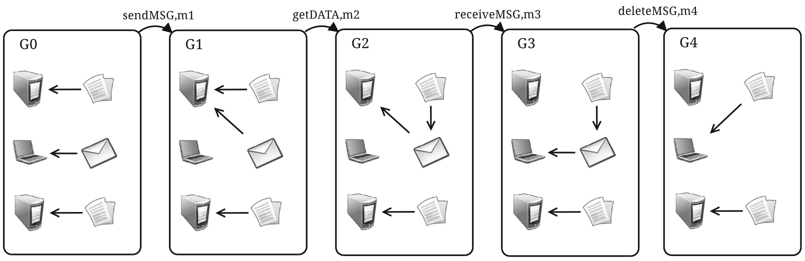

Fig. 3. Graph derivation.

For a given graph G, it may be possible to have several possible rewritings $G \overset{p,m}{\Longrightarrow} G'$ of distinct rules or even the same rule in distinct parts of the graph. Operationally, the simplest solution to this is to consider a *non-deterministic* choice of which rule and match to apply. If there are two possible rewritings $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ from the same graph G, we say that they are in conflict iff one of the rewritings disables the subsequent application of the other in the same part of the graph (usually by deleting something that the other rewritings needed). When two rewritings are not conflicting there are said to be parallel independent.

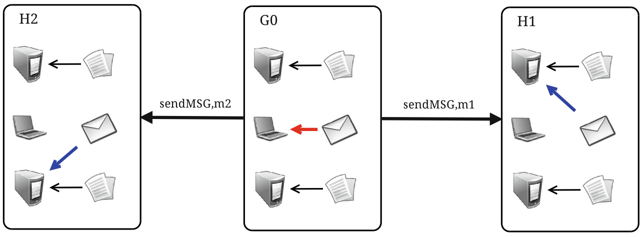

Fig. 4. Conflicting graph derivations.

Example 4. Figure 4 presents a conflict between two distinct application of the rule sendMSG. Each application deletes the arrow from the message to the client and creates a new arrow from the message to a server. Since the application of one removes the arrow needed by the other, they cannot be both executed, and therefore they are conflicting.

3 Evolution of Graph Transformation Systems

We now consider the case of how to represent the evolution of a graph transformation system. Changes in systems and models may occur due to very distinct reasons, such as the correction of errors, addition of new features or simply structural reorganizations (refactorings). Either way, a generic way of framing the evolution of a given model is to consider that some of its elements have been removed, added or preserved. Notice that GTSs have only two components: one which defines structural restrictions (the type graph) and one which defines the system execution (the set of graph rewriting rules), and we need to specify in which way those elements may be modified.

Before dealing with the formal definitions, let us introduce a simple example of model evolution. Although very straightforward, our example graph transformation system of Fig. 2 has some behaviors that could be considered defects in comparison with the original modeler's intention. It is not uncommon during the modeling stage to obtain an incorrect approximation of the intended behavior, and to successively refine the specification until it faithfully encodes the original concept. The next example highlights the problems with the original model.

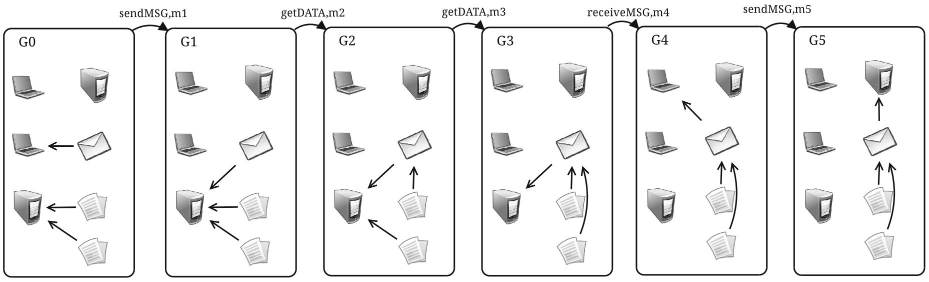

Fig. 5. Graph derivation exposing problems in the model.

Example 5. Figure 5 presents a derivation of the example GTS over an initial graph consisting of two clients and two servers. There are at least three potential issues:

- The first issue can be seen in the rewriting $G2 \xRightarrow{\mathsf{getData},m3} G3$, where a second data node is loaded over the only message. Although this may not seem a problem at first sight, it completely disables the subsequent application of rule deleteMSG over the message. The reason is that the deletion of the message node would leave a dangling edge from the data node that is not transferred to the client. In the double-pushout approach, dangling edges prevent the rewriting (although in other approaches, such as the single-pushout approach [5], the rewriting would occur and the dangling edge would be deleted, leaving the data node astray).
- The second issue is shown by rewriting $G3 \xRightarrow{\mathsf{receiveMSG},m4} G4$, since the message we have sent from a particular client has returned to a different one. This is clearly the result of not storing a reference to the original sender, which enables the receiveMSG rule to return the message to any of the available clients.

– The third issue may be perceived by the fact that even when a given message returns from a server to a client, nothing prevents it to be re-sent to another server instead of being deleted and have its content delivered. This is shown in $G4 \xRightarrow{\mathsf{sendMSG},m5} G5$, where a received message is re-sent (although in this case it would not be possible to delete the message due to the first issue).

In order to *correct* these issues, the modeler may consider the following modifications:

– The creation of a new kind of edge from messages to clients, in order to mark the original sender, and thus solving the second issue;
– The use of a token over messages which is removed when data is loaded. If we assume that each message starts with at most one token, we can prevent the loading of multiple data. Moreover, if we modify the rule sendMSG to require messages to have tokens, we can ensure that only new messages are sent to servers. The implementation of those tokens may be as simple as adding a self-edge over the message.

The evolved GTS that incorporates these modifications is shown in Fig. 6. Even if the presented evolution may seem artificial (since it would not be that hard to build the correct model from the start), we claim this example illustrates the nature of the modifications that also occur in more complex scenarios. Notice also that both original and the evolved model assume some structural properties of the graph that will be transformed by them, such that messages start with a unique token, and that messages and data cannot be located at two or more places simultaneously. In this particular case, all changes were in the sense of adding new kinds of edges to the type graph, and adding edges to the left-hand side, interface and right-hand side of rules but, in general, we can also expect some deprecated elements of the original specification to be deleted. The following definition formalizes what we mean by evolution.

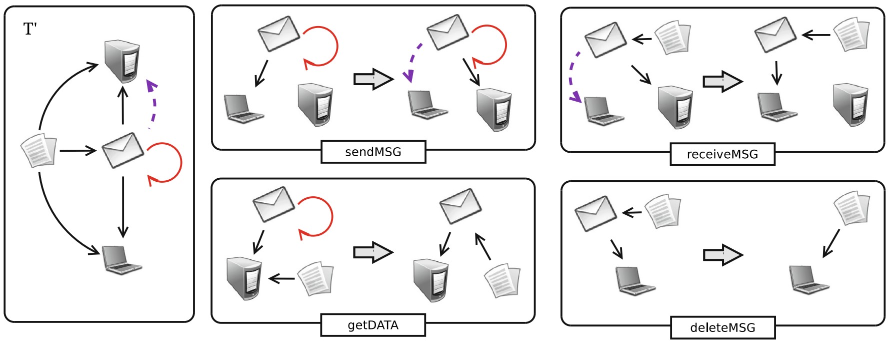

Fig. 6. Evolved graph transformation system for clients and servers.

Definition 1 (Evolution of Graph Transformation System). *Given two GTSs $\mathcal{G}_1 = (T_1, P, \pi_1)$ and $\mathcal{G}_2 = (T_2, P, \pi_2)$ with the same set of rule names P, we define an* evolution *between them as a pair (E_T, E_P) where*

- *E_T is an injective span $T_1 \leftarrowtail T_K \rightarrowtail T_2$ representing a modification in the type graph;*
- *E_P is a function mapping each rule name $p \in P$ to a commutative diagram (in the category of graphs) with the format shown in Fig. 7, named* evolutionary span *of p, where the left rule is $\pi_1(p) = L_1 \leftarrow K_1 \rightarrow R_1$ (T_1-typed), the central rule is $p_K = L_K \leftarrow K_K \rightarrow R_K$ (T_K-typed), the right rule is $\pi_2(p) = L_2 \leftarrow K_2 \rightarrow R_2$ (T_2-typed), and all morphisms in the top surface are monomorphisms. By an abuse of language, we will denote the evolutionary span $E_P(p) = \pi_1(p) \leftarrowtail p_K \rightarrow \pi_2(p)$ as an injective span which is T_{1+2} typed, where T_{1+2} is the object of the pushout of $E_T = T_1 \leftarrow T_K \rightarrow T_2$.*

Notice that this definition assumes that we have a fixed set of rule names which is kept constant across the evolution (i.e. we do not add or remove new rules).

$$\begin{tikzcd}
 & & R_1 \arrow[dddl] & & R_K \arrow[ll, tail] \arrow[rr, tail] \arrow[dddl] & & R_2 \arrow[dddl] \\
 & K_1 \arrow[ur, tail, "r1"] \arrow[dl, tail, "l1"'] \arrow[dd] & & K_K \arrow[ll, tail] \arrow[rr, tail] \arrow[ur, tail] \arrow[dl, tail] \arrow[dd] & & K_2 \arrow[ur, tail, "r2"'] \arrow[dl, tail, "l2"] \arrow[dd] & \\
L_1 \arrow[dr] & & L_K \arrow[ll, tail] \arrow[rr, tail] \arrow[dr] & & L_2 \arrow[dr] & & \\
 & T_1 \arrow[drr] & & T_K \arrow[ll, tail] \arrow[rr, tail] & & T_2 \arrow[dll] & \\
 & & & T_{1+2} \arrow[u] & & &
\end{tikzcd}$$

Fig. 7. Evolutionary span as a diagram in the category of graphs.

4 Inter-level Conflicts

In this section, we consider the possible interaction of an evolution over the potential rewritings of a rule, and introduce the notion of *inter-level conflict*.

When we compare two possible rewritings over the same graph, we say that they are in conflict when the execution of one disables the execution of the other in the resulting graph. Now, instead of comparing two possible rewritings from the same graph, we intend to compare an arbitrary graph rewriting $G \xRightarrow{p,m_0} H$ of a given rule $p = L \leftarrow K \rightarrow R$ with an arbitrary rule span $p \leftarrow p' \rightarrow p''$ denoting the rule evolution. This situation can be represented by the diagram shown in Fig. 8 (in the category of T_{1+2}-typed graphs).

We consider that there is independence (or non-interaction) between the rewriting and the evolution when the evolution *does not* disable the rewriting. In order for this to occur, we need to be able to rewrite the graph G with rule p'' over the same place as the original rewriting (i.e. over an equivalent match m_0'' of p'' over G). This is formalized by our notion of *inter-level independence*.

Definition 2 (Inter-level Independence and Conflict). *Let $\rho = G \overset{p,m_0}{\Longrightarrow} H$ be a graph rewriting where $p = L \leftarrow K \rightarrow R$ and let $\theta = p \leftarrow p' \rightarrow p''$ be a evolutionary span of rule p. We say that ρ and θ are (inter-level) independent iff there is a match $m_0'' : L'' \rightarrow G$ (as shown in Fig. 8) such that*

1. $m_0 \circ f_L = m_0'' \circ g_L$

2. m_0'' satisfies double-pushout gluing conditions for $p'' = L'' \xleftarrow{l''} K'' \xrightarrow{r''} R''$

We define ρ and θ to be in (inter-level) conflict iff they are not independent.

$$\begin{tikzcd}
R \arrow[ddd] & & R' \arrow[ll] \arrow[rr] & & R'' & & \\
& K \arrow[ul, "r"] \arrow[dr, "l"'] \arrow[ddd] & & K' \arrow[ll] \arrow[rr] \arrow[ul] \arrow[dr] & & K'' \arrow[ul, "r''"'] \arrow[dr, "l''"] & \\
& & L \arrow[ddd, "m_0"] & & L' \arrow[ll, "f_L"'] \arrow[rr, "g_L"] & & L'' \arrow[dddllll, bend left, "m_0''"] \\
H & & & {} \arrow[rr, Rightarrow, "\text{evolution}"'] & & {} & \\
& D \arrow[ul] \arrow[dr] & & & & & \\
& {} \arrow[uul, Rightarrow, "\text{rewriting}"] & G & & & &
\end{tikzcd}$$

Fig. 8. Graph rewriting and rule evolution.

In other words, we have a conflict between a rule application and a rule evolution whenever the evolved rule does not have any match for G (that is compatible with the original match m_0) or when all compatible matches violate some gluing condition of the evolved rule.

Example 6 (Inter-level Conflict). Figure 9 depicts four situations that cause inter-level conflicts. We do not show the intermediate rule (of evolution) and graph (of rewriting) to help visualization, and we consider them to be the intersection between the shown components.

(a) This situation reflects the obvious case when we are increasing the requirements (left-hand side) of a rule, and thus the transformation cannot be applied over graphs that do not have the new requirements. This particular situations shows the evolution of rule sendMSG, and the conflict arises because we enforce that rules must contain a self-edge in order to be sent.

(b) The rule depicted in this case does not occur in our example GTS, but allows us to illustrate another kind of inter-level conflict arising when we consider non-injective matches. The original rule matches against two messages, creating a self-edge over each of them. We have a valid DPO rewriting by matching both messages in the left-hand side over the single message of the graph, and applying the transformation. If, however, we change the rule as shown in the evolution, forcing the rule to delete one of the messages, the rewriting would not be possible. This happens because the evolved rule would be trying to simultaneously delete and preserve the same message, which violates the identification condition.

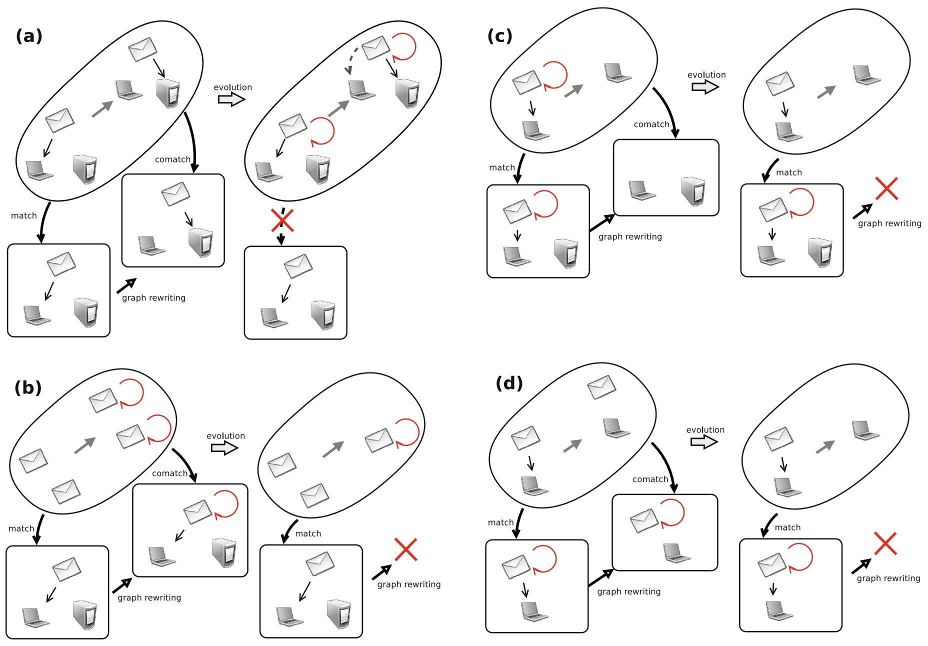

Fig. 9. Situations causing inter-level conflicts between evolution and rewriting.

(c) This situation is not as obvious as the first two, since it shows that conflicts may arise even when decreasing the requirements for a rule application. The shown rule deletes a message with self-edge located in a client. Hence, the rule is applicable over the depicted graph, modifying it as expected. However, if the evolution modifies the rule in such a way that it does not delete the self-edge, the same rewriting becomes impossible because it would leave a dangling edge in the resulting graph.

(d) This case shows that evolution may create a conflict by changing the preservation of a node into deletion. Since the message node is preserved, the original rule can be applied over messages that have incident edges. However, if we change the rule in a way that it deletes the message, these rewritings are not possible anymore due to the violation of the dangling condition.

In the double-pushout approach, dependencies and conflicts are dual to each other in the sense that two rewritings $\delta_1 : G \Rightarrow G_1$ and $\delta_2 : G \Rightarrow G_2$ are conflicting iff there is a dependency between $\delta_1^{-1} : G_1 \Rightarrow G$ and δ_2. In the same way, inter-level conflicts are said to be caused by evolutions that disable some graph rewriting, we can define that a graph rewriting depends on an evolution if it was enabled by it, i.e., when the particular rule modification makes it possible. For instance, consider the reverse of the situation (c) depicted in Fig. 9 (reading the evolution from right-to-left, adding the self-edge to the LHS instead of removing it). Clearly, this modification in the rule turns a match that would violate the dangling condition over G into a valid one, allowing the respective first-order rewriting to occur in the modified rule. This reasoning suggests that the adequate notion of inter-level dependency may be seen as a conflict between the *inverse* of the evolution and the graph rewriting, confirming that the symmetry we observe between conflicts and dependencies in traditional DPO graph rewritings extends toward the inter-level scenario.

5 Inter-level Confluence

The notions of conflict and independence are usually related to the notion of *confluence*. In particular, DPO rewriting satisfies *local confluence* (also known as local Church-Rosser) which states that independent rewritings can be applied in any order (or even in parallel), resulting in the same graph. Formally, if $G \xRightarrow{p_1,m_1} H_1$ and $G \xRightarrow{p_2,m_2} H_2$ are not conflicting, then there are rewritings $H_1 \xRightarrow{p_2',m_2'} H$ and $H_2 \xRightarrow{p_1',m_1'} H$ which result in the same graph H.

Our notion of inter-level conflict focus on applicability of a rule which may change in a way that is not necessarily conservative of old behavior. In this sense, *confluence* would mean that the final result of a rewriting would be the same independent of the evolution having ocurred or not. This is not expected in general, as the next example shows.

Example 7 (Inter-level Independence without Confluence). Case (a) in Fig. 10 depicts a non-confluent, inter-level independent scenario. The original rule deletes a message over a client, and the evolution has the effect of adding another element to be deleted (a server). In this case, the resulting graphs of the rewritings of evolved and original rules are clearly not isomorphic.

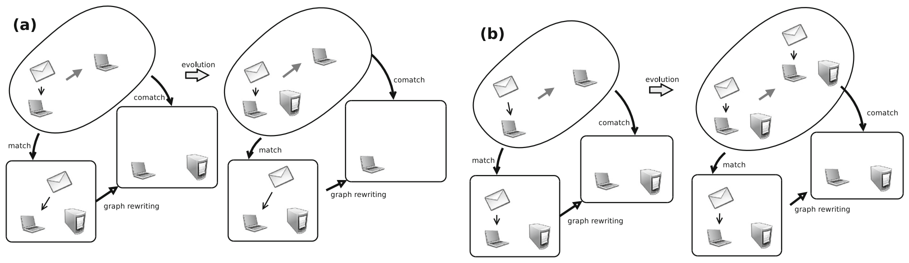

Fig. 10. Inter-level independent evolution and rewriting with non-confluence (a) and confluence (b).

There are some situations where the evolution adds or removes only itens which are *preserved* by the rewriting, without increasing or decreasing its deleted and created elements.

Example 8 (Inter-level Independence with Confluence). In the scenario (b) depicted in Fig. 10, the evolution forces the rule to delete the message *in the presence* of a server, which is maintained intact by the rule. Inter-level independence ensures that the new rule can be applied, i.e. the graph has at least one server to provide a match for the new rule. Since both rules delete and create the same amount of elements, we have confluence.

These conservative evolutions can be identified by the fact that, when seeing them as a diagram in the category of T_{1+2}-typed graphs, all squares in the evolutionary span are pushouts. For these evolutions, we are guaranteed to have confluence, as we demonstrate next.

Lemma 1 (Inter-level Confluence). *Let $\rho = G \xRightarrow{p,m_0} H$ be a graph rewriting where $p = L \leftarrow K \rightarrow R$ and let $\theta = p \leftarrow p' \rightarrow p''$ be a evolutionary span of rule p such that they are inter-level independent. Let us call $\rho'' = G \xRightarrow{p'',m_0''} H''$ the rewriting of the modified rule $p'' = L'' \leftarrow K'' \rightarrow R''$.*

If all squares in θ are pushouts in the category of T_{1+2}-typed graph, then the evolution and graph rewriting are confluent (the graphs H and H'' are isomorphic).

Proof. Consider the following depiction of the situation above as a diagram in the category of T_{1+2}-typed graphs (all squares in the top and the sides are pushouts).

$$\begin{tikzcd}
R \arrow[ddd] & & & R' \arrow[lll] \arrow[rrr] & & & R'' \arrow[ddd] & & \\
& K \arrow[ul, "r"'] \arrow[dr, "l"] \arrow[ddd] & & & K' \arrow[ul] \arrow[lll] \arrow[rrr] \arrow[dr] & & & K'' \arrow[ul, "r''"'] \arrow[dr, "l''"] \arrow[ddd] & \\
& & L \arrow[ddd, "m_0"] & & & L' \arrow[lll, "f_L"'] \arrow[rrr, "g_L"] & & & L'' \arrow[ddd, "m_0''"] \\
H & & & & & & H'' \arrow[llllll, "\cong"'] & & \\
& D \arrow[ul] \arrow[dr] \arrow[rrrrrr, dash, "\cong"] & & & & & & D'' \arrow[ul] \arrow[dr] & \\
& & G \arrow[rrrrrr, "="] & & & & & & G
\end{tikzcd}$$

1. by the property of pushout composition between the pushouts on the top and the sides, we obtain double-pushout diagrams denoting the rewritings $G \xRightarrow{p',m_0 \circ f_L} H$ and $G \xRightarrow{p',m_0'' \circ g_L} H''$;
2. due to inter-level independence, we know that $m_0' = m_0 \circ f_L = m_0'' \circ g_L$;
3. because pushout complements are unique (up to isomorphism) in adhesive categories such as typed graphs (Theorem 4.26 in [2]) we have $D \cong D''$;
4. because pushouts are unique (up to isomorphism) in general categories we have $H \cong H''$. □

6 Inter-level Critical Pair Analysis

Critical pair analysis is a static analysis technique which shows, for a given GTS, all possible conflicts (or dependencies) between rule applications. Usually, this technique is available at modeling and analysis tools (such as AGG) where it presents valuable information to the modeler regarding potential behavior of the system.

Consider a GTS $\mathcal{G} = (T, P, \pi)$. Roughly speaking, critical pair analysis consists of the following steps:

1. for each pair (p_1, p_2) where $p_1, p_2 \in P$, calculate all possible overlaps of their LHSs (for conflicts) or all possible overlaps of the RHS of one rule with the LHS of the other one (for dependencies);
2. for each overlap, verify if the corresponding rewritings are conflicting or not (respectively, dependent or not);
3. present a table $size(P) \times size(P)$ containing the number of conflicts or dependencies identified between each pair of rules.

As a rule-based model grows, it becomes increasingly hard for the modeler to identify all the possible interactions between the rewriting rules. In this way, the information provided by critical pair analysis allows the identification and correction of flaws at an earlier stage of the modeling process. For instance, if we take the original system shown in Fig. 2 as an example, the problem of resending loaded messages appears as a *dependency* between rules receiveMSG and sendMSG. Usually, the output table of the method is interactive, and allows for the modeler to see the conflicting or dependency situation visually.

We envision that our notion of inter-level conflict can be used in a similar way to aid the modeler to foresee the potential effects of a given model evolution. We propose a method for executing *inter-level critical pair analysis* as follows.

Definition 3 (Inter-level Critical Pair Analysis). *Given a graph transformation system $\mathcal{G}_1 = (T_1, P, \pi_1)$ and an evolution (E_T, E_P) of $\mathcal{G}_1$ into $\mathcal{G}_2 = (T_2, P, \pi_2)$, we proceed as follows:*

1. *for each rule name $p \in P$,*
   (a) *take its evolution $E_P(p) = q \leftarrow q' \rightarrow q''$;*
   (b) *generate a set $R(p)$ of relevant graphs for q (see Definition 5);*
   (c) *generate all pairs (q, m), where $m : \mathrm{LHS}(q) \rightarrow G$ is a match for some graph $G \in R(p)$ satisfying DPO gluing conditions;*
   (d) *for each pair (q, m), detect if the rewriting $G \overset{q,m}{\Longrightarrow} H$ and the evolution $E_P(p)$ are inter-level conflicting or not.*
2. *present a table $size(P) \times 1$ containing the number of inter-level conflicts for the evolution of each rule in P.*

One important part of this definition is the calculation of the relevant graphs $R(p)$, which need to include all possible scenarios that would lead to conflicts after the evolution. For instance, we need to account for *(i)* the lack of matches (absence of m''_0), *(ii)* the violation of identification conditions and *(iii)* the violation of dangling conditions. The information required to build graphs that may trigger *(i)* and *(ii)* is available in the LHSs of the rules q, q' and q''. The situation *(iii)*, however, requires that we take into consideration edges which may not occur in the LHSs of the rules (as shown in case (d) of Fig. 9). For this purpose, we define the dangling extension of the LHS of a rule, which is used in the calculation of the set of relevant graphs.

Definition 4 (Dangling Extension). *Let $q : L^T \xleftarrow{l} K^T \xrightarrow{r} R^T$ be a finite T-typed rule where $\tau : L \to T$ is the typing morphism of L. Let $delNodes(q)$ be the set of nodes of L which are deleted by q. Given a node n of L, let $S(n)$ be the set of all edges $e \in E(T)$ such that $source(e) = \tau(n)$ and, respectively, let $T(n)$ be the set of all edges $e \in E(T)$ such that $target(e) = \tau(n)$. Define L^+ as the graph obtained from L by creating, for each $n \in delNodes(q)$ and for each edge type $e \in S(n) \uplus T(n)$, a new e-typed edge instance. Each new instance connects the node n to a fresh node instance at the other end. We denote $L \hookrightarrow L^+$ the obvious inclusion of L into its dangling extension.*

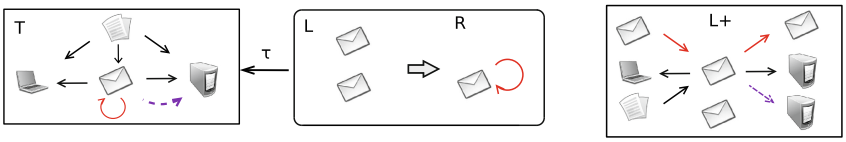

Fig. 11. Example of dangling extension.

Example 9 (Dangling Extension). Fig. 11 depicts the dangling extension of the LHS of a rule which deletes a message and creates a self-edge over another message.

Definition 5 (Relevant Graphs). *Given a graph transformation system $\mathcal{G}_1 = (T_1, P, \pi_1)$, an evolution (E_T, E_P) of $\mathcal{G}_1$ into $\mathcal{G}_2 = (T_2, P, \pi_2)$ and a rule name $p \in P$, we calculate the set of relevant graphs $R(p)$ as follows:*

1. *let $L \leftarrow L' \rightarrow L''$ be the span of LHSs of $E_P(p)$ (as shown in Fig. 8).*
2. *let G be the object of the colimit of $L^+ \hookleftarrow L \leftarrow L' \rightarrow L'' \hookrightarrow (L'')^+$*
3. *define $R(p)$ to be the set of all partitions of all subgraphs of G.*

The presented definition for relevant graphs is conservative in the sense that it does not focus on efficiency but rather on ensuring that every possible conflicting situation is captured. However, implementations of inter-level critical pair analysis should focus on creating the smallest subset of $R(p)$ containing all inter-level conflicts. As a very simple (and obvious) example of application of inter-level critical pairs, consider that the evolution of rule sendMSG shown in Fig. 6 essentially adds new elements to the rule structure, requiring the rule to preserve a self-edge over messages. This creates an inter-level critical pair, shown in the item (a) of Fig. 9, where the rule is not applicable. This information would be available to the modeler as soon as the evolution is specified, and, in this particular case, would alert for the need of preparing the initial state with self-edges in messages.

7 Related Work

Many approaches [1,4,8] represent model evolution by means of rewritings in components of rules, generally introducing a notion of compatibility (preservation of behavior) between the original and evolved systems. In this paper we take the evolution as an information obtained externally, either manually or via some other mechanism (possibly rewriting), and the aim is only to characterize the effect of evolution over the applicability of rules. Notice that the preservation of behavior is not assumed and we only present a (rather straightforward) sufficient condition for it. On the other hand, we can employ inter-level critical pair analysis in all situations where it is possible to obtain an evolutionary span for rules.

The problem of extending the evolution from meta-models (e.g. type graph) to models (e.g. typed graphs and typed rules) is considered in [12]. This is in contrast with our approach, where the relationship between the evolution of the type graph and the evolution of typed graph rules is encoded statically in the definition of evolution.

In terms of structure, evolutionary spans are similar to *triple graph rules* [10].

8 Concluding Remarks

In this work we have addressed the issue of relating structural modifications in rules (of GTSs) and their respective rewritings in order to detect potential conflicts. We introduced a way to represent the *evolution* of a GTS, defined a notion of *inter-level conflicts* and discussed how they can be used in *inter-level critical pair analysis*. Although the main contribution of this paper is conceptual, we foresee practical applications of the introduced concepts in the implementation of evolution assistants in tools such as AGG or Groove. Notice also that the proposed notion of inter-level conflict is applicable whenever we can characterize the rewriting as a double-pushout diagram, and evolution as a span of rules. For instance, the same definition could be generalized towards Adhesive HLR Systems [3], since those generalize DPO graph transformation. Important instances of this framework include algebraic specifications, Petri nets, typed attributed graph transformation system, among others.

One aspect that could be questioned in our treatment is the fact that the notion of evolution does not include addition or removal of rules. It would be possible to describe deletion (resp. creation) of rules as an evolution from (resp. to) the empty rule if we allowed extra unassigned rule names in both original and evolved GTS. The empty rule is always applicable, and does not have any conflict or dependency with other rules. For more on this, we refer the reader to [6]. Regarding future work, we consider the implementation of inter-level critical pair analysis in a graph transformation tool, the further development of the presented theory (for instance, considering rules with negative application conditions) and the application of these concepts to study the behavior of *second-order graph grammars* [6,7].

References

1. Ehrig, H., Ehrig, K., Ermel, C.: Refactoring of model transformations. Electron Commun. EASST **18** (2009). http://dblp.uni-trier.de/rec/bib/journals/eceasst/EhrigEE09
2. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science, An EATCS Series. Springer, Berlin (2005)
3. Ehrig, H., Habel, A., Padberg, J., Prange, U.: Adhesive High-Level Replacement Categories and Systems. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 144–160. Springer, Heidelberg (2004)
4. Ermel, C., Ehrig, H.: Behavior-preserving simulation-to-animation model and rule transformations. Electr. Notes Theor. Comput. Sci. **213**(1), 55–74 (2008)
5. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theoret. Comput. Sci. **109**(1–2), 181–224 (1993)
6. Machado, R.: Higher-order graph rewriting systems. Ph.D. thesis, Instituto de Informatica - Universidade Federal do Rio Grande do Sul (2012). http://hdl.handle.net/10183/54887
7. Machado, R., Ribeiro, L., Heckel, R.: Rule-based transformation of graph rewriting rules: towards higher-order graph grammars. Theoretical Computer Science (2015, to appear)
8. Parisi-Presicce, F.: Transformations of graph grammars. In: Graph Gramars and Their Application to Computer Science, 5th International Workshop, Williamsburg, VA, USA, Selected Papers, pp. 428–442, 13–18 November 1994
9. Rensink, A.: The GROOVE simulator: a tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
10. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) Graph Transformations. Lecture Notes in Computer Science, vol. 5214, pp. 411–425. Springer, Berlin (2008)
11. Taentzer, G.: AGG: a tool environment for algebraic graph transformation. In: Münch, M., Nagl, M. (eds.) AGTIVE 1999. LNCS, vol. 1779, pp. 481–488. Springer, Heidelberg (2000)
12. Taentzer, G., Mantz, F., Lamo, Y.: Co-transformation of graphs and type graphs with application to model co-evolution. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 326–340. Springer, Heidelberg (2012)