UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JONAS SANTOS BEZERRA

Occurrence Graph Grammars with Negative Application Conditions

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof^{a}. Leila Ribeiro

Porto Alegre
June 2019

CIP — CATALOGING-IN-PUBLICATION

Santos Bezerra, Jonas

Occurrence Graph Grammars with Negative Application Conditions / Jonas Santos Bezerra. – Porto Alegre: PPGC da UFRGS, 2019.

104 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2019. Advisor: Leila Ribeiro.

1. Graph Grammars. 2. Occurrence Graph Grammars. 3. Negative Aplication Conditions. 4. Semantics. I. Ribeiro, Leila. II. Título.

“Poste, kiam ŝi pripensis la aferon,

ŝi opiniis ke tio ja estis mirinda,

sed kiam ĝi okazis ĉio ŝajnis tute natura.”

— Alico en Mirlando

ACKNOWLEDGEMENT

I'm very happy with everything I've accomplished so far and although I still have a long way to go, I must thank some amazing people in my life without whom this path would be a lot harder, if not impossible.

First of all, I would like to thank Prof. Leila Ribeiro for giving me the opportunity to work with her even without previously knowing me: You are such an inspiring advisor and working with you always made me feel like I can improve myself and push me to the next level. Working with you was, at the same time, very challenging and joyful. I also need to thank professors Rodrigo Machado, Érika Cota and Lúcio Duarte, for always being there for conversations and feedback about this work, you were really a great help.

I owe a very special thank you to Calebe and Marília, for helping me settle in Porto Alegre when I knew nothing, nowhere and nobody here. You guys really made my life a lot easier.

To Andrei, Guilherme, Leonardo and Ana, my colleagues and friends, it was both an honour and a pleasure to work and live with you. I will always cherish our memories together as some of my favourites. To all the guys from lab 202, Diego, Fabi, Marcelo, Marina, Marlo, Felipe Tanus, Michele, Felipe Grando, Jéssica e Pedro: we need to go out more. To my very first friends in Porto Alegre, Shauna and Maurício, thank you for all the good moments.

To my old friends, Wendell, Gabriel, Malu and Jéssica, who were always by my side despite the 3178km of distance, your constant contact and kindness helped me through a lot of critical moments. I love you guys.

Ao meu irmão, Mateus, com quem eu sei que sempre posso contar e por quem eu sempre tentei dar o melhor de mim, de tal forma que ele tenha tanto orgulho de quem eu sou e da família que formamos: obrigado por me aceitar como eu sou.

Finalmente, para mainha, Dona Necila... eu jamais poderei expressar suficientemente o quão a senhora foi importante em todas as minhas conquistas até agora. A senhora sempre foi a pessoa que mais admiro na vida, por toda sua luta, dedicação e sacrifícios para dar a mim e aos meus irmãos todas as oportunidades que a senhora não teve. Esse mestrado é tão seu quanto meu. Vou te amar pra sempre.

ABSTRACT

Graph Grammars are based on the application of rules that are able to modify graphs, as such, they provide a suitable formalism to model complex systems in an intuitive and precise manner, providing both a graphical language and a solid formal background for systems analysis. Therefore, they have been used in a wide range of applications within Computer Science, specially in the field of Model-Driven Development. Particularly, the study of the Semantics of Graph Grammars, i.e. which graphs belong to the language of a grammar and which derivations are possible within the context of a grammar, provides a powerful framework for reasoning about the execution behaviour of systems modelled as Graph Grammars. There are several different ways of specifying the Semantics of Graph Grammars. One notable possibility is the use of Occurrence Graph Grammars, which encodes the Semantics in a structure that is also a Graph Grammar itself. Occurrence Graph Grammars differ from other semantic models such as Unfolding and Canonical Derivations mainly by providing a more compact, easier to analyse structure. They were introduced in the nineties and used ever since, however the original definitions lack the inclusion of Negative Application Conditions, additional structures imposed over the rules of a grammar to better tune their possible applications according to the execution context. Given the important role Negative Application Conditions play in the modelling and analysis of complex systems as Graph Grammars nowadays, this thesis presents an extension of the framework of Occurrence Graph Grammars to include them. It also presents its implementation in Verigraph, a system specification and verification tool based on graph rewriting.

Keywords: Graph Grammars. Occurrence Graph Grammars. Negative Aplication Conditions. Semantics.

Gramáticas de Grafos de Ocorrência com Condições Negativas de Aplicação

RESUMO

Gramáticas de Grafos baseiam-se na aplicação de regras que modificam grafos, fornecendo assim um formalismo adequado para a modelagem de sistemas complexos de forma intuitiva e precisa, além de fornecer uma notação gráfica descomplicada e uma base formal sólida para a análise de sistemas. Dados tais atributos, essas gramáticas possuem uma ampla gama de aplicações dentro da Ciência da Computação, especialmente no campo do Desenvolvimento Orientado a Modelos. Particularmente, o estudo da semântica de Gramáticas de Grafos (isto é, quais grafos pertencem à linguagem da gramática e quais derivações são permitidas no contexto da gramática) provê uma poderosa ferramenta para compreender e analisar o comportamento de sistemas modelados como Gramáticas de Grafos. Existem diversas formas de especificar a semântica de Gramáticas de Grafos, uma delas é o uso de Gramáticas de Grafos de Ocorrência que codificam tal semântica em estruturas que também são, por sua vez, Gramáticas de Grafos. O uso de Gramáticas de Grafos de Ocorrência ao invés de outros modelos semânticos, como por exemplo *Unfolding* e Derivações Canônicas, possui a vantagem de fornecer uma estrutura mais compacta e fácil de analisar. Gramáticas de Ocorrência foram introduzidas nos anos noventa e utilizadas desde então, porém as definições originais não incluem o uso de Condições Negativas de Aplicação, estruturas adicionais anexadas às regras de uma gramática para refinar as possíveis aplicações das regras em determinados contextos. Dada a atual importância das Condições Negativas de Aplicação na modelagem de sistemas complexos, essa dissertação propõe uma extensão da teoria das Gramáticas de Grafos de Ocorrência de forma a incluí-las, além de apresentar a implementação desta teoria no Verigraph, uma ferramenta de especificação e verificação de sistemas baseada em reescrita de grafos.

Palavras-chave: Gramáticas de Grafos. Gramáticas de Grafos de Ocorrência. Condições Negativas de Aplicação. Semântica.

LIST OF ABBREVIATIONS AND ACRONYMS

DPO	Double Pushout
GG	Graph Grammar
GTS	Graph Transformation System
GUI	Graphical User Interface
NAC	Negative Application Condition
OGG	Occurrence Graph Grammar
PO	Pushout
SPO	Single Pushout

LIST OF FIGURES

CONTENTS

1 INTRODUCTION

Graph grammars are a suitable formalism to model complex systems in an intuitive and precise manner, providing both a graphical language and a solid formal background for systems analysis. In this framework, system states are modelled as graphs, while transitions between different states are modelled as graph transformation rules (EHRIG et al., 2006). A graph transformation rule generically has the form $L \overset{p}{\Rightarrow} R$, where there is at least one left side graph L containing a pattern to be found (a match) in order for the rule to be applied over an instance graph and a right side graph R corresponding to the effect of applying such a rule. Moreover, additional structures may be used to provide rules a better tune of which kinds of matches are acceptable to perform a transformation. For example: Graph Constraints, Negative Application Conditions (NACs) and Nested Application Conditions, to cite some of the most commonly used (HABEL; PENNEMANN, 2005).

There exist several approaches that might not only change the exact format of a rule, but also define different ways a rule can be applied over a given match. Only considering the domain of Category Theory, there is already a handful of different approaches for graph grammars: the Single Pushout (SPO) (EHRIG et al., 1997), Double Pushout (DPO) (CORRADINI et al., 1996), Sesqui-Pushout (SqPO) (CORRADINI et al., 2006), AGREE (CORRADINI et al., 2015), among others. Each approach has its own advantages/disadvantages regarding the kinds of "operations" they allow or forbid in the system under modelling. In spite of the approach used, there is a way to describe the behaviour of the system together with means to analyse several properties about this behaviour, such as termination, concurrency and reachable states. Among the several analysis techniques provided by graph grammars, we have:

- Critical Pair Analysis and Critical Sequence Analysis: critical pair analysis enables the verification of which rules conflict with (i.e. prohibit) the application of others and why; critical sequence analysis, which rules depend on the execution of others to be applied and why (LAMBERS; EHRIG; TAENTZER, 2008); such analyses provide insights about the possible execution flows of the system.

- Calculation of Concurrent Rules: a concurrent rule summarizes in one rule the combined results of applying several different rules. In other words, it represents the combination of several different rules which can be then applied as a "one step" transformation rule (LAMBERS et al., 2008; BEZERRA; RIBEIRO, 2016).

- State Space Exploration and Model Checking: permits to verify, in an exhaustive fashion, whether the (graph grammar) model of a system satisfies a given specification and to prove the satisfaction of its properties (RENSINK, 2004).
- Unfolding, Graph Processes, Occurrence Graph Grammars and Canonical Derivations: they are all different means to provide the semantics of Graph Grammars, allowing us to check which graphs belong to the language of a grammar and/or which concrete derivations are possible within that grammar (CORRADINI; MONTANARI; ROSSI, 1996; RIBEIRO, 1996).

Graph grammars have had a wide range of applications within Computer Science, specially in the field of model-driven software development, where the transformation of visual models is a vital part of the process, therefore a natural application of graph grammars (ROZENBERG, 1997). As evidence of such suitability, several non-trivial systems have been modelled and studied under the optics of graph grammars, such as telephone communications (RIBEIRO, 1996), elevator control (LAMBERS, 2010), railroad control (PENNEMANN, 2009) and integration of service-oriented systems (GIESE; VOGEL; WATZOLDT, 2015). Furthermore, there are a number of software tools to support the use of graph grammars, such as AGG, a tool environment for algebraic graph transformation (TAENTZER, 2000); Groove, a tool for state space generation (RENSINK, 2004) and Verigraph, a software specification and verification tool based on graph rewriting (BEZERRA et al., 2017).

Besides its powerful applications, the use of graph grammars as a framework for modelling systems provides us with a great advantage over other formalisms: it makes it possible for non-specialists in the field to generate graph grammar models of a system and then benefit from the rigorous analyses it offers without the need for a deep understanding of its underlying theory. For example, (JUNIOR et al., 2015; BEZERRA et al., 2016; COTA et al., 2017) explain how to generate graph grammars from a set of textual requirement documents such as use cases, functional specifications and other kinds of guidelines by means of a systematic methodology. They also present guidance towards using different graph grammars analysis techniques in order to improve and verify these documents and, consequently, the systems they describe.

The field of Graph Grammars is a very active one, and researches continuously develop new ideas, such as new graph transformation approaches (e.g. Sesqui-Pushout, AGREE), analysis techniques (e.g. Essential Critical Pairs), tools (e.g. Verigraph) and ways to apply them. Additionally, we may also benefit from the combination of already existing techniques, which is what we do in this thesis. In our work, we combine two concepts of the Graph Grammar Theory: Occurrence Graph Grammars, defined for the SPO and DPO approaches by (RIBEIRO, 1996; CORRADINI; MONTANARI; ROSSI, 1996), with Negative Application Conditions, defined generically by (HABEL; HECKEL; TAENTZER, 1996), which has not been done so far.

Occurrence Graph Grammars (OGGs) provide a semantics for Graph Grammars encoded in structures that are also Graph Grammars themselves. This semantics, which tells us which graphs are part of the grammar language and which graph transformations are possible within the context of the original grammar, may be used for the analysis of system execution in a summarized fashion and also in practical applications, such as test case generation, without the need of usage of a supplementary structure or formalism.

Negative Application Conditions (NACs) are extensions of a side of a rule encoding patterns that, if found in the match (or sometimes the comatch) of a rule, forbids the corresponding transformation. In theory, they do not increase graph grammars expressive power in relation to rules without them (HABEL; HECKEL; TAENTZER, 1996). In practice, they allow the modelling of systems in a much more concise and compact manner. Therefore they became really necessary when modelling complex, real-like systems (CORRADINI et al., 2013; CORRADINI; HECKEL, 2014).

Our work consists of the development of an extension for the framework of Occurrence Graph Grammars in the Double Pushout (DPO) approach in order to incorporate Negative Application Conditions, alongside with the implementation of this extension in the Verigraph system, a generic graph rewriting system based on Category Theory and written in Haskell. This implementation choice makes it possible for the source code of the tool to be close to the theory domain as well as allowing other researches to implement new approaches or different models of graphs while benefiting from the already implemented techniques (as long as they conform to the categorial constructions).

Thus, the main contributions of this thesis can be summarized as (1) the creation of an extension of the framework of Occurrence Graph Grammars (in the DPO approach) in order to include Negative Application Conditions and (2) the implementation of this extension in Verigraph, a software specification and verification system based on graph transformations, which is now also the first tool in the field to implement the construction of Occurrence Graph Grammars for general Graph Grammars, even when considering OGGs without NACs.

Structure of the Thesis:

Chapter 2: In this chapter we review the basic notions of graph transformation systems, specifically under the Double-Pushout (DPO) approach. We also introduce Negative Application Conditions (NACs) and the basic notions of parallel and sequential independence of rules, which are needed for the construction of Occurrence Graph Grammars with NACs.

Chapter 3: In this chapter we first present an overview of doubly-typed graph grammars and other concepts necessary to accomplish the construction of Occurrence Graph Grammars, as well as how these Occurrence Grammars can be used to represent the semantics of their original grammar. After reviewing these concepts, we present our extension to previous works in Occurrence Graph Grammars to include the notion of Negative Application Conditions, which is part of our thesis contribution.

Chapter 4: This chapter presents an overview of the Verigraph system, which was used to implement the techniques presented in this thesis. Verigraph in itself represents a novelty in the field of graph transformations, being the first tool in the area implemented in a functional language, which favoured its source code to be very close to the problem domain itself.

Chapter 5: This chapter explains in depth the step-by-step construction of an Occurrence Graph Grammar with the help of some running examples. Additionally, it demonstrates how this was implemented in Verigraph, while also providing some insight about how it can be used for test cases generation.

Chapter 6: This chapter discuss related work to the one presented in this thesis. Specifically focusing on the literature about the Semantics of Graph Grammars and how to use it for test cases generation. It also lists some software tools related to Verigraph.

Chapter 7: This chapter summarizes our results and presents our conclusions. Moreover, it shows remaining open problems and future work.

Appendix A: This appendix contains a brief review of category theory and the categorial constructions used in this thesis.

2 GRAPH GRAMMARS

The theory of graph grammars and graph transformation systems is based on the application of rules that are able to modify graphs. Using this framework, it is possible to model complex systems as graph transformation systems, where graphs represent system states, while rules model transitions between states.

Graph grammars have a wide range of application in computer science, not only because graphs are a natural and intuitive way of modelling complex situations, but also because it is possible to reason about several properties of the modelled systems using this formalism (EHRIG et al., 2006; ROZENBERG, 1997).

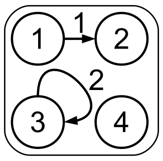
We now review the basic concepts and some analysis techniques for graph transformations that are used in this work. We use the algebraic approach, which is based on Category Theory. For the proofs of theorems and facts, the reader is referred to (CORRADINI et al., 1996), however a brief review of the main categorial concepts used in this thesis is presented in appendix A.

Definition 2.1 (Graph)**.** A graph is a tuple $G = (V, E, s, t)$ where: V is a set of nodes, E is a set of edges and $s, t : E \to V$ are two total functions that map each edge in E to its source and target in V.

□

Example 2.2 (Graph)**.** Figure 2.1 shows a simple graph G with $V = \{1, 2, 3, 4\}$, $E = \{1, 2\}$, $s = \{(1, 1), (2, 3)\}$ and $t = \{(1, 2), (2, 3)\}$. ■

Figure 2.1: A graph example



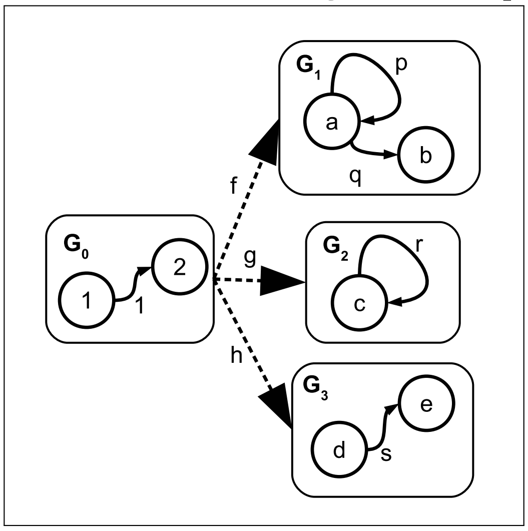
Definition 2.3 (Graph Morphism)**.** Given two graphs G_1, G_2 with $G_i = (V_i, E_i, s_i, t_i)$ for i in $[1, 2]$, a graph morphism $f : G_1 \to G_2$ between them is a pair $f = (f_V, f_E)$ where $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ are total functions that preserve the source and target functions, i.e. $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$. □

Example 2.4 (Graph Morphisms)**.** Figure 2.2a shows three morphisms $f : G_0 \to G_1$, $g : G_0 \to G_2$ and $h : G_0 \to G_3$, which are a monomorphism, an epimorphism and a isomorphism, respectively.

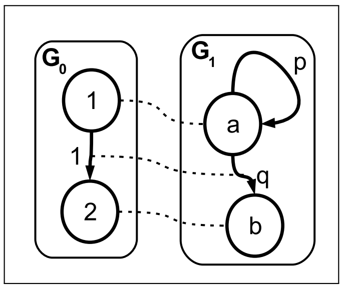
Morphism f maps node $\bigcirc_1$ to $\bigcirc_a$, node $\bigcirc_2$ to $\bigcirc_b$ and edge $\curvearrowleft_1$ to $\curvearrowleft_q$; g maps both nodes $\bigcirc_1$ and $\bigcirc_2$ to $\bigcirc_c$ and edge $\curvearrowleft_1$ to $\curvearrowleft_r$; and h maps node $\bigcirc_1$ to $\bigcirc_d$, node $\bigcirc_2$ to $\bigcirc_e$ and edge $\curvearrowleft_1$ to $\curvearrowleft_s$.

Figure 2.2b shows the morphism f in an expanded (explicit) notation. Both notations are used through this work.

Figure 2.2: Graph morphism examples



(a) Compact graph morphisms



(b) Expanded graph morphism

■

Definition 2.5 (Typed Graph and Typed Graph Morphism)**.** A type graph is a distinguished graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$ where V_{TG} and E_{TG} are called the node and edge type alphabets, respectively.

A typed graph is a pair $G^T = (G, type)$ consisting of a graph G and a graph morphism $type : G \longrightarrow TG$.

Given two typed graphs $G_1^T = (G_1, type_1)$ and $G_2^T = (G_2, type_2)$, a typed graph morphism $f : G_1^T \longrightarrow G_2^T$ is a graph morphism $f : G_1 \longrightarrow G_2$ such that $type_2 \circ f = type_1$:

$$\begin{tikzcd}
G_1 \arrow[rr, "f"] \arrow[dr, "type_1"'] & = & G_2 \arrow[dl, "type_2"] \\
& TG &
\end{tikzcd}$$

□

Example 2.6 (Typed Graph and Typed Graph Morphism Example)**.** Figure 2.3 shows a type graph T, and four graphs G_0, G_1, G_2, G_3 where only G_0 and G_1 are valid *T-typed* graphs. In this graphical representation, the shapes of the nodes represent different types. Thus, a square node can only be mapped to the square node in T, and so on.

Notice that G_2 can not be a *T-typed* graph because the type graph does not have a node of type $\Diamond$, neither an edge type with source and target in the $\Box$ type. Similarly, G_3 is not a valid *T-typed* graph because, although there is an edge type between a $\triangle$ and a $\Box$ types, the source of this type of edge must be a $\Box$ and the target a $\triangle$.

Figure 2.4 shows a typed graph morphism $f : G_0^T \rightarrow G_1^T$, where f maps node $\bigcirc_a$ to $\bigcirc_1$, $\Box_b$ to $\Box_1$ and edge $\curvearrowleft_e$ to $\curvearrowleft_2$.

Figure 2.3: *T-typed* valid and invalid graphs

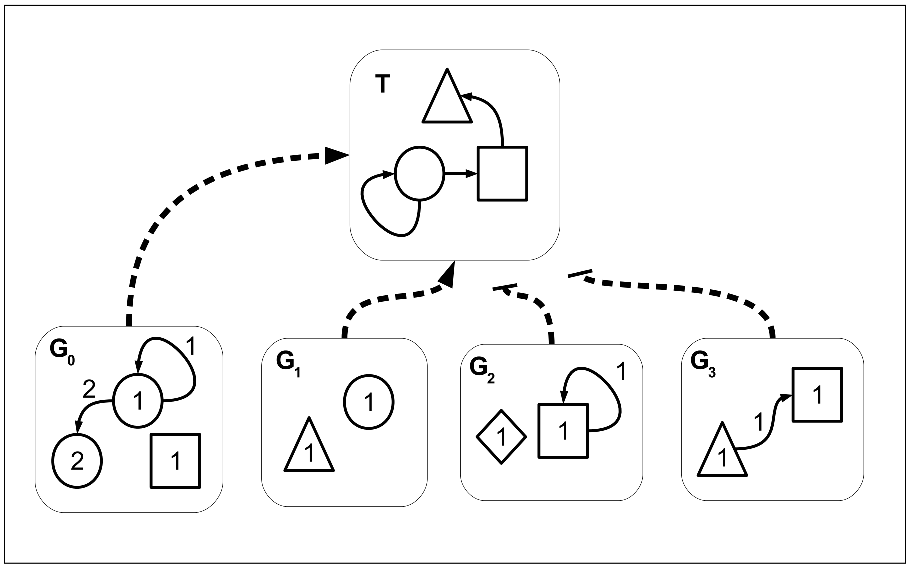
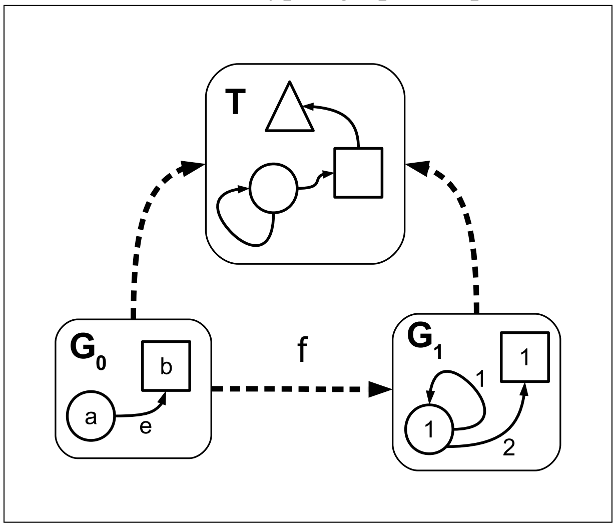


Figure 2.4: A typed graph morphism



■

Remark (Categories of Graphs and Typed Graphs)**.** We call **Graph** the category whose objects are graphs and arrows are graph morphisms. Similarly, we have that $\mathbf{TGraph_T}$ is the category whose objects are T−typed graphs and whose arrows are T−typed graph morphisms. □

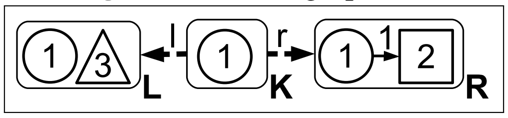
Definition 2.7 (Graph Rule)**.** A (typed) graph rule[1] $p = \left(L \xleftarrow{l} K \xrightarrow{r} R\right)$ is a span of (typed) graph monomorphisms $l : K \longrightarrow L$ and $r : K \longrightarrow R$ where the (typed) graphs L, K and R are called the left-hand side, gluing graph and right-hand side, respectively. The elements (nodes and edges) that appear in K are said to be *preserved* by the rule, while the elements in L, but not in K are *deleted* by it. Similarly, elements in R that do not appear in K are *created* by the rule.

Given a (typed) graph rule p, its inverse rule is defined by $p^{-1} = \left(R \xleftarrow{r} K \xrightarrow{l} L\right)$. □

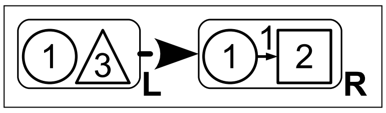
Example 2.8 (Graph Rule Example and Notation)**.** Figure 2.5 shows an example of a graph rule which reads a node of the type ○, deletes a node of the type △ and then creates a node of the type □ with an edge between the ○ and the □.

Figure 2.5a presents the rule in the standard DPO notation, while Figure 2.5b depicts the same rule in a compact notation, where the gluing graph is omitted. Sometimes we use the compact notation to make the figures smaller. Notice that the compact notation does not cause any semantic loss as the gluing graph can be obtained as the "intersection" between the left and right graphs, and the morphisms l and r as the inclusions of K in L and R, respectively.

Figure 2.5: DPO graph rule



(a) Standard DPO rule notation



(b) Compact DPO rule notation

■

Definition 2.9 (Graph Transformation)**.** Given a (typed) graph rule $p = \left(L \xleftarrow{l} K \xrightarrow{r} R\right)$ and a (typed) graph G with a (typed) graph morphism $m : L \longrightarrow G$, called match, a direct (typed) graph transformation $G \overset{p,m}{\Longrightarrow} H$ from G to a (typed) graph H is a double-pushout (DPO) diagram such as:

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "m'"] \\
G & D \arrow[l, "f"] \arrow[r, "g"'] & H
\end{tikzcd}$$

[1]Also called graph transformation rule or graph production.

For $\mathbf{TGraph_T}$ there are two conditions, called *gluing or application conditions*, that must be satisfied so that the pushouts (1) and (2) exist and the graph rule can be applied:

- the *dangling condition* requires that no node can be deleted if it has incident edges that are not also deleted (otherwise the result of this deletion would not be a graph).
- the *identification condition* requires the match to not identify a deleted element with a preserved or (another) deleted one.

□

Example 2.10 (Graph Transformation Examples)**.** Figure 2.6a shows a transformation where the rule depicted in Figure 2.5 is successfully applied over a graph instance G_0.

Figure 2.6b shows the same rule being applied over a graph instance G_1 which does not satisfy the gluing conditions, more specifically it does not satisfy the dangling condition.

■

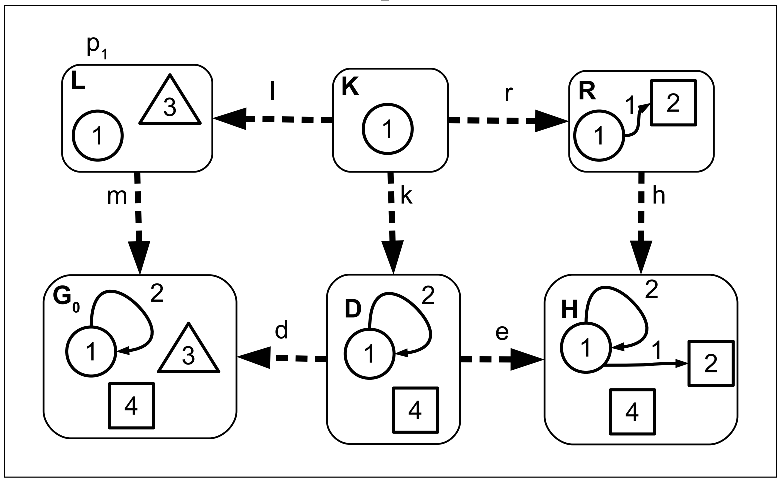
The gluing condition expresses in a positive manner whether it is possible to perform a transformation. Given a rule and a match to an instance graph, the pattern found by the match must satisfy the dangling and the identification conditions in order to be considered valid. However, sometimes it is also convenient to specify forbidden contexts, in which the transformation can not be applied even if it satisfies the gluing conditions. An elegant form of achieving this is by means of Negative Application Conditions (NACs), defined in (HABEL; HECKEL; TAENTZER, 1996).

A NAC extends one side of a rule by describing a pattern that should not be found around the match or comatch, otherwise the application is forbidden. Although NACs do not provide more expressive power, they make the modelling and understanding of graph grammars much easier. They are nowadays essential to the modelling of complex systems (CORRADINI; HECKEL, 2014).

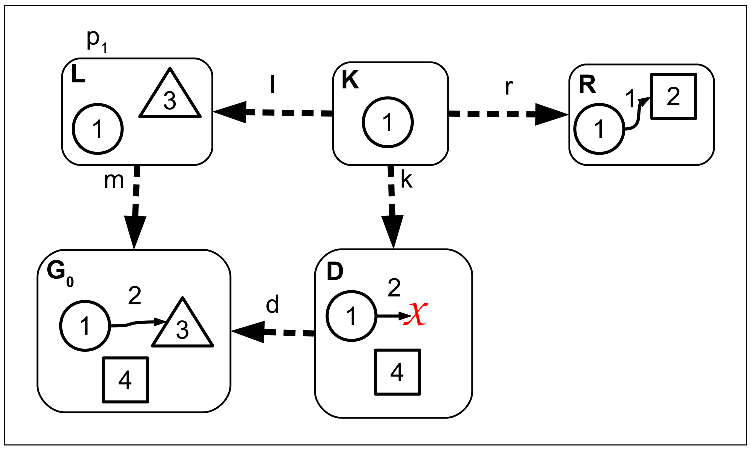
Definition 2.11 (Negative Application Condition)**.** A *left* negative application condition over a graph rule $p = \left(L \xleftarrow{l} K \xrightarrow{r} R\right)$ is of the form $NAC(n)$, where $n : L \to N$ is an arbitrary (typed) graph morphism. A match $m : L \to G$ of a rule p satisfies[2] $NAC(n)$ on L, written $m \models NAC(n)$, iff $\nexists\ q : N \to G$ with q being a monomorphism and $q \circ n = m$.

[2]When a NAC is satisfied it is also said that the NAC is *not triggered* and vice versa.

Figure 2.6: Graph transformation



(a) Successfully applied graph transformation



(b) Failing graph transformation due to the dangling condition

$$\begin{tikzcd}
N \arrow[dr, dotted, "q"', "\nmid" marking] & L \arrow[l, "n"'] \arrow[d, "m"] \\
& G
\end{tikzcd}$$

A match $m : L \longrightarrow G$ satisfies a set $NAC_L = \{NAC(n_i) | i \in I\}$ of left $NACs$, iff $m \models NAC(n_i)\ \forall i \in I$.

Analogously, a *right* negative application condition over a graph rule $p = \left(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R\right)$ is of the form $NAC(n)$, where $n : R \longrightarrow N$ is an arbitrary (typed) graph morphism. A comatch $m' : R \longrightarrow H$ of a rule p satisfies $NAC(n)$ on R (written $m' \models NAC(n)$) iff $\nexists\, q : N \longrightarrow H$ with q being a monomorphism and $q \circ n = m'$.

$$\begin{tikzcd}
R \arrow[r, "n"] \arrow[d, "m'"'] & N \arrow[dl, dotted, "q", "\nmid" marking] \\
H &
\end{tikzcd}$$

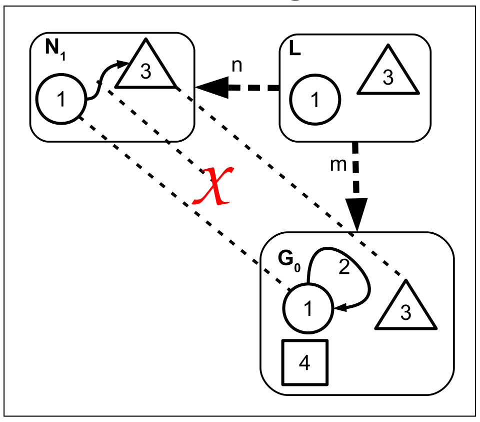
Also, a comatch $m' : R \longrightarrow H$ satisfies a set $NAC_R = \{NAC(n_i) | i \in I\}$ of right $NACs$, iff $m' \models NAC(n_i)\ \forall i \in I$.

□

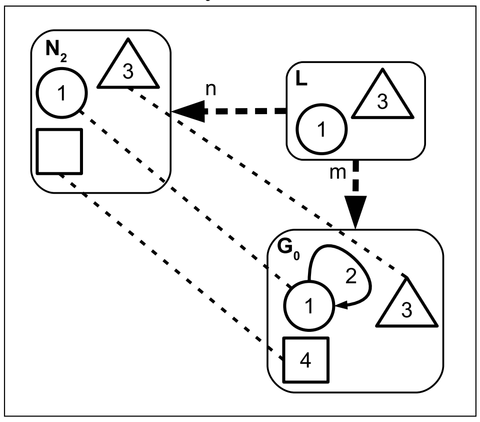
Example 2.12 (NAC and NAC satisfiability)**.** Figure 2.7a shows a NAC that is satisfied (i.e. not triggered) over a match m, as there is no possible way of mapping the edge between $\bigcirc_1$ and $\square_1$ in N to an edge in G_0 such that the resulting triangle commutes. Therefore, if the match over G_0 also satisfies the gluing conditions for the corresponding rule the transformation can be applied.

On the other hand, Figure 2.7b shows a NAC that is triggered (i.e. not satisfied) over the match m: all the elements in N can be mapped to G_0 such that the resulting triangle commutes. Therefore, even if the match satisfies the gluing conditions, the transformation can not be applied, as the pattern forbidden by the NAC was found on the instance graph.

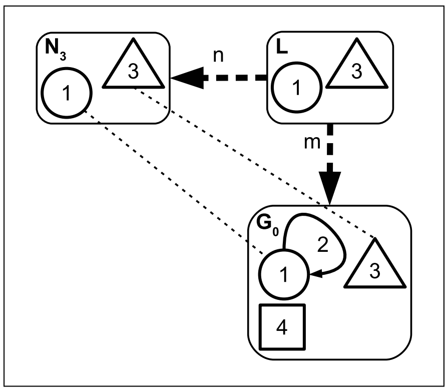
Figure 2.7: NACs and NAC satisfiability



(a) A satisfied NAC



(b) A triggered NAC



(c) A trivially-triggered NAC

■

Definition 2.13 (Trivially-Triggered NACs)**.** Given a $NAC(n)$, where $n : L \rightarrow \hat{L}$ is an isomorphism, and a match $m : L \rightarrow G$ which is a monomorphism, we call $NAC(n)$ a *trivially-triggered NAC* as, for every monomorphic m, there will always exist a $q : \hat{L} \rightarrow G$ injective such that $q \circ n = m$.

A trivially-triggered NAC $n : L \rightarrow \hat{L}$ is also denoted $NAC(L)$. If a rule p has a trivially-triggered NAC then p can never be applied, as the NAC will never be satisfied. An example of a trivially-triggered NAC is shown on Figure 2.7c. □

Definition 2.14 (Graph Transformation System and Graph Grammar)**.** A typed graph transformation system is a pair $GTS = (TG, P)$ where TG is the type graph of the system and P is a set of typed graph rules with NACs.

A typed graph grammar is a pair $GG = (GTS, I)$ where GTS is a typed graph transformation system and I is a typed start graph. It can also be notated as $GG = (TG, I^{TG}, P)$.

□

Example 2.15 (Mail Server Graph Transformation System)**.** Figure 2.8 depicts a graph transformation system that models a client-server scenario for a very simple e-mail application. The type graph (a) defines four kinds of nodes: 👤 for users, ✉ for messages, 📄 for data and 🗄 for mail servers, while the edges specify where each kind of node can be located at a given time. Finally, this system has four actions modelled as graph rules, which can be summarized as:

(b) *Send message:* a client writes a message which they send to a server, however there is a NAC forbidding the message of being sent if it has a piece of data attached to it.

(c) *Get data:* a piece of data is obtained from a server and attached to a message.

(d) *Receive message:* a server sends a message with attached data to a client.

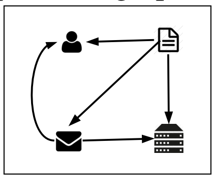
(e) *Delete message:* a client obtains a piece of data from a received message and this message is destroyed.

■

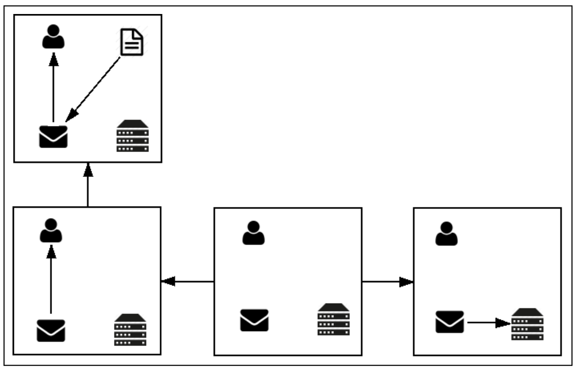
2.1 Parallel and Sequential Independence

One of the characteristics that make Graph Transformation Systems and Graph Grammars suitable formalisms for modelling and reasoning about parallel and/or concurrent systems is the possibility of checking whether the transformations given by two graph rules over the same instance graph can be applied (1) at the same time or (2) in

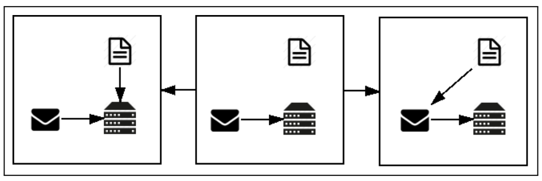
Figure 2.8: Mail application graph transformation system



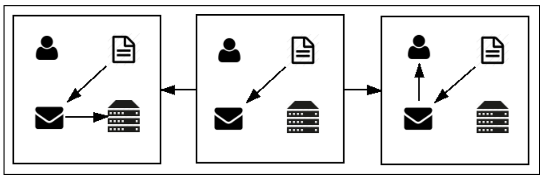
(a) Type Graph



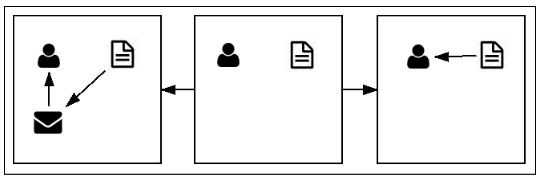
(b) Send message



(c) Get data



(d) Receive message



(e) Delete message

any interchangeable order. In the first case we say that the transformations are parallel independent; in the later we say that they are sequential independent.

In this section, we review both what it (formally) means for two graph transformations to be independent and how to check it. Notice that when we are reasoning about graph transformations (rules application) the (in)dependence is concrete, while for the case of graph rules the (in)dependence is potential, as it depends on the particular way the rules are applied.

Definition 2.16 (Causal Dependency)**.** Given two graph rules $p_1 = (L_1 \leftarrow K_1 \rightarrow R_1)$, $p_2 = (L_2 \leftarrow K_2 \rightarrow R_2)$ with NACs $n_1 : L_1 \rightarrow N_1$ and $n_2 : L_2 \rightarrow N_2$, we say that transformations $t_1 : H_1 \overset{p_1,m_1}{\Longrightarrow} E$ and $t_2 : E \overset{p_2,m_2}{\Longrightarrow} H_2$ as is the diagram below are *causally dependent* iff at least one of the following situations occurs:

1. $\nexists h_{12} : R_1 \rightarrow D_2$ such that $d_2 \circ h_{12} = m'_1$
2. $\exists! h_{12} : R_1 \rightarrow D_2$ such that $d_2 \circ h_{12} = m'_1$ but $e_2 \circ h_{12} \not\models NAC_{p_1^{-1}}$
3. $\nexists h_{21} : L_2 \rightarrow D_1$ such that $e_1 \circ h_{21} = m_2$
4. $\exists! h_{21} : L_2 \rightarrow D_1$ such that $e_1 \circ h_{21} = m_2$ but $d_1 \circ h_{21} \not\models NAC_{p_2}$

$$\begin{tikzcd}
N_1 & & & & N_2 & & \\
L_1 \arrow[u, "n_1"] \arrow[d, "m_1"'] & K_1 \arrow[l] \arrow[r] \arrow[d] & R_1 \arrow[rrrd, dotted, bend left, "h_{12}"] \arrow[rd, "m_1'"'] & & L_2 \arrow[u, "n_2"'] \arrow[llld, dotted, bend right, "h_{21}"'] \arrow[ld, "m_2"] & K_2 \arrow[l] \arrow[r] \arrow[d] & R_2 \arrow[d] \\
H_1 & D_1 \arrow[l, "d_1"] \arrow[rr, "e_1"'] & & E & & D_2 \arrow[ll, "d_2"] \arrow[r, "e_2"'] & H_2
\end{tikzcd}$$

We say that rules p_1 and p_2 are (potentially) causally dependent iff a diagram like the one above fulfilling at least one of the mentioned situations exists. □

Intuitively, each case of dependency can be regarded as follows:

1. a *deliver-delete* dependency: p_2 deletes (from graph E) at least one element that was created or preserved by p_1.

2. a *forbid-produce* dependency: p_2 creates on H_2 at least one element that would trigger the NAC N_1^{-1}.

3. a *produce-use* dependency: p_1 creates (on graph E) at least one element needed for p_2 to be applied which did not exist on H_1.

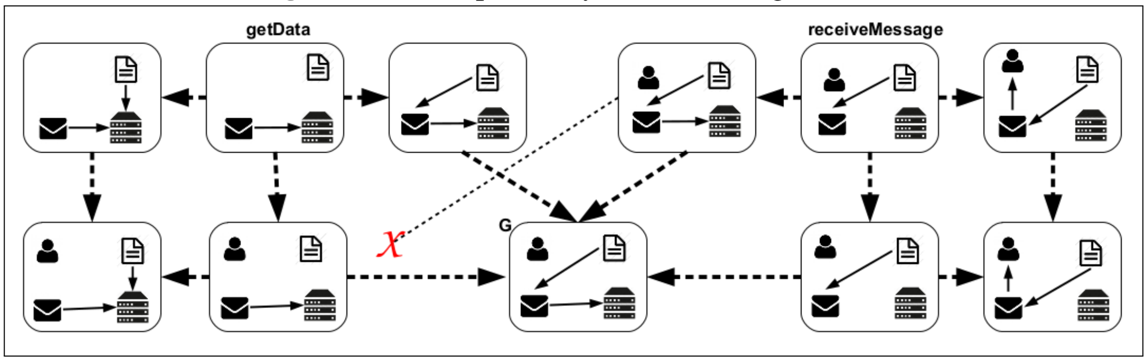
4. a *delete-forbid* dependency: p_1 deletes (from graph H_1) at least one element that would trigger the NAC N_2, thus allowing the application of p_2 on E.

In this work we mostly consider dependencies (and conflicts) between rules. Thus, when we say that two rules are causally dependent it means that there may be a situation in which the application of one rule actually depends on the application of the other, but there might also be situations in which both these rules are applied independently.

Example 2.17 (Dependency situation in the mail server grammar)**.** Figure 2.9 shows a dependency situation between the rules *getData* and *receiveMessage*. In this case, the dependency is of a *produce-use* kind: there may be a situation in which the edge between the *piece of data* and the *message* at the instance graph created by *getData* are necessary for *receiveMessage* to be applied.

In the diagram, it is not possible to find a morphism from the left-hand-side of *receiveMessage* to the gluing graph of *getData* such that the transformation is still valid. Thus, *receiveMessage* is causally dependent on *getData*. ■

Figure 2.9: A dependency in the server grammar



Definition 2.18 (Conflict)**.** Given two graph rules $p_1 = (L_1 \leftarrow K_1 \rightarrow R_1)$ and $p_2 = (L_2 \leftarrow K_2 \rightarrow R_2)$ with NACs $n_1 : L_1 \rightarrow N_1$ and $n_2 : L_2 \rightarrow N_2$, we say that transformations $t_1 : E \overset{p_1,m_1}{\Longrightarrow} H_1$ and $t_2 : E \overset{p_2,m_2}{\Longrightarrow} H_2$ as in the diagram bellow are in conflict iff at least one of the following situations occurs:

1. $\nexists h_{12} : L_1 \rightarrow D_2$ such that $d_2 \circ h_{12} = m_1$
2. $\exists! h_{12} : R_1 \rightarrow D_2$ such that $d_2 \circ h_{12} = m_1$ but $e_2 \circ h_{12} \not\models NAC_{p_1}$
3. $\nexists h_{21} : L_2 \rightarrow D_1$ such that $d_1 \circ h_{21} = m_2$
4. $\exists! h_{21} : L_2 \rightarrow D_1$ such that $d_1 \circ h_{21} = m_2$ but $e_1 \circ h_{21} \not\models NAC_{p_2}$

$$\begin{tikzcd}
 & & N_1 & & N_2 & & \\
R_1 \arrow[d] & K_1 \arrow[l] \arrow[r] \arrow[d] & L_1 \arrow[u, "n_1"] \arrow[dr, "m_1"] \arrow[rrrd, dotted, bend left, "h_{12}"] & & L_2 \arrow[u, "n_2"'] \arrow[dl, "m_2"'] \arrow[llld, dotted, bend right, "h_{21}"'] & K_2 \arrow[l] \arrow[r] \arrow[d] & R_2 \arrow[d] \\
H_1 & D_1 \arrow[l, "e_1"] \arrow[rr, "d_1"'] & & E & & D_2 \arrow[ll, "d_2"] \arrow[r, "e_2"'] & H_2
\end{tikzcd}$$

We say that rules p_1 and p_2 are (potentially) conflicting iff a diagram like the one above fulfilling at least one of the mentioned situations exists. □

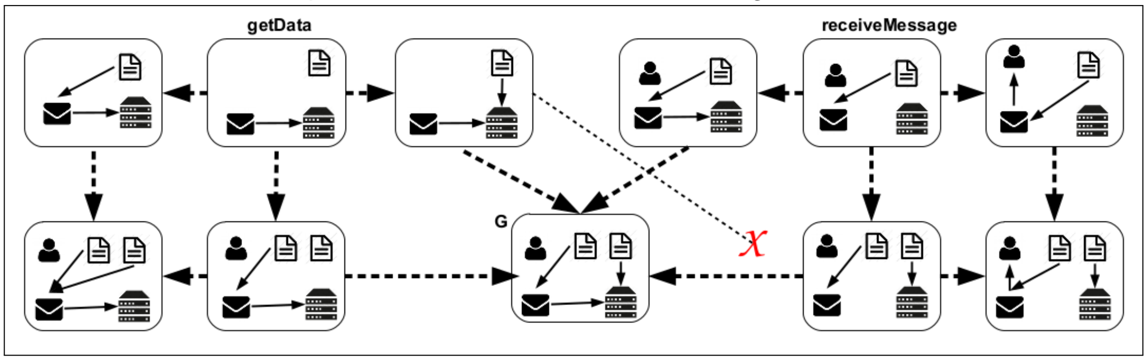
Intuitively, each conflict case can be regarded as:

1. a *delete-use* conflict: p_2 deletes (from graph E) at least one element needed for p_1 to be applied.
2. a *produce-forbid* conflict: p_2 produces (on graph H_2) at least one element that triggers the NAC N_1.
3. a *delete-use* conflict: p_1 deletes at least one element needed for p_2 to be applied

4. a *produce-forbid* conflict: p_1 creates at least one element that triggers the NAC N_2.

Example 2.19 (Conflict situation in the mail server grammar)**.** Figure 2.10 shows a conflict situation involving the rules *getData* and *receiveMessage*. This is a *delete-use* conflict: *receiveMessage* deletes from the overlapping graph an edge between the *message* and the *server* which may be necessary for *getData* to be applied. Considering the matches in Figuere 2.10, both transformations are valid. However, once *receiveMessage* is applied, it is no longer possible to apply *getData*. This is represented in the diagram by the fact that it is not possible to find a morphism from the left-hand-side of *getData* to the gluing graph of *receiveMessage* such that the transformation from there is still valid.

Figure 2.10: A conflict in the server grammar



■

3 OCCURRENCE GRAPH GRAMMARS WITH NACS

Occurrence Graph Grammars (OGGs) were defined for the Single-Pushout (SPO) approach by (RIBEIRO, 1996), and for the Double-Pushout (DPO) approach by (CORRADINI; MONTANARI; ROSSI, 1996). In both cases they consist of a way of representing the concurrent semantics of a graph grammar as a graph grammar itself. It also presents the advantage of having a more compact structure than other semantic models for graph grammars. For example, with Unfolding (RIBEIRO, 1996) the semantics of a graph grammar is given by the set of all possible derivations of that grammar, which may be an infinite structure, whereas with OGGs we have a grammar which already encodes such derivations in a finite way.

The aim of an OGG is to describe all possible states and changes of states of the graph grammar from which it was constructed. This is possible because occurrence grammars can be regarded as an execution history of the underlying grammar. This history is encoded under the forms of (1) a *core graph* containing all elements ever used in the execution of the grammar, (2) a set of *actions*, which are rule applications typed over this core graph and (3) a set of *relations* between these actions and the elements of the core graph.

These relations express dependencies among core graph elements, such as which of them must occur together, at the same state, which ones must be created/deleted one after the other, which elements must never occur together, and so on. They also encode restrictions over the application of rules, such as which rules are sequentially dependent on others or whether it is possible to successfully apply all the rules of a grammar, etc.

However, Negative Application Conditions are not addressed by the original definitions of occurrence grammars. Still, NACs are nowadays essential for modelling complex systems as graph grammars, given they provide more compact mechanisms to control the application of rules (HABEL; HECKEL; TAENTZER, 1996; LAMBERS et al., 2008; CORRADINI; HECKEL, 2014). In this chapter, after reviewing the works of (RIBEIRO, 1996) and (CORRADINI; MONTANARI; ROSSI, 1996) in OGGs, an extension of the occurrence grammars framework towards contemplating negative application conditions is presented, which is part of our thesis contribution.

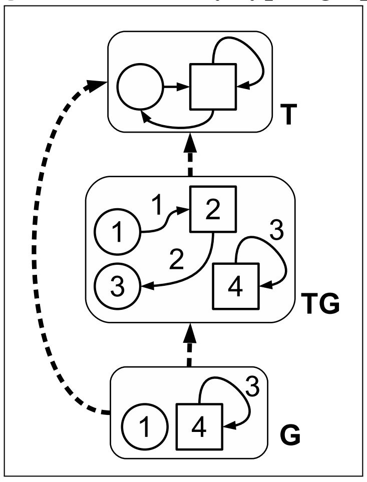
3.1 Overview of Occurrence Graph Grammars without NACs

Definition 3.1 (Doubly-Typed Graph)**.** Given a type graph T, a *doubly-typed graph* G^{TG^T} over T is a tuple $G^{TG^T} = \left(G^T, TG^T, t^{G^T} : G^T \rightarrow TG^T\right)$ where G^T and TG^T are typed graphs over T and $t^{G^T} : G^T \rightarrow TG^T$ is a typed graph morphism in $\mathbf{TGraph_T}$. We call TG^T the *double-type graph* and t^{G^T} the double-typing morphism. □

Remark (Typing Morphism)**.** As we are interested in occurrence graph grammars, we consider only doubly-typed graphs whose typing morphism $type_{TG} : TG \rightarrow T$ is an epimorphism. This has the effect that every element present in T is the image of at least one element from TG. □

Example 3.2 (Doubly-Typed Graph Example)**.** Figure 3.1 shows a doubly-typed graph G^{TG^T}. ■

Figure 3.1: Doubly-typed graph



Definition 3.3 (Doubly-Typed Graph Morphism)**.** Given two doubly-typed graphs G^{TG^T} and H^{TG^T} and a graph morphism $g^T : G^T \rightarrow H^T$, we say that g^T is a TG^T*-doubly-typed graph morphism* if the following diagram commutes:

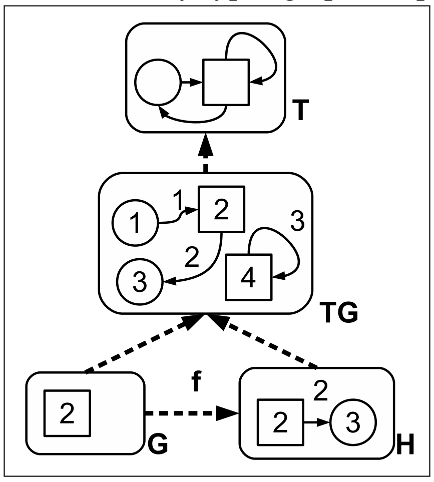
$$\begin{tikzcd}
G \arrow[rr, "g"] \arrow[dr, "t^G"'] & & H \arrow[dl, "t^H"] \\
& TG \arrow[d, "type_{TG}"] & \\
& T &
\end{tikzcd}$$

□

Notice that the (single) type morphisms $type_G : G \rightarrow T$ and $type_H : H \rightarrow T$ can be obtained respectively as $type_{TG} \circ t^G$ and $type_{TG} \circ t^H$.

Example 3.4 (Doubly-Typed Graph Morphism Example)**.** Figure 3.2 shows a doubly-typed graph morphism $f : G^{TG^T} \to H^{TG^T}$. ■

Figure 3.2: Doubly-typed graph morphism



Remark. (RIBEIRO, 1996) defined different kinds of doubly-typed graph morphisms based on whether the double-type graphs and the type graphs are or are not the same. In this work we are only interested in the case where all doubly-typed graphs share exactly the same double-type and type graphs. Therefore, through the rest of this work, we refer to TG^T*-doubly-typed graph morphisms* simply by *doubly-typed graph morphisms.* □

Definition 3.5 (Doubly-Typed Graph Rule)**.** A doubly-typed graph rule $p^{TG^T} = \left(L^{TG^T} \leftarrow K^{TG^T} \to R^{TG^T}\right)$ is a span of injective doubly-typed graph morphisms $l : K \to L$ and $r : K \to R$.

$$\begin{tikzcd}
L \arrow[dr] & K \arrow[l] \arrow[r] \arrow[d] & R \arrow[dl] \\
 & TG \arrow[d] & \\
 & T &
\end{tikzcd}$$

Given a doubly-typed graph rule $p^{TG^T} = \left(L^{TG^T} \leftarrow K^{TG^T} \to R^{TG^T}\right)$, its inverse rule is defined by $\left(p^{TG^T}\right)^{-1} = \left(R^{TG^T} \leftarrow K^{TG^T} \to L^{TG^T}\right)$.

Let the double-typing morphisms from L^{TG^T}, K^{TG^T} and R^{TG^T} be t^{L^T}, t^{K^T} and t^{R^T}, respectively. For a rule $a = p^{TG^T}$ we call:

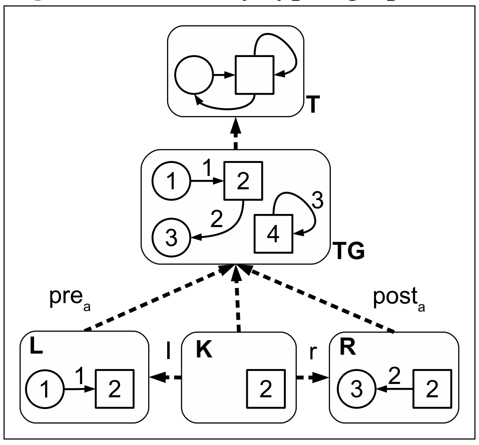
- $L_a = L_T$, $K_a = K_R$ and $R_a = R_T$, the left, gluing and right graphs of a.
- $pre_a = t^{L^T} : L^T \to TG^T$, the *pre-condition* of a.

- $post_a = t^{R^T} : R^T \longrightarrow TG^T$, the *post-condition* of a.

□

Example 3.6 (Doubly-Typed Graph Rule Example)**.** Figure 3.3 shows a doubly-typed graph rule which deletes an edge $\curvearrowleft_1$ and a node $\bigcirc_1$, preserves a $\square_2$ and creates a node $\bigcirc_3$ and an edge $\curvearrowleft_2$. ■

Figure 3.3: Doubly-typed graph rule



Definition 3.7 (Negative Application Conditions on Doubly-Typed Graph Rules)**.** A *left* negative application condition over a doubly-typed graph rule p^{TG^T} is of the form $NAC(n^T)$, where $n^T : L^T \longrightarrow N^T$ is an arbitrary (single-)typed graph morphism.

A (doubly-typed) match morphism $m^{TG^T} : L^{TG^T} \longrightarrow G^{TG^T}$ of a rule p^{TG^T} satisfies $NAC(n^T)$ on L^{TG^T}, written $m^T \models NAC(n^T)$, iff $\nexists\ q^T : N^T \longrightarrow G^T$ where q^T injective and $q^T \circ n^T = m^T$.

$$\begin{tikzcd}
T & TG \arrow[l] \\
N \arrow[u] \arrow[dr, dotted, "q"', "|" marking] & L \arrow[l, "n"'] \arrow[u] \arrow[d, "m"] \\
& G \arrow[uu, bend right=60]
\end{tikzcd}$$

A match $m^{TG^T} : L^{TG^T} \longrightarrow G^{TG^T}$ satisfies a set $NAC_L = \{NAC\left(n_i^T\right) | i \in I\}$ of left $NACs$, iff $m^T \models NAC\left(n_i^T\right)\ \forall i \in I$.

Right negative application conditions are defined analogously for the right hand side of a rule and its comatch. □

Remark. We could have defined NACs whose morphisms are doubly-typed graph morphisms, which would then act specifically over doubly-typed graphs. However, the use of this kind of NAC would imply mapping all possible ways a NAC typed over T could be translated into a NAC typed over TG, which would lead to a doubly-typed grammar with a set of NACs much bigger than the original one. Thus we do not use this kind of NACs given our interest in a compact and more abstract characterization for the semantics of graph grammars. Therefore, we use *single-typed* NACs as the only NAC type in all of our *doubly-typed graph grammars*, and $NAC(n)$ as a synonym of $NAC(n^T)$. □

In a grammar without NACs, if there is a sequence of graph transformations $t_0 \dots t_n$ where each pair (t_i, t_{i+1}) of consecutive transformations is sequentially independent, then it is possible to *switch* the order of application for any pair in that sequence an arbitrary number of times and still achieve the same final graph as a result (up to isomorphism). This property is called *switch equivalence* (CORRADINI et al., 2013).

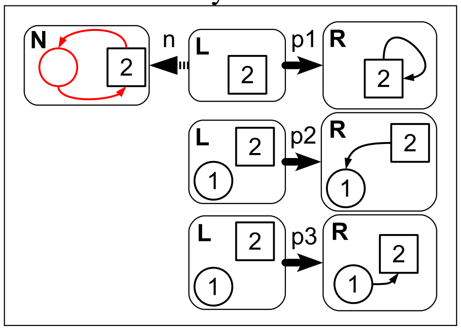
However, the switch equivalence does not always hold when the grammar has NACs. This happens because there may be situations where the NAC of a rule can be triggered by the cumulative effect of applying two (or more) other rules, while the same rules would not trigger that NAC in isolation. This may lead to a situation where conflicts and dependencies are not *stable under switch*, which means that conflicts or dependencies that do not occur in a given sequence of transformations may arise if one or more pairs of transformations are switched. An example is shown in Figure 3.8.

Example 3.8 (Switch Equivalence)**.** The rules depicted in Figure 3.4 show a situation where the independence between rules is not stable under switch equivalence.

In this example, all rules are independent and also do not conflict with each other in the sense of Definitions 2.16 and 2.18. Thus, in theory, it should be possible to apply these rules in any order or in parallel. Particularly, it may be possible to apply all rules in the order they appear in the figure: $[p_1, p_2, p_3]$. It may also be possible to apply them in the orders $[p_1, p_3, p_2]$, $[p_2, p_1, p_3]$ and $[p_3, p_1, p_2]$. However, it is not possible to perform a derivation using the rules in the orders $[p_2, p_3, p_1]$ or $[p_3, p_2, p_1]$. This problem arises from the fact that p_2 and p_3 independently create a piece of the pattern forbidden by the NAC of p_1 in a way that, although the effect of each rule by itself does not trigger the NAC of p_1, their combined effects do. ■

In order to avoid this problem while we construct a canonical representation of several possible derivations for a set of rules, we restrict the use of NACs to a special type of NACs called *incremental NACs*. *Incremental NACs* were originally defined in (CORRADINI et al., 2013) and (CORRADINI; HECKEL, 2014). They have the property of

Figure 3.4: Instability of conflicts under shift



extending the forbidden context of a match by a single edge or a single node. Thus, each NAC forbids only one element at a time and therefore there is no possible way to trigger a NAC by the cumulative effects of more than one rule.

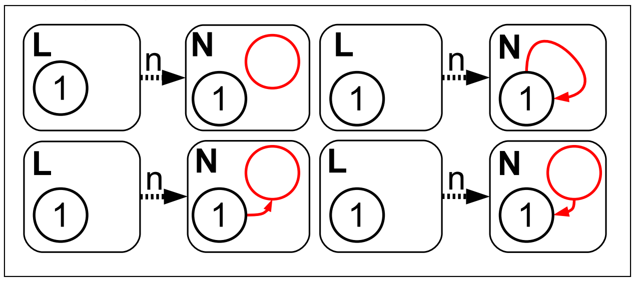
Definition 3.9 (Incremental NACs)**.** Given a monomorphism $n : L \to N$, $NAC(n)$ is said to be incremental if for any possible pair of decompositions $g_1 : L \to O_g; g_2 : O_g \to N$ and $f_1 : L \to O_f; f_2 : O_f \to N$ as in the diagram below, where all morphisms are monos and $f_1; f_2 = n = g_1; g_2$, there exists a mediating morphism $o_1 : O_g \to O_f$ or $o_2 : O_f \to O_g$, such that the resulting triangles commute.

$$\begin{tikzcd}
 & O_g \arrow[dr, "g_2"] \arrow[dd, dotted, bend right, "o_1"'] & \\
L \arrow[ur, "g_1"] \arrow[rr, "n"] \arrow[dr, "f_1"'] & & N \\
 & O_f \arrow[ur, "f_2"'] \arrow[uu, dotted, bend right, "o_2"'] &
\end{tikzcd}$$

□

Example 3.10 (Incremental NACs)**.** Figure 3.5 shows all possible (canonical) formats that any valid incremental NAC may assume.

Figure 3.5: Canonical Incremental NACs



■

At first, it may seem that we are losing expressive power by restricting the NACs used in our grammars to incremental NACs only. However (CORRADINI et al., 2013) have shown two important results regarding them. First, they showed that incremental NACs are sufficient to model most of real applications using GTSs. Second, they presented an algorithm to compile rules with general NACs to rules with incremental NACs only, generating a new GTS that is able to simulate the original one.

Notation (Set Operations over Graphs)**.** Given a graph G, we sometimes view them as being composed of a set $V(G)$ of nodes and a set $E(G)$ of edges, denoted $G = V(G) \cup E(G)$, in order to allow the use of set operations over this graph. For example, we say that an element (a node or an edge) x is a member of G, denoted $x \in G$, iff $x \in V(G) \cup E(G)$. Moreover, any operations involving multiple graphs are applied *setwise*. For example, given two graphs G_1 and G_2, the difference $D = G_1 - G_2$ between them is the union of the differences between their sets of nodes and their sets of edges. Therefore $D = V(D) \cup E(D)$ where $V(D) = V(G_1) - V(G_2)$ and $E(D) = E(G_1) - E(G_2)$.

Any other set operations applied over graphs are regarded likewise.

□

Definition 3.11 (Triggering element)**.** Given a rule $p = \left(L \xleftarrow{l} K \xrightarrow{r} R\right)$ with an incremental, non-trivially triggered NAC $n : L \longrightarrow N$, and a monomorphic match $m : L \longrightarrow G$, where there is an injective $q : N \longrightarrow G$, therefore $m \not\models NAC(n)$. There is exactly one element that completes the match towards triggering the NAC. This element is present in the difference between the images of q and m.

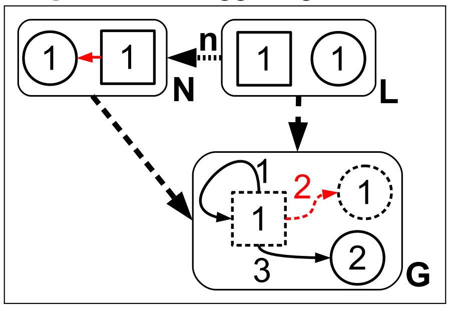
Let $G_{|N}$ be the image of q, $G_{|L}$ the image of m, and $D = G_{|N} - G_{|L}$. The triggering element of this NAC is:

- $x \in E(D)$, if $E(D) \neq \varnothing$;
- $x \in V(D)$ otherwise.

□

Example 3.12 (Triggering Element)**.** Given that incremental NACs extend the match by forbidding only one element at a time, this element can be easily identified at the instance graph: it corresponds to the solely element mapped by the morphism $q : N \longrightarrow G$ which is not also mapped by $m : L \longrightarrow G$. Figure 3.6 shows an example. The elements in the instance graph mapped by either q or m are represented by dashed lines. Notice that the only element which is not in the image of both morphisms is $\curvearrowleft_2$, therefore $\curvearrowleft_2$ is the triggering element of the NAC for this match. ■

Figure 3.6: Triggering Element



Definition 3.13 (Doubly-Typed Graph Grammars)**.** A *doubly-typed graph grammar* is a tuple $GG = \left(TG^T, I^{TG^T}, P\right)$ where TG^T is the double-type graph of the grammar, I^{TG^T} is a doubly-typed graph called the *initial graph* and P is a set of doubly-typed graph rules. We call $in_{GG} = I^T \rightarrow TG^T$, the morphism that maps the initial graph into the double-type graph.

□

Definition 3.14 (Core Graph)**.** Given a doubly-typed graph grammar $GG = \left(C^T, I^{C^T}, P\right)$, C^T is a *core graph* iff the following two conditions hold:

(uniqueness of origin) $\forall x \in C^T \colon \exists! y \in (I^T \uplus (\uplus_{i \in P}(R_i - K_i))$:

$$x = \begin{cases} in_{GG}(y), \text{if } y \in I^T \\ post_i(y), \text{if } y \in R_i - K_i \end{cases}$$

(uniqueness of deletion) $\forall x \in C^T \colon \exists^{\leq 1} y \in \uplus_{i \in P}(L_i - K_i)$:

$$x = \begin{cases} pre_i(y), \text{if } y \in L_i - K_i \end{cases}$$

□

The first condition assures that every element in the *core graph* was either already present in the initial graph or was created by one and only one rule. The second condition assures that for every element that is deleted, it is deleted only once by only one rule. The idea is that each element within a *core graph* has a unique origin. At the same time, the *core graph* contains all elements created, deleted or preserved by all rules in its underlying grammar.

In (RIBEIRO, 1996) the second condition was not used, because of a peculiarity of the SPO approach where more than one rule can delete the same element at the same time, while this would raise a conflict, and therefore be forbidden, in the DPO approach. As a consequence, the occurrence grammars defined by (RIBEIRO, 1996) are inherently non-deterministic, whereas ours are deterministic. In practice, this means that the semantics of a graph grammar in the SPO approach can be achieved with only one occurrence grammar, while in the DPO approach we need a set of them.

Example 3.15 (Doubly-Typed Graph Grammar and Core Graph Example)**.** Figure 3.7a shows a doubly-typed graph grammar, whose double-type is not a core graph. That happens because $\square_2$ is created by both rules p_1 and p_2, as well as $\bigcirc_1$ is deleted by both p_2 and p_3.

On the other hand, Figure 3.7b shows a doubly-typed graph grammar whose double-type is also a core graph: $\square_2$ and $\curvearrowleft_1$ are created by p_1, $\curvearrowleft_3$ by p_2, $\bigcirc_3$ and $\curvearrowleft_2$ by p_3 and $\bigcirc_1$ is present on the initial graph. Also, the deleted elements are deleted only once: $\bigcirc_1$ and $\curvearrowleft_1$ by p_3.

It is important to notice that, even though the TG graphs in both grammars are isomorphic, only the one in Figure 3.7b is a core graph. This can be explained by looking at their underlying grammars (initial graphs and rules), where one of them satisfies the conditions presented in Definition 3.14, whereas the other does not.

■

Notation (Restriction to Image)**.** Given an arbitrary morphism $f : A \rightarrow X$, we denote as $f' : A \rightarrow X_{|A}$ the morphism derived from f where $X_{|A}$ is the image of f.

For two arbitrary morphisms $f : A \rightarrow X$ and $g : B \rightarrow X$, we denote as $f' : A \rightarrow X_{|AB}$ and $g' : B \rightarrow X_{|AB}$ the morphisms derived from f and g where $X_{|AB}$ is the joint image of both f and g. □

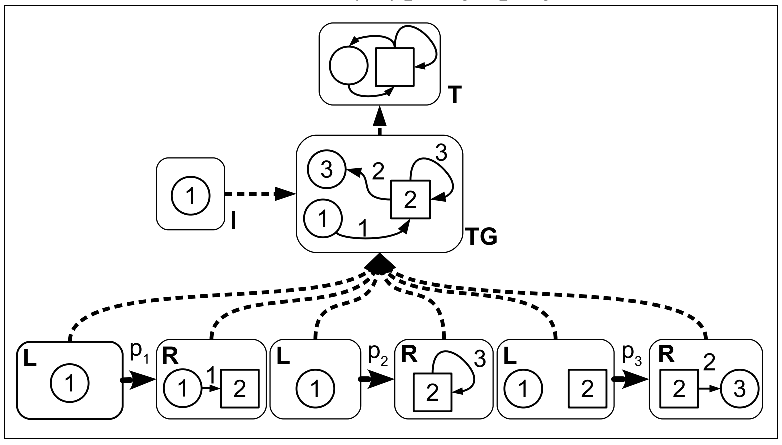
Definition 3.16 (Strongly Safe (Doubly-Typed) Graph Grammars)**.** Given $GG = \left(C^T, I^{C^T}, P\right)$ a doubly-typed graph grammar, GG is said to be *strongly safe* if its double-type graph is also a core graph.

Each rule in a strongly safe graph grammar is also called an *action*. We say that an action a creates an element e iff $e \in R(a) - K(a)$. Similarly, a deletes e iff $e \in L(a) - K(a)$. Finally, if e is present in $K(a)$, we say that a preserves e. □

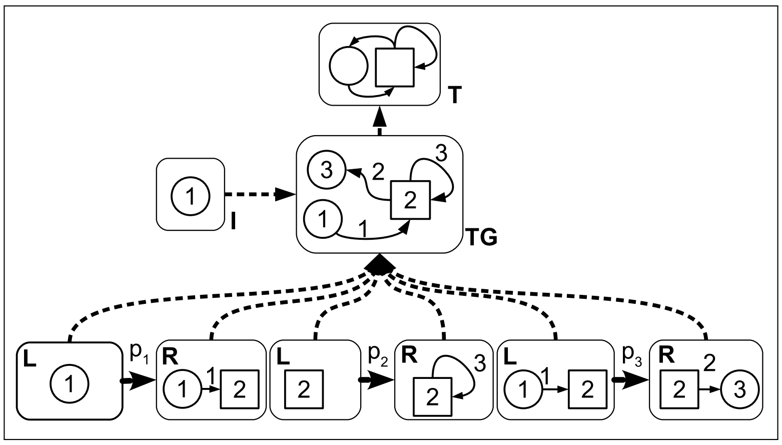
In the context of strongly safe graph grammars, we use a slightly different interpretation of a graph transformation:

- Isolated actions are always applied over a subgraph of the core graph: the match of

Figure 3.7: Doubly-typed graph grammars



(a) A grammar with a double-type graph that is not a core graph.



(b) A grammar with a double-type graph that is also a core graph.

an action is equal to its pre-condition $pre_a : L \longrightarrow C^T_{L_a}$, as well as the comatch is equal to its post-condition $post_a : R \longrightarrow C^T_{R_a}$.

- When searching for conflicts (resp. dependencies) between two actions, the overlapping between them is the restricted image of their matches $E = C^T_{|L_1L_2}$ (resp. comatch and match $E = C^T_{|R_1L_2}$). The derived graphs are calculated accordingly when the DPO transformations exist, i.e. $E \overset{a_1}{\Longrightarrow} H_1$ (resp. $E \overset{a_1^{-1}}{\Longrightarrow} H_1$), $E \overset{a_2}{\Longrightarrow} H_2$. Notice that, as the transformations are always concrete regarding the core graph, we have $H_1 \hookrightarrow C^T, H_2 \hookrightarrow C^T$.

- When searching for conflicts (resp. dependencies) between two actions a_1, a_2 with NACs, we check the NAC satisfiability only in the overlapping and the derived graphs rather than the entire core graph.

By restricting the overlapping of actions to the images of their matches (resp.

comatch and match) we consider only the local effects of these actions. Thus resulting in two properties of our interest: (1) We avoid dangling conditions with edges in the core graph that do not directly participate in the interaction of these actions, similarly, (2) the NACs of these actions are not triggered by elements that, despite of being present in the core graph, do not directly participate in the interaction of these two actions. These restrictions, together with the use of incremental NACs, allow us to locally compute the conflicts and dependencies for each pair of actions, without the necessity of dealing with global effects other actions might have.

Remark (Strongly Safe Grammars)**.** Throughout the remaining of this work, we use only strongly safe grammars whose set P of actions is finite and each action in P is distinct from the others.

$\square$

Example 3.17 (Strongly Safe Graph Grammar)**.** Figure 3.7b also depicts a strongly safe graph grammar as the double-type graph of the grammar is, in fact, a core graph. $\blacksquare$

3.2 Relations within Strongly Safe Graph Grammars without NACs

Given a strongly safe graph grammar, its core graph contains all elements used (created, read or deleted) during one possible execution of the grammar. Moreover, as each element has a unique origin, the core graph can be considered to contain the entire "execution history" of its underlying grammar.

We are here interested in some of the properties that can be found by looking at this history. Particularly, we want to define the kind of relations that exist among actions and elements, whether it is possible to find sequences in which all actions are applied and which graphs can be considered valid (reachable) by this grammar.

In (RIBEIRO, 1996), causal, conflict and occurrence relations for strongly safe graph grammars were defined. There, the graph transformation approach used was the *Single Pushout* (SPO) without NACs. (CORRADINI; MONTANARI; ROSSI, 1996) also defined a different notion of causal relation, equivalent to the occurrence relation in the previous mentioned work, with respect to the DPO approach without NACs. Both authors use these relations to find out whether all the actions of a strongly safe graph grammar are applicable and prove the above mentioned properties about them. However, these relations alone are not sufficient to prove such properties when the actions have NACs. Here we recall the work of (CORRADINI; MONTANARI; ROSSI, 1996), since it already uses the DPO approach, then extend it to create an equivalent notion of *occurrence relation* that works for grammars in the DPO approach with NACs.

Definition 3.18 (Causal Relation)**.** *(This is the same causal relation defined in (CORRADINI; MONTANARI; ROSSI, 1996) for the DPO approach without NACs.)* Given $GG = \left(C^T, I^{C^T}, P\right)$ a strongly safe graph grammar, actions $a_1, a_2 \in P, a_1 \neq a_2$ and an element $e \in N(C^T) \cup E(C^T)$, then:

1. If a_1 deletes e, then $e <_c a_1$.
2. If a_1 creates e, then $a_1 <_c e$.
3. If a_1 creates e and a_2 preserves e, then $a_1 <_c a_2$.
4. If a_1 preserves e and a_2 deletes e, then $a_1 <_c a_2$.
5. The *causal relation* $\leq_c$ of $P \cup N(C^T) \cup E(C^T)$ is the reflexive and transitive closure of $<_c$.

□

This relation represents conditions over creation, use (preservation) and deletion of elements by the actions used to characterize executions of the underlying rules. In any of the derivations represented by this strongly safe graph grammar, an action a must occur after all actions that create the elements it preserves or deletes. Analogously, a must occur before all actions that delete the elements created or preserved by it.

In (CORRADINI; MONTANARI; ROSSI, 1996) it is shown that if this relation is a *partial order*, then any total order with respect to it is a sequencing in which all productions of the underlying grammar are applicable.

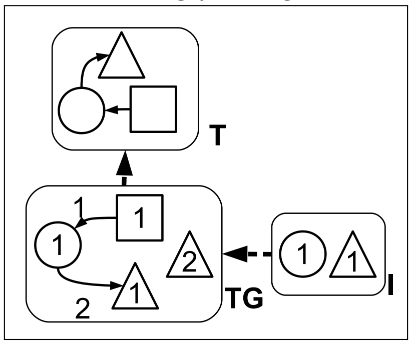
Example 3.19 (Causal Relation in Grammars without NACs)**.** Given the strongly safe grammar corresponding to the core and initial graphs in Figure 3.8a and the set of rules in Figure 3.8b, we have that:

- $a_1 <_c \triangle_2$
- $a_2 <_c \square_1$ and $a_2 <_c \curvearrowleft_1$
- $a_3 <_c \curvearrowleft_2$

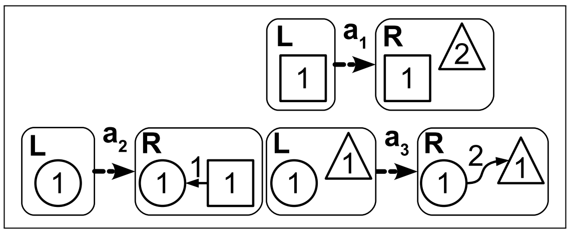
- $a_2 <_c a_1$ by creation/preservation of $\square_1$

The causal relation for this grammar is (without the pairs due to reflexivity): $a_1 \leq_c \triangle_2, a_2 \leq_c \square_1, a_2 \leq_c \curvearrowleft_1, a_3 \leq_c \curvearrowleft_2, a_2 \leq_c a_1, a_2 \leq_c \triangle_2$. Notice that the only pair in this relation where both elements are actions is $a_2 \leq_c a_1$, therefore, we have that all actions in this grammar can be applied as long as a_2 is applied before a_1 (a_2 crates the element $\square_1$ which is necessary for a_1 to be applied). In particular, the following sequences of actions are valid and lead to the same resulting graph: $[a_2, a_1, a_3], [a_2, a_3, a_1], [a_3, a_2, a_1]$.

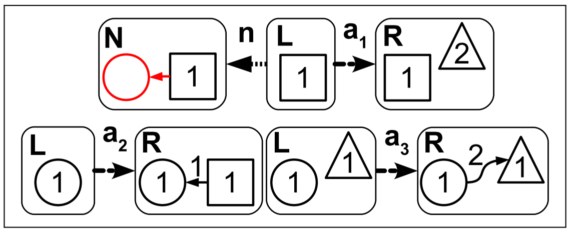
Figure 3.8: Strongly safe grammar GG1



(a) Core and initial graphs



(b) A strongly safe grammar without NACs



(c) A strongly safe grammar with NACs

■

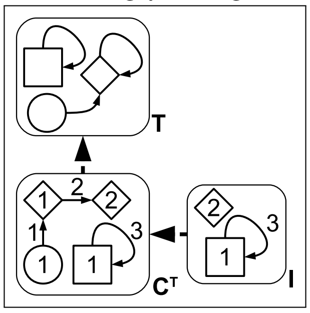
The same definition can be attempted in a strongly safe grammar where actions are equipped with NACs. However, as shown in examples 3.20 and 3.21, it lacks the same properties as in the case without NACs.

Example 3.20 (Causal Relation in Grammars with NACs)**.** Consider the strongly safe grammar corresponding to the core and initial graphs in Figure 3.8a and the set of rules in Figure 3.8c.

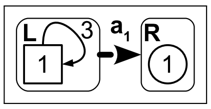
We have the same causal relation as the one presented in Example 3.19, since the structure of the actions is the same in both examples, except for the NAC in the action a_1 on Figure 3.8c. In this example, we still have that a_2 must be applied first in order for a_1 to be applied. However, besides creating the $\square_1$ needed for a_1, a_2 also creates a $\curvearrowleft_1$ from $\square_1$ to $\bigcirc_1$ which is a pattern forbidden by the NAC of a_1. Therefore, we have that a_2 also causes a *produce-forbid* conflict with a_1. Moreover, a_3 is the only other action in this grammar and it does not delete any element that could undo the forbidden pattern. Thus, there is no possible way of applying all actions of this grammar, even though the causal relation is a partial order. ■

Example 3.21 (Causal Relation in Grammars with NACs)**.** Consider the strongly safe grammar depicted on Figure 3.9, for which we have $a_1 \leq_c a_3, a_2 \leq_c a_3, a_2 \leq_c a_4, a_2 \leq_c a_1$ as causal relation: in order to know if all the actions of a strongly safe grammar without NACs can be applied, it would be sufficient to check whether the causal relation is a partial order.

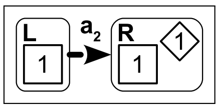
Figure 3.9: Strongly safe grammar GG2



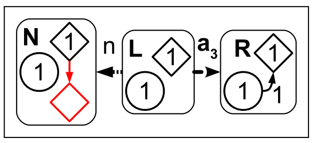
(a) Core and initial graphs



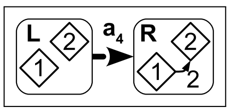
(b) Action a_1



(c) Action a_2



(d) Action a_3



(e) Action a_4

In particular, if we do not consider NACs for the grammar in Figure 3.9, any total order of actions compatible with the partial order of the causal relation would be a valid sequencing of this grammar. As an example, $[a_2, a_1, a_3, a_4]$ and $[a_2, a_4, a_1, a_3]$ are valid sequences (without NACs) for this grammar.

However, not all previous sequences are valid when NACs come into play. Specifically, $[a_2, a_4, a_1, a_3]$ is not valid, because if a_4 is applied before a_3 it creates $\curvearrowleft_2$, triggering the NAC of a_3, which can no longer be applied. ■

3.3 Relations within Strongly Safe Graph Grammars with NACs

It is important to notice that the causal relation presented in the previous section is always concrete. This means that if an action is dependent on (resp. conflicting with) another one, it happens because one of them creates (resp. deletes) at least one of the concrete elements necessary for the other to be applied (resp. prevented of being applied).

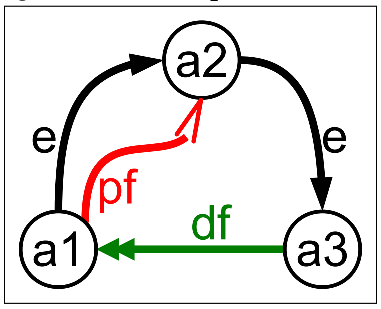
Moreover, the causal relation must always be respected whenever we try to find a total order in which all the actions of a strongly safe grammar are applicable. Nonetheless, for grammars equipped with NACs it is necessary to include the conflicts and dependencies created by NACs in this relation.

This inclusion of conflicts and dependencies induced by NACs gives rise to a new problem: we can not just add those conflicts and dependencies directly into the causal relation because they are *potential* instead of concrete. They may or may not happen depending on which specific total ordering of application (among all possibilities) was performed. Therefore, we need a way to identify under which conditions these potential conflicts (resp. dependencies) appear, in order to know which among the possible total orderings also respect the restrictions imposed by NACs.

Example 3.22 (Interaction between causal relation and NACs)**.** Let a_1, a_2, a_3 be three actions of the same strongly safe grammar. Suppose that a_1 creates elements used by a_2 and a_2 creates elements used by a_3, therefore by the causal relation we know that $a_1 \leq_c a_2 \leq_c a_3$. Now suppose that when a_2 is applied, it creates an element that would be forbidden by a NAC of a_1 and also that a_3 deletes this element. Following the classical notions of dependency and conflict of graph grammars with NACs, as shown in definitions 2.16 and 2.18, a_2 would cause a (potential) produce-forbid conflict on a_1, thus a_1 should be applied before a_2 ($a_1 <_{pf} a_2$). Likewise, a_1 would be (potentially) dependent by delete-forbid on a_3, consequently a_3 should be applied before a_1 ($a_3 <_{df} a_1$). By adding these produce-forbid and delete-forbid directly into the causal relation we would have the situation depicted in Figure 3.10. It is easy to see that the resulting relation is not a partial order, therefore there can not be a total ordering for this set of actions.

Notwithstanding, by analysing the causal relation we can affirm that this configuration ensures that neither the conflict nor the dependency exist in any concrete execution of this grammar. For the conflict, this happens because a_2 can only happen after a_1, thus the element forbidden by the NAC of a_1 can only exist after a_1 itself was already applied. Likewise, the dependency that is identified because a_3 deletes the element that would

Figure 3.10: Graph of relations



be forbidden by the NAC of a_1 does not exist either, because a_1 was executed before its triggering element even existed. ■

Definition 3.23 (Conflict and Dependency Characterization)**.** Given a strongly safe graph grammar $GG = \left(C^T, I^{C^T}, P\right)$, every conflict (resp. dependency) induced by NACs between two distinct actions $a_i, a_j \in P$ is said to be *potential* (as we initially do not know whether this particular situation occurs during an execution of the underlying grammar).

Given a potential conflict (resp. dependency) situation in a strongly safe grammar, we say that it is:

- *concrete*, if it always occur in any execution of the grammar;
- *abstract*, if it occurs only in some of the executions;
- *non-existent*, if it is never possible for it to occur in any execution of the grammar.

□

Definition 3.24 (Abstract Conflicts and Dependencies)**.** Given a strongly safe graph grammar $GG = \left(C^T, I^{C^T}, P\right)$, an abstract conflict/dependency is a tuple of distinct actions $t = (a_i, a_j, a_k)$. In any total ordering $<_t$ of actions in P, t restricts the application of a_i such that either $a_i <_t a_j$ or $a_k <_t a_i$.

□

The definition of abstract conflicts and dependencies translates to "the action a_i can only be applied either before a_j or after a_k, but never in between their applications". This situation models the following configuration of actions: a_j creates an element x, which triggers a NAC of a_i, while a_k deletes this same element. Therefore, a_i can not be applied while x exists.

We already know that at least the causal relation must be a partial order in order to be possible to apply all the actions of a grammar. Nevertheless, the following problem remains to be solved:

Given a strongly safe grammar $GG = \left(C^T, I^{C^T}, P\right)$ with at least two actions $a_1, a_2 \in P$ in a potential delete-forbid or produce-forbid situation, under which circumstances this dependency or conflict exists and must be considered in the actions application ordering?

In the following, we categorize these conflicts and dependencies using their triggering elements and pertinence of these elements into the causal relation in order to address this problem.

Definition 3.25 (Delete-Forbid Relation in Strongly Safe Graph Grammars)**.** Let $GG = \left(C^T, I^{C^T}, P\right)$ be a strongly-safe graph grammar, where P is a set of actions with incremental, non-trivially triggered NACs only.

$$\begin{tikzcd}
\mathbf{a_1} & & N_1^{-1} & & N_2 \arrow[llll, dotted, bend right, "q"'] \arrow[ddllll, dotted, "q"'] & & \mathbf{a_2} \\
L_1 \arrow[d] & K_1 \arrow[l] \arrow[r] \arrow[d] & R_1 \arrow[u, "n_1"] \arrow[dr, "post_1"'] & & L_2 \arrow[u, "n_2"'] \arrow[dl, "pre_2"] \arrow[dlll, dotted, "h_{21}"'] & K_2 \arrow[l] \arrow[r] \arrow[d] & R_2 \\
H_1 & D_1 \arrow[l, "e_1"] \arrow[rr, "d_1"'] & & C^T_{|R_1 L_2} & & D_2 \arrow[ll] &
\end{tikzcd}$$

Let $a_1, a_2 \in P$ be in a potential delete-forbid dependency according to the diagram above, where a_1 deletes from graph H_1 an element $x \in N(C^T) \cup E(C^T)$ which is the triggering element of a NAC N_2 of a_2 for the extended match $e_1 \circ h_{21}$. This delete-forbid dependency is:

- *concrete*: iff $(x \leq_c a_2) \vee (x \in I^{C^T})$.
- *abstract*: iff $(\exists a_3 \in P \mid x \in R_3 - K_3) \wedge (x \nleq_c a_2) \wedge (a_2 \nleq_c x)$.
- *non existent*: otherwise.

If the above dependency is characterized as concrete, we have that $a_1 <_{df} a_2$. If it is abstract, $t_{df} = (a_2, a_3, a_1)$ is is the set of abstract conflicts and dependencies of the grammar GG. If it is non existent, the dependency is simply discarded in further analysis, since the orderings of execution are not affected by it.

$\square$

According to our classification, we have that if a delete-forbid dependency is concrete, i.e. $a_i <_{df} a_j$, then we can only consider total orderings of $\leq_c$ where $a_i < a_j$. For the abstract case, we can only consider total orderings of $\leq_c$ where $a_i < a_j$ or $a_k < a_j$.

If it is non existent there is no need to impose any new restrictions over the ordering of $\leq_c$. Thus, we need to demonstrate that:

Theorem 3.26. *For each potential* delete-forbid *in a strongly safe graph grammar, definition 3.25 correctly classifies these dependencies.*

Proof. Given actions $a_1, a_2 \in P$ in a potential *delete-forbid* where a_1 deletes an element $x \in C^T$ which is also the triggering element of a NAC in a_2. The following situations are possible:

Triggering element is present on the initial graph:

Let x be an element which is not created by any action in P, meaning that x is present on the initial graph: $x \in I^{C^T}$. In such configuration, the delete-forbid is *concrete*, given that x exists before the application of a_2 (or any other action), preventing the application of the action until it gets deleted. Therefore, we have $a_1 <_{df} a_2$.

Triggering element is related to the action:

If $a_2 \leq_c x$, it means that x was either created by a_2 or by another action that must to occur after a_2. In this configuration the delete-forbid dependency is *non existent* as the element x can not exist to trigger N_2 before a_2 was already applied.

On the other hand, if $x \leq_c a_2$, it means that x must exist at some moment before a_2 is applied. In this configuration, we have that x must be deleted in order for a_2 to be applied. Since a_1 is the only action that deletes x (otherwise the underlying grammar would not be a strongly safe grammar) this delete-forbid is *concrete* and we have that $a_1 <_{df} a_2$.

Triggering element is not related to the action:

Let x be not related to a_2, but created to a third action $a_3 \in P$ (notice that in this configuration we have that $a_3 <_c a_1$).

If a_3 is not related to a_2, in which case we have that a_1 and a_2 *would be* in a concrete delete-forbid dependency if a_3 were applied before a_2. On the other hand, the same dependency *would be* non existent if a_3 were applied after a_2. This situation depicts an *abstract* produce-forbid dependency as we have that either $a_1 <_{df} a_2$ or $a_2 < a_3$ are possible. Therefore (a_2, a_3, a_1) is an abstract dependency.

Finally, if a_3 is related to a_2, then a_2 is related to x, as a_3 creates x, which corresponds to our second case.

□

Definition 3.27 (Produce-Forbid Relation in Strongly Safe Graph Grammars)**.** Given $GG = \left(C^T, I^{C^T}, P\right)$ a strongly safe graph grammar, where P is a set of actions with incremental, non-trivially triggered NACs only.

$$\begin{tikzcd}
\mathbf{a_1} & & N_1 & & N_2 \arrow[ddllll, dotted, bend right, "q"'] & & \mathbf{a_2} \\
R_1 \arrow[d] & K_1 \arrow[l] \arrow[r] \arrow[d] & L_1 \arrow[u, "n_1"] \arrow[dr, "pre_1"] & & L_2 \arrow[u, "n_2"'] \arrow[dl, "pre_2"'] \arrow[dlll, dotted, bend right, "h_{21}"'] & K_2 \arrow[l] \arrow[r] \arrow[d] & R_2 \\
H_1 & D_1 \arrow[l, "d_1"] \arrow[rr, "e_1"'] & & C^T_{|L_1L_2} & & D_2 \arrow[ll] &
\end{tikzcd}$$

Let $a_1, a_2 \in P$, be in a potential produce-forbid conflict according to the diagram above, where a_1 creates on H_1 an element $x \in N(C^T) \cup E(C^T)$ which is the triggering element of a NAC N_2 of a_2 for the extended match $d_1 \circ h_{21}$. This produce-forbid conflict is:

- *concrete*: iff $(\nexists a_3 \in P \mid x \in L_3 - K_3) \wedge (x \leq_c a_2 \vee a_2 \nleq_c x)$;
- *abstract*: iff $(\exists a_3 \in P \mid x \in L_3 - K_3) \wedge (a_3 \nleq_c a_2) \wedge (a_2 \nleq_c a_3)$;
- *non existent*: otherwise.

□

Similarly to the dependency situation, the produce-forbid conflict can have different meanings according to the causal relation of its grammar. Nonetheless, once we are dealing with strongly safe grammars and this element is created by one of the actions involved in the conflict, we do not have to worry about the initial graph. Thus, we need to demonstrate that:

Theorem 3.28. *For each potential produce-forbid in a strongly safe graph grammar, definition 3.27 correctly classifies these conflicts.*

Proof. Given actions $a_1, a_2 \in P$ in a potential *produce-forbid* conflict and an element $x \in C^T$ which is the triggering element of the NAC in this conflict. The following situations are possible:

Triggering element is related to the action:

Let $a_2 \leq_c x$, which means that x was either created by a_2 or by another action that must occur after a_2 has been applied. In such a configuration, the produce-forbid conflict is non existent as the element x can not exist to trigger N_2 before the application of a_2.

Now let $x \leq_c a_2$, which means that x existed before a_2 was applied, which leads to two possible sub cases:

- Assume that there is no other action a_3 which deletes x. In this situation, we have both that the triggering element x exists before the application of a_2 and that x is never deleted. Therefore, the produce-forbid relation between a_1 and a_2 is *concrete* and we have that the $a_2 <_{pf} a_1$.
- Assume that there exists a third action $a_3 \in P$ which deletes x. This means that, even though a_1 and a_2 are in a concrete produce-forbid conflict, this conflict can be "annulated" by the application of a_3, which must then happen before a_2. Therefore, this conflict is non-existent if $a_3 <_c a_2$, otherwise it is concrete and we have that $a_2 <_{pf} a_1$.

Triggering element is not related to the action:
Let a_2 be not related to x in the causal relation.

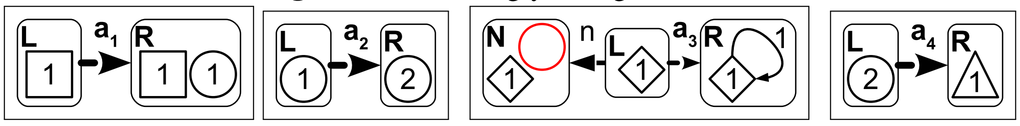
Suppose that there is no other action a_3 which deletes x, then we know for a fact that once a_1 has been applied and x has been created it is no longer possible to apply a_2. Therefore, this produce-forbid relation is *concrete* and we have that $a_2 <_{pf} a_1$.

On the other hand, suppose that there is an action a_3 which deletes x (and thus $a_1 <_c a_3$) and a_3 not related to a_2 by the causal relation. In this configuration, we have an *abstract* conflict, because a_2 can not be applied after a_1 has been applied, unless a_3 is also applied before a_2, disabling the produce-forbid conflict. This situation depicts an abstract delete-forbid conflict as we have that either $a_2 <_{pf} a_1$ or $a_3 < a_2$ are possible. Therefore (a_2, a_1, a_3) is an abstract conflict.

$\square$

Example 3.29 (Conditional Relations)**.** Consider the strongly safe grammar show in Figure 3.11 (the core, typed and initial graphs were omitted).

Figure 3.11: Strongly safe grammar GG3



(a) Action a_1 **(b)** Action a_2 **(c)** Action a_3 **(d)** Action a_4

The causal relation of this grammar is: $a_1 \leq_c a_2, a_2 \leq_c a_4, a_1 \leq_c a_4, a_3 \leq_c a_3$. Without the NACs, any sequentialization where the order $a_1 < a_2 < a_4$ is maintained would be valid, such as $[a_1, a_2, a_3, a_4]$, $[a_1, a_3, a_2, a_4]$, $[a_3, a_1, a_2, a_4]$ and $[a_1, a_2, a_4, a_3]$.

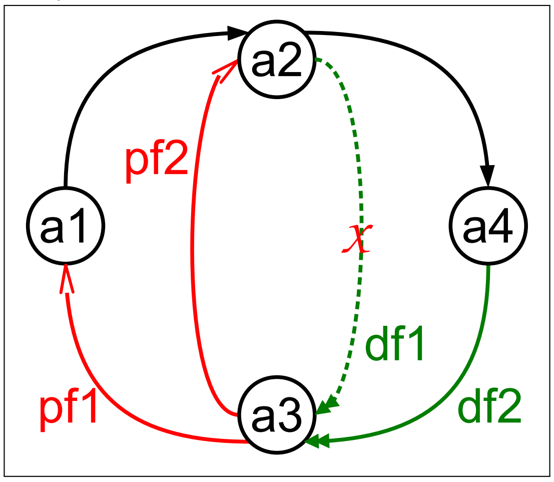
However, as this grammar have NACs, the following conditional conflicts and dependencies have been identified:

- delete-forbids: $a_4 <_{df} a_3$ caused by the deletion of $\bigcirc_2$.
- produce-forbids: $a_3 <_{pf} a_1$ caused by creation of $\bigcirc_1$, and $a_3 <_{pf} a_2$ caused by the creation of $\bigcirc_2$.

Notice that, despite of the fact that a_2 deletes $\bigcirc_1$, which triggers the NAC of a_3, $a_2 <_{df} a_3$ is not a delete-forbid dependency because a_2 also creates $\bigcirc_2$, an element that still triggers the same NAC. Therefore the transformations when searching for the dependency between a_2 to a_3 are not valid.

None of these conflicts or dependencies are *concrete*, they depend on how the actions are applied according to the causal relation to exist. This situation is summarized in Figure 3.12, where the causal relation, the conflicts and dependencies are represented (without the explicit representation of transitivity and reflexivity). At first, if we are to consider all the relations as they are calculated by the diagrams themselves, there would be no possible sequentialization for this actions, denoted by the cycle in the ordering graph.

Figure 3.12: Cycle due to conditional conflicts and dependencies



Regarding the element $\bigcirc_1$, we have an abstract produce-forbid conflict as a_3 can be applied before a_1 creates it or after a_2 deletes it. Thus it is possible to apply a_3 as long as it is not applied between a_1 and a_2. As for the element $\bigcirc_2$, we have an abstract produce-forbid conflict as a_3 can be applied before a_2 creates it or after a_4 deletes it.

In fact, we have that the actions of this particular grammar can be executed in any total ordering of its causal relation $a_1 \leq_c a_2, a_2 \leq_c a_4, a_1 \leq_c a_4, a_3 \leq_c a_3$ that also respects the *restrictions* given by the abstract dependency/conflict tuples (a_3, a_1, a_2) and (a_3, a_2, a_4). There exist two such sequentializations, namely: $[a_3, a_1, a_2, a_4]$ and $[a_1, a_2, a_4, a_3]$. ■

Remark (Abstract Dependencies and Conflicts)**.** The existence of an abstract produce-forbid conflict, triggered by an element x is always conditioned to the existence of an action which deletes x. Given an action a_1 which creates x, an action a_2 whose NAC forbids x and provided a configuration where a_1 was applied before a_2, we have that a_2 can be applied only after an action a_3 which deletes x has been applied. In general, for each abstract produce-forbid conflict $a_2 <_{pf} a_1$ caused by an element x, we have that a_2 must be successfully applied before a_1 or after an action a_j where a_i deletes x and $a_i \leq a_j$.

Analogously, for each abstract delete-forbid dependency $a_1 <_{df} a_2$ caused by an element x, we have that a_2 must be successfully applied after a_1 or before an action a_j where a_i creates x and $a_j \leq a_i$. □

There is one last situation that may arise with the addition of NACs in Occurrence Graph Grammars. Given a strongly safe graph grammar $GG = \left(C^T, I^{C^T}, P\right)$ with NACs, if there is an action $a_i \in P$ which has a NAC triggered by an element $x \in I^{C^T}$, i.e. an element that is already present in the initial graph, then there must be an action $a_j \in P$ which deletes x and a "mandatory" concrete delete-forbid dependency $a_j <_{df} a_i$, otherwise GG can not be an Occurrence Graph Grammar given that action a_i would never be applicable in any total ordering of actions.

Having the conflicts and dependencies classified as described above, we can now use these characterizations to extend the original causal relation in order to work with strongly safe graph grammars with (incremental) NACs. The main idea is that non-existent conflicts and dependencies can be discarded, while the concrete ones must be considered together with the causal relation and the abstract ones impose extra restrictions over the causal and (concrete) produce-forbid and delete-forbid relations.

Definition 3.30 (Occurence Relation)**.** Given a strongly safe grammar $GG = \left(C^T, I^{C^T}, P\right)$, let $\leq_{df}$ be the set of all its *concrete* delete-forbids pairs and $\leq_{pf}$ be the set of all its *concrete* produce-forbids pairs. Then, its occurrence relation $\leq_o$ of $(P \cup N(C^T) \cup E(C^T))$ is defined as the transitive and reflexive closure of $\leq_c \cup \leq_{df} \cup \leq_{pf}$. □

Definition 3.31 (Occurence Relation Restrictions)**.** Given a strongly safe grammar $GG = \left(C^T, I^{C^T}, P\right)$, its *occurrence relation restrictions* are the sets containing all its abstract produce-forbid conflicts and delete-forbid dependencies. □

Definition 3.32 (Occurrence Graph Grammars)**.** Let $GG = \left(C^T, I^{C^T}, P\right)$ be a strongly safe graph grammar. GG is an *occurrence graph grammar* with NACs iff:

1. its occurrence relation $\leq_o$ is a partial order such that, for any $n \in N(C^T)$, $e \in E(C^T)$ such that $n = s(e)$ or $n = t(e)$, and for any $p \in P$, we have that:
    - if $p \leq_o n$, then $p \leq_o e$
    - if $n \leq_o p$, then $e \leq_o p$;
2. Let $Min = \{x \in N(C^T) \cup E(C^T) \mid \nexists y \in N(C^T) \cup E(C^T) \bullet y \leq_o x\}$, and $G(Min) = (N, E, s, t)$ such that $N = Min \cap N(C^T)$, $E = Min \cap E(C^T)$, $s = s_{|E}$, and $t = t_{|E}$; then $I^{C^T} = G(Min)$;
3. $\forall p \in P$, p satisfies the *identification condition*;
4. $\forall x \in N(C^T) \cup E(C^T)$, x is consumed by at most one production in P, and it is created by at most one production in P.
5. $\forall p_i \in P$ with NAC n_i, if the triggering element x of n_i is such that $x \in I^{C^T}$, then there must exist another action $p_j \in P$ which deletes x and $p_j <_{df} p_i$ must be a concrete delete-forbid dependency.
6. there is at least one total ordering of the actions $a_1, \ldots, a_n \in P$ that respects the occurrence relation restrictions.

□

As was said in the beginning of this chapter, the idea is that an occurrence graph grammar is a suitable way to describe the semantics of a graph grammar, in the sense that it represents both all possible states and changes of states while also being a graph grammar itself.

Our notion of Occurrence Graph Grammar is an extension of that of (CORRADINI; MONTANARI; ROSSI, 1996). The first four conditions of our definition correspond to those of the Definition 19 in their work, which assure that it is possible to find a sequence of actions that respects the order of creation and deletion of elements given by the causal relation, while extending it to also respect the order imposed by concrete produce-forbids and delete-forbids of the grammar. In other words, it assures that there is no cycle of conflicts and dependencies that can prevent the application of the grammar.

The fifth condition works over a specific triggering of NACs which may not be captured by the delete-forbid and produce-forbid relations, which happens when the triggering element of the NAC of an action already exists in the initial graph. If there is an action whose application may be prevented by the existence of an element in the initial graph, then there must be another action responsible for the deletion of that same element which in turn must always be executed before the action with the triggered NAC.

The sixth condition deals with the conflicts and dependencies induced by NACs that do not participate in the concrete relations, thus generating *intervals* of actions inside of which specific actions can not be applied. If it is possible to find a total ordering of actions regarding the occurrence relation which also obeys the set of restrictions, then it is possible to execute the underlying single typed grammar.

Definition 3.33 (Derivation of an Occurrence Graph Grammar with NACs)**.** Given an $OGG = \left(C^T, I^{C^T}, A\right)$ and a total order $s = a_1, a_2, \ldots, a_n$ of all actions in A which respects the occurrence relation $\leq_o$ and the occurrence relation restrictions of OGG, the derivation of OGG regarding s is given by the diagram below, where each individual transformation $t_i : H_{i-1} \xRightarrow{a_i, pre_{a_i}} H_i$ is concrete over the elements of the core graph, as in Definition 3.16.

$$\begin{tikzcd}
L_1 \arrow[d, "pre_{a_1}"'] & K_1 \arrow[l] \arrow[r] \arrow[d] & R_1 \arrow[dr, "post_{a_1}"] & & \dots \arrow[dl, "pre_{a_2}"'] \arrow[dr, "post_{a_{n-1}}"] & & L_n \arrow[dl, "pre_{a_n}"'] & K_n \arrow[l] \arrow[r] \arrow[d] & R_n \arrow[d, "post_{a_n}"] \\
I^{C^T} & D_1 \arrow[l] \arrow[rr] & & H_1 & \dots \arrow[l] \arrow[r] & H_{n-1} & & D_n \arrow[ll] \arrow[r] & F^{C^T}
\end{tikzcd}$$

A derivation for an OGG starts at the initial graph I^{C^T} and terminates at the final graph F^{C^T} of OGG, and every intermediary graph is a subgraph of the core graph, i.e. $D_i \hookrightarrow C^T$, $H_i \hookrightarrow C^T$. □

Thus, an occurrence graph grammar represents a set of computations (derivations), where each possible total order of actions that is compatible with the occurrence relation and restrictions leads to one specific derivation. The fact that each H_i is indeed a graph and that the squares are pushouts was proven in the original definition of occurrence graph grammars of (CORRADINI; MONTANARI; ROSSI, 1996) (our definition just poses additional restrictions on possible total orders of actions that may lead to derivations). Of course, we have to guarantee, additionally, that each pre_{a_i} satisfies the corresponding NACs. This is shown by the following theorem that also states that any derivation described by an occurrence graph grammar is indeed a derivation of its underlying graph grammar.

Theorem 3.34. *Given a graph grammar $GG = (T, I^T, P)$ and an Occurrence Graph Grammar with NACs $OGG = \left(C^T, I^{C^T}, A\right)$. If we have that:*

- *$\forall a \in A$, its underlying rule $p(a) \in P$*

- $type_C \circ t^T(I^{C^T}) \equiv I^T$

Then any derivation represented by OGG is a derivation of GG.

Proof. Given a Graph Grammar $GG = (T, I^T, P)$ and an Occurrence Graph Grammar $OGG = \left(C^T, I^{C^T}, A\right)$ built from GG. It is possible to construct a derivation of OGG for any total ordering of A which satisfies the occurrence relation and the occurrence relation restrictions of OGG.

Let $d = I^{C^T} \xRightarrow{a_1, pre_1} H_1^{C^T} \xRightarrow{a_2, pre_2} H_2^{C^T} \ldots H_{n-1}^{C^T} \xRightarrow{a_n, pre_n} F^{C^T}$ be one of the possible derivations of OGG. Given the type morphism $t : C \longrightarrow T$ of the core graph and the type morphism $dt_i : H_i \longrightarrow C^T$ of a C^T-double typed graph, it is possible to compose them in a (simple) type morphism $t \circ dt_i : H_i \longrightarrow T$. Using the resulting family of (simple) type morphisms obtained by such composition, it is possible to create a diagram $d' = I^T \xRightarrow{p_1, m_1} H_1^T \xRightarrow{p_2, m_2} H_2^T \ldots H_{n-1}^T \xRightarrow{p_n, m_n} H_n^T$ which has the format of a derivation of GG.

If the graph grammar GG does not have NACs, the occurrence graph grammar OGG also does not have NACs, and it holds true that d' is a derivation of GG, since it was constructed as given by the original definition of occurrence graph grammars in (CORRADINI; MONTANARI; ROSSI, 1996).

Thus, we need to show that with the addition of (incremental) NACs in GG, our extension for the construction of occurrence graph grammars with NACs still guarantees that the derivation d' obtained from d is a valid derivation of GG.

Let $p_i = L_i^T \longleftarrow K_i^T \longrightarrow R_i^T \in P$ be a rule with an incremental NAC $n_i : L_i^T \longrightarrow N_i^T$ which participates in the (candidate) derivation d'. Assume that n_i is triggered by an elementi x around the match $m_i : L_i^T \longrightarrow H_{i-1}^T$ in the derivation. There are four possible cases that could lead to this configuration:

1. $x \in I^T$ and there is no rule which deletes it.

2. $x \in I^T$, but the rule p_k which deletes it was applied after p_i, i.e. $p_i < p_k$.

3. x was created by a rule p_j applied before p_i and there is no other rule which deletes x, i.e $p_j < p_i$.

4. x was created by a rule p_j applied before p_i and the rule p_k which deletes x was applied after p_i, i.e. $p_j < p_i < p_k$.

The construction of the occurrence graph grammar would identify these problems as listed below:

In the first case, the condition which requires the existence of another action p_j responsible for the deletion of x with a concrete delete-forbid dependency $p_j <_{df} p_i$ between them would be violated, therefore OGG would not be a valid Occurrence Graph Grammar.

In the second case, there would be a concrete delete-forbid dependency between actions $a_k <_{df} a_i$, therefore this dependency would be present on the occurrence relation and any total ordering where $a_i < a_k$ would not satisfy the OGG constraints.

In the third case, there would be a concrete produce-forbid conflict between actions $a_i <_{pf} a_j$, this conflict would thus be present on the occurrence relation and any total ordering where $a_j < a_k$ would not satisfy the OGG constraints.

In the fourth case, there would be an abstract delete-forbid dependency and/or an abstract produce-forbid conflict denoted by the tuple (a_i, a_j, a_k), which would be included in the set of occurrence relation restrictions and forbid any total ordering of actions where $a_j < a_i < a_k$.

Therefore, given that either of the situations described above would result in a total ordering of actions in A which does not satisfy the constraints of an Occurrence Graph Grammar with NACs $OGG = \left(C^T, I^{C^T}, A\right)$, d' not being a derivation of GG would contradict the fact the d is a derivation of OGG, because it would be based on a total ordering which does not satisfy the constraints of its underlying Occurrence Graph Grammar.

□

Theorem 3.35. *Given a Graph Grammar with NACs $GG = (T, I^T, P)$, any derivation of GG generates an Occurrence Graph Grammar with NACs $OGG = \left(C^T, I^{C^T}, A\right)$.*

Proof. Let $d = I^T \xRightarrow{p_1, m_1} H_1 \xRightarrow{p_2, m_2} H_2 \dots H_{n-1} \xRightarrow{p_n, m_n} H_n$ be a derivation of GG. We have that d provides a total order of rule applications or actions. Therefore, if we calculate the colimit of the diagram defined by derivation d, as defined in (CORRADINI; MONTANARI; ROSSI, 1996), the result is a graph C^T containing all the elements ever used in that derivation. This colimit also identifies the common elements among the rules and initial (and final) graph. Thus, the colimit object C^T is also a core graph.

Given the rules $p_1, p_2, \dots, p_n$ used in the derivation, if we consider their morphisms to C^T as typing morphisms, we arive at a set of double-typed graph rules $A = (a_1, a_2, \dots, a_n)$ over C^T. Similarly, if we consider the morphism from the initial graph I^T to the colimit as a typing morphism, we arive at a double-typed graph I^{C^T}. Thus, the grammar defined by $OGG = (C^T, I^{C^T}, A)$ is in fact a strongly safe graph grammar, therefore a candidate to be an occurrence graph grammar with NACs.

We now have to prove that (1) the occurrence relation $\leq_o$ underlying OGG is a partial order and that (2) there is at least one total ordering of $\leq_o$ which also respects the occurrence relation restrictions generated by the analysis of conflicts and dependencies induced by NACs in OGG.

If we use the same order of actions in A given by the application of their correspondent graph rules from P in the derivation d, we have a total order $<_o$ which allows the application of all actions in A, thus $\leq_o$ must be a partial order.

As for the occurrence relation restrictions, we have at least one total order which satisfies them, also given the same order of rules application in d. If this order did not satisfy these restrictions, then there would be at least one triggered NAC disabling a transformation in d, therefore d would not be a valid derivation. $\square$

Thus, we conclude that the set of occurrence graph grammars that can be built from the derivations of a graph grammar describe exactly all derivations (computations) of this grammar and can be regarded as its semantics.

Definition 3.36 (Semantics of a Graph Grammar)**.** Given a Graph Grammar $GG = (TG, I, P)$ the semantics of GG is described by the set of $OGGs$ corresponding to its derivations. $\square$

4 VERIGRAPH TOOL OVERVIEW

Verigraph (BEZERRA et al., 2017) is a new tool for simulating and analizing graph grammars implemented in Haskell[1], a purely functional programming language. The tool is being developed by the Verites group[2] with two particular aims. The first one is to build a software tool that serves as an implementation of standard constructions and analysis for graph grammars, while also being as closely related to the theory as possible. The second, to provide a framework for exploring new ideas and techniques in graph grammars and other category theory related topics (BEZERRA; RIBEIRO, 2016; COSTA et al., 2016; COSTA; MACHADO; RIBEIRO, 2016; BECKER, 2014).

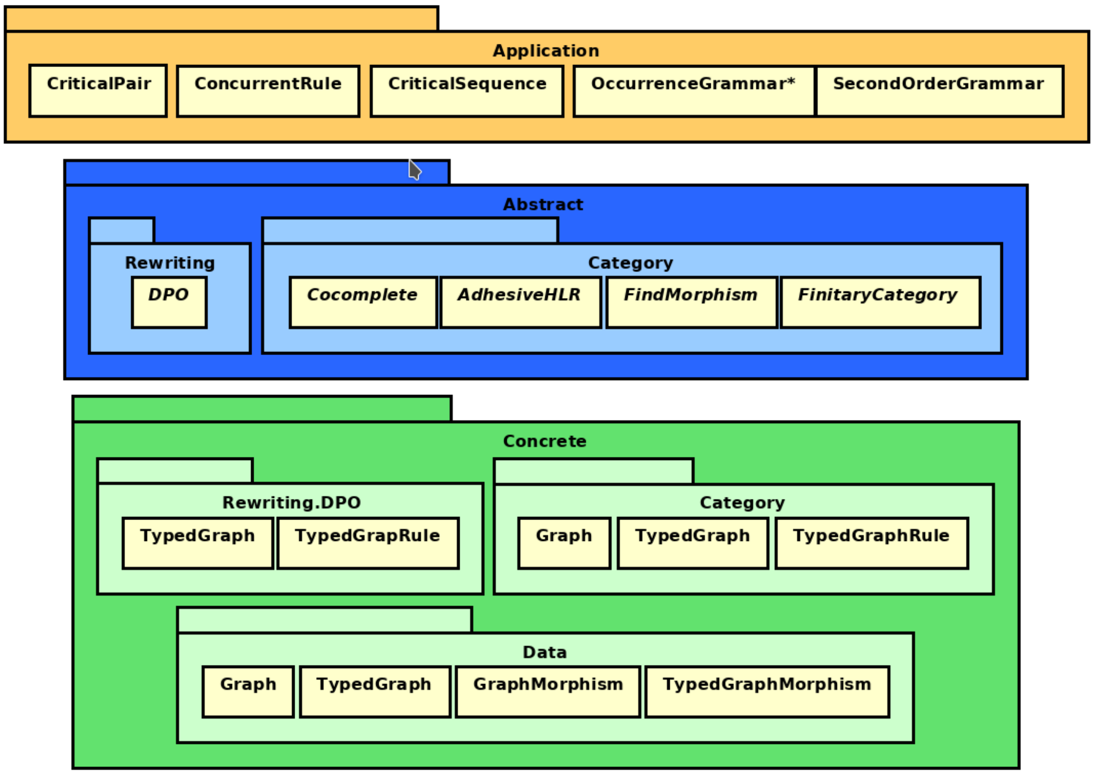
Regarding category theory, Verigraph implements important basic constructions such as coequalizers, coproducts, colimits, pushout complements, initial pushouts, pullbacks, negative application conditions, constraints, among others. The implementation of these constructions follows a very similar approach to the one used in (RYDEHEARD; BURSTALL, 1988), where categorial concepts are implemented as types in the ML programming language and constructive proofs of theorems in category theory are built as ML programs.

The implemented categorial constructions are used as a basis to implement several graph grammar analyses, such as critical pair analysis (LAMBERS; EHRIG; OREJAS, 2006), state space generation and model checking (BECKER, 2014), concurrent rules generation (BEZERRA; RIBEIRO, 2016) and higher-order graph transformations (MACHADO; RIBEIRO; HECKEL, 2015). They were also used to implement the construction of occurrence graph grammars with NACs explained in depth in Chapter 3.

The analysis algorithms are implemented in a generic functional style, having the advantage of being closely related to the formal definitions, thus making it easier to reason about them and to inspect for correctness. In addition, Verigraph benefits from a layered architecture, shown on Figure 4.1, where it is easy to reuse the same analysis algorithms (top layer) for other categories different than $\mathbf{TGraph_T}$ (bottom layer), as long as they implement the contracts given by the type classes (middle layer) defined on the system. Examples of these type classes are shown on Figures 4.2 and 4.3. To implement the constructions defined in this work, it sufficed to add the module `OcurrenceGrammar` which reused the already existing categories and analysis algorithms present in other modules.

[1]The source code is available at <https://github.com/Verites/verigraph>
[2]<http://www.ufrgs.br/verites>

Figure 4.1: Verigraph architecture



There are tools for analysing graph grammars which are similar to Verigraph in some aspects, such as AGG (TAENTZER, 2000) and GROOVE (RENSINK, 2004). Recently, (DECKWERTH, 2016) introduced a Java framework for static verification of graph transformations also based in category theory. However, to our knowledge, Verigraph is the only tool that integrates static and dynamic analyses, second-order specifications and provides support for new categorial constructions and algorithms, besides being the only tool in this field implemented in a pure functional language (COSTA et al., 2016). Moreover, Verigraph is a free and open source software, available online for the community in one of the biggest platforms for software repositories currently available. In addition to it, not only its source code, but also its roadmap is public and open to suggestions and collaborations from outside the Verites group.

In the next sections of this chapter, we demonstrate basic aspects of Verigraph implementation. First we present general categorial constructions which are the basic foundations of the tool; then we provide details about the implementation of concrete objects and categories, specifically focusing on $\mathbf{TGraph_T}$; after, we present the implementation of some analysis algorithms and show how they can be reused by other categories.

4.1 Implementation of Categorial Constructions

The first basic type class in Verigraph is `Morphism`, shown in Figure 4.2, which serves as the minimal contract for any category to be implemented in the tool. Notice how the contract of this type class reflects the category definition (see Definition A.1).

Figure 4.2: Morphism Type Class

```haskell
class (Eq m) => Morphism m where
    type Obj m :: *
    compose  :: m -> m -> m
    domain   :: m -> Obj m
    codomain :: m -> Obj m
    id       :: Obj m -> m
    isMonomorphism :: m -> Bool
    isEpimorphism :: m -> Bool
    isIsomorphism :: m -> Bool
```

All the other type classes in the tool that are related to category theory are somehow defined in terms of `Morphism`. For example, the `Cocomplete` type class shown in Figure 4.3 defines some of the most basic categorial constructions used in Verigraph, such as coequalizers, coproducts and pushouts.

Notice that in the `Cocomplete` definition any category that implements the functions `calculateCoequalizer` and `calculateCoproduct` automatically has a standard implementation of the `calculatePushout` function based only on these two constructions. This is due to the fact that, whenever a category has coequalizers and coproducts, it is possible to calculate any (finite) colimit based only on these two constructions, as demonstrated in (PIERCE, 1991).

We took advantage of this result to implement not only the `calculatePushout`, but also the calculation of the `colimit` of a diagram. The later being used in the generation of occurrence graph grammars as is shown in chapter 5.

Another interesting characteristic of the `Morphism` type class is that, even though `pushouts` and `colimits` are implemented in terms of `coproducts` and `coequalizers`, the programmer can override the default implementation and provide his/her own (categorial specific) implementation. This could be useful, for example, when for a given category **C**, a particular algorithm to calculate the pushout is known to be more optimized than using the composition of basic operations.

In addition to `Morphism`, Verigraph has several other important type classes,

Figure 4.3: Cocomplete Type Class

```haskell
class (Morphism m) => Cocomplete m where
  calculateCoequalizer :: m -> m -> m
  calculateNCoequalizer :: NonEmpty m -> m
  calculateCoproduct :: Obj m -> Obj m -> (m,m)
  calculateNCoproduct :: NonEmpty (Obj m) -> [m]

  calculatePushout :: m -> m -> (m, m)
  calculatePushout f g = (f', g')
    where
      b = codomain f
      c = codomain g
      (b',c') = calculateCoproduct b c
      gc' = compose g c'
      fb' = compose f b'
      h = calculateCoequalizer fb' gc'
      g' = compose b' h
      f' = compose c' h
```

some examples are:

- `FindMorphism` for finding morphisms between objects of a category;
- `AdhesiveHLR` for operations that AdhesiveHLR categories (e.g. $\mathbf{TGraph_T}$) are guaranteed to have, such as calculating initial pushouts and pushout complements (when they exist);
- `DPO` for operations related to DPO graph rewriting approach, such as inversion of rules.

As for the concrete categories used, currently there are three specific implementations in Verigraph. Besides **Graph** and $\mathbf{TGraph_T}$, which were reviewed on chapter 2, there is also an implementation of $\mathbf{TSpan_T}$, where we have $T-$typed graph morphism spans are objects and span morphisms are arrows or, from a more concrete perspective, DPO graph rules as objects and morphisms between rules as arrows.

4.2 Implementation of Graph Grammars

The main concrete structures in Verigraph are (typed) graph grammars, which is currently the focus of the Verites group. The basic implementation begins with the `Graph` type, which consists of a list of nodes and a list of edges together with an API for graph manipulation with basic functions.

The `Graph` definition on Haskell is show on Figure 4.4. The `Graph` API is not shown, but it includes basic graph operations such as `insertNode`, `insertEdge`, `removeNode`, `removeEdge`, `incomingEdges`, `outgoinEdges`, `sourceOf`, `targetOf`, among others.

Figure 4.4: Graph implementation

```haskell
data Node a = Node
{ getNodePayload :: Maybe a
}

data Edge a = Edge
{ getSource      :: NodeId
, getTarget      :: NodeId
, getEdgePayload :: Maybe a
}

data Graph a b = Graph
{ nodeMap :: [(NodeId, Node a)]
, edgeMap :: [(EdgeId, Edge b)]
}
```

We use `Graph` to progressively build the morphisms necessary to implement the categories **Graph**, **TGraph$_\mathbf{T}$** and **TSpan$_\mathbf{T}$**. A graph morphism consists of a graph as domain, a graph as codomain and relations that map the nodes and edges in the domain graph to the ones in the codomain one. A typed graph is regarded as a simple graph morphism and a typed graph morphism consists of a typed graph as domain, a typed graph as codomain and a graph morphism relating the two of them.

Figure 4.5 shows all categories currently implemented in Verigraph based on their morphisms. Moreover, all concrete morphisms presented implement the `Morphism` type class. Figure 4.6 shows how `TypedGraphMorphism` implements `Morphism` type class in order to provide the **TGraph$_\mathbf{T}$** category.

Similar implementations were done for `GraphMorphism` and `RuleMorphism`.

4.3 Implementation of the Analysis Algorithms

The analysis algorithms are also implemented at a high level of abstraction, based on categorial definitions and their implementation as type classes. For example, the code for calculating conflicts or dependencies between two rules was first implemented for

Figure 4.5: Basic concrete morphisms of Verigraph.

```haskell
data GraphMorphism a b = GraphMorphism
{ getDomain     :: Graph a b
, getCodomain   :: Graph a b
, nodeRelation :: R.Relation G.NodeId
, edgeRelation :: R.Relation G.EdgeId
}

type TypedGraph a b = GraphMorphism a b

data TypedGraphMorphism a b = TypedGraphMorphism
{ getDomain    :: TypedGraph a b
, getCodomain :: TypedGraph a b
, mapping      :: GraphMorphism a b
}

data RuleMorphism a b = RuleMorphism
{ rmDomain          :: Production (TypedGraphMorphism a b)
, rmCodomain        :: Production (TypedGraphMorphism a b)
, mappingLeft       :: TypedGraphMorphism a b
, mappingInterface :: TypedGraphMorphism a b
, mappingRight      :: TypedGraphMorphism a b
}
```

Figure 4.6: Typed graph morphism implementing morphism type class.

```haskell
instance Morphism (TypedGraphMorphism a b) where
  type Obj (TypedGraphMorphism a b) = TypedGraph a b
  domain = getDomain
  codomain = getCodomain
  compose t1 t2 = TypedGraphMorphism (domain t1) (codomain t2) $
   ↪  compose (mapping t1) (mapping t2)
  id t = TypedGraphMorphism t t (M.id $ domain t)
  isMonomorphism = isMonomorphism . mapping
  isEpimorphism = isEpimorphism . mapping
  isIsomorphism = isIsomorphism . mapping
```

TGraph$_{\mathbf{T}}$, but since it is based on the abstraction of `DPO` type class this piece of code can be reused by any other category implementing the `DPO` contract.

Furthermore, Figure 4.7 shows a piece of code with functions responsible for testing whether an overlapping pair of two rules rises a conflict or a dependency for one of those rules. Notice how this code resembles the definitions of delete-use conflict (Definition 2.18) and produce-use dependency (Definition 2.16).

As an example of its application at other categories we have **TSpan$_{\mathbf{T}}$**, which also

Figure 4.7: Delete-Use and Produce-Use Implementation

```
-- | Rule @p1@ is in a delete-use conflict with @p2@ if @p1@
↪  deletes something that is used by @p2@. This function
↪  verifies the non existence of h21: L2 -> D1 such that d1 .
↪  h21 = m2
isDeleteUse :: (DPO m) => Production m -> (m, m) -> Bool
isDeleteUse p1 (m1,m2) = null h21
  where
    --gets only the morphism d1 from D1 to G
    (_,d1) = calculatePushoutComplement m1 (getLHS p1)
    h21 = findAllPossibleH21 m2 d1

isProduceUse :: (DPO m) => Production m -> (m, m) -> Bool
isProduceUse p1 (m1',m2) = null h21
  where
   --gets only the morphism d1 from D1 to G
   (_,e1) = calculatePushoutComplement m1' (getRHS p1)
   h21 = findAllPossibleH21 m2 e1
```

implements the `DPO` type class and benefits from the same algorithms for finding conflicts and dependencies. This also can be used for different categories based on graphs, algebras, logics and so on.

Besides basic categorial constructions and several analysis techniques for graph grammars, Verigraph was also used to implement the construction of Occurrence Graph Grammars and the relations presented in Chapter 3. This construction is presented in more detail in the following chapter.

5 CONSTRUCTION OF OCCURRENCE GRAPH GRAMMARS WITH NACS

In this chapter, we present the process of constructing Occurrence Graph Grammars with NACs (OGGs) in a systematic manner and how this process was implemented in Verigraph. We make use of some example grammars in order to illustrate the explanation. We also applied this process in more complex graph grammars: One of the grammars models the system of a restaurant, with functionalities such as login of employees, reservation and cancellation of tables, accommodation of clients, serving tables, among others. Another models the system of an e-Store, with functionalities such as browse and search catalogue, registration of clients, login, maintain shopping cart, effectuate purchase, etc. The grammars, their textual specifications (use cases), together with their generated OGGs can be found at the Verites repository for case studies[1].

Our first example is the *Mail Server Graph Grammar* presented on Chapter 2, which are reintroduced. Then we proceed to the explanation of the necessary steps to build an Occurrence Graph Grammar according to the definitions presented on Chapter 3. Our second example is a *Traffic-Light Graph Grammar*, which is introduced later on this chapter and also used to explain the process of constructing an OGG. The implementation of this process in Verigraph is also part of this work contribution, as Verigraph is the first tool in the field to implement the construction of Occurrence Graph Grammars for general Graph Grammars, even when considering OGGs without NACs. As a possible practical application of this, at the end of the chapter, we provide some insight about how OGGs can be used to generate test cases for the system they model.

Example 5.1. The grammar used here illustrates a mail server scenario for a simple e-mail application composed of four rules, which are described in the following and depicted in Figure 5.1. In order to better present the ideas of this chapter in terms of its implementation, we use the rules in the format provided by AGG (TAENTZER, 2000), which is also the format used as input and output by Verigraph. This format depicts the LHS and RHS graphs of rules, but does not explicitly show the interface graph, which can be inferred by identification numbers correlating items of LHS and RHS.

(a) *Send message:* a client writes a message which they send to a server, however there is a NAC forbidding the message of being sent if it has a piece of data attached to it.

(b) *Get data:* a piece of data is obtained from a server and attached to a message.

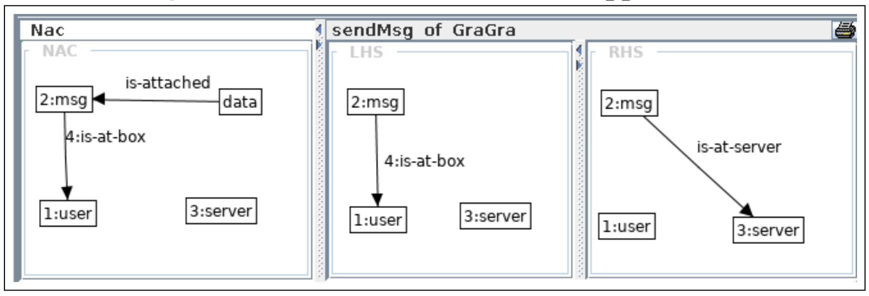
[1]https://github.com/Verites/case-studies

(c) *Receive message:* a server sends a message with attached data to a client.

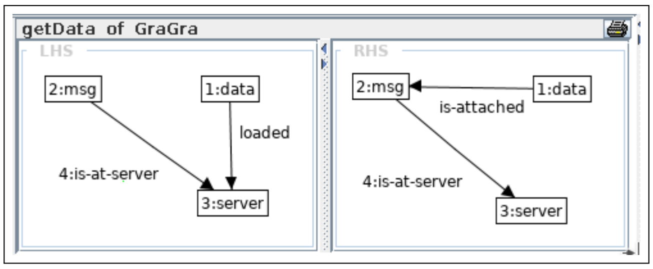
(d) *Delete message:* a client obtains a piece of data from a received message and this message is destroyed.

∎

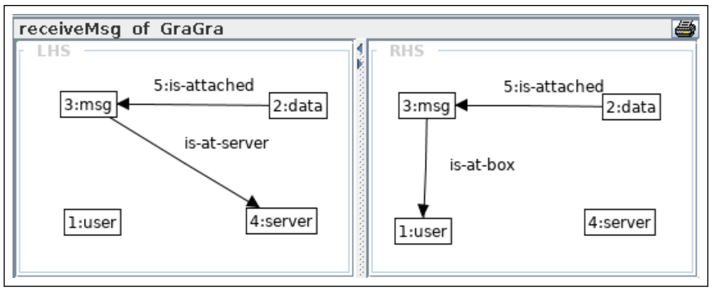
Figure 5.1: Rules for a mail server application



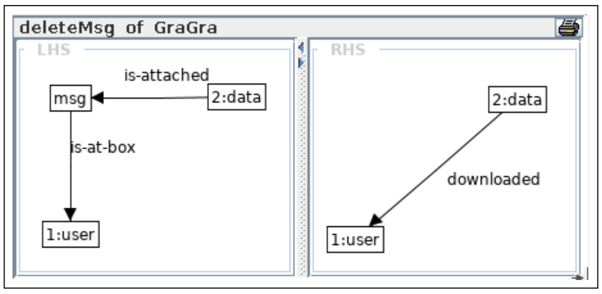
(a) Rule *send message*



(b) Rule *get data*



(c) Rule *receive message*



(d) Rule *delete message*

5.1 Selecting a Computation of GG

In this work we defined only deterministic Occurrence Graph Grammars (OGGs) with NACs, and therefore, in contrast to (RIBEIRO, 1996), where the semantics of a Graph Grammar is represented by only one (non-deterministic) Occurrence Graph Grammar, here we need a set of Occurrence Graph Grammars to represent the behaviour of a Graph Grammar. Nevertheless, this set is usually much smaller than the set of all derivations of a Graph Grammar since each OGG represents a (shift-)equivalent set of derivations, that is, a set of derivations that are equivalent with respect to switching the order of independent steps. In the following we describe how to construct one OGG, the complete behaviour of a GG would be described by the (possibly infinite) set of OGGs that can be constructed with the rules of the underlying GG. Note that, even in the case of (RIBEIRO, 1996), where the semantics was described by only one OGG, this structure may be infinite in the case the system has the possibility of a non-terminating computation.

Thus, in order to construct an Occurrence Graph Grammar OGG for a graph grammar $GG = (TG, I, P)$, we need (1) a collection[2] of rules F based on P, which represent the rules that are applied in the computation depicted by the OGG, and (2) a way of specifying how the rules in F interact among themselves. The latter is needed to define which elements are common throughout the rules. To this purpose, we use an *input-output relation*, depicting connections between the rules in F. The objects in this relation identify which elements must be the same between pairs of rules. This construction is similar to the construction of concurrent rules in AGG. The difference is that instead of building a rule, the result of our construction is an OGG, i.e., represents a computation.

Definition 5.2 (Input-Output Relation)**.** Given a collection F of rules, an *input-output relation IO* over F is a set of typed-graph morphism spans of the form $R_x \leftarrow IO_i \rightarrow L_y$ each of which connects two distinct rules $x, y \in F$. □

The IO_i object of each span works similarly to the gluing graph K of a rule, but instead of identifying elements that are the same in both left and right sides of a single rule, it identifies elements that are necessarily the same between the right and left sides of two different rules. An input-output relation for a collection of rules has an appearance similar (but not necessarily equal) to the following diagram, where there are several IO objects connecting the right-hand side of a rule with the left-hand side of another one.

[2]We use the term collection instead of set because, in this case, a rule can appear more than once since each rule in F represents the application of a rule in P.

$$\begin{tikzcd}
 & & & & IO_6 \arrow[dddllll, dashed] \arrow[dddrrrr, dashed] & & & & \\
 & IO_3 \arrow[ddl, dashed] \arrow[ddrrrr, dashed] & & & & & & IO_4 \arrow[ddllll, dashed] \arrow[ddr, dashed] & \\
 & & IO_1 \arrow[d, dashed] \arrow[dr, dashed] & & IO_5 \arrow[dll, dashed] \arrow[drr, dashed] & & IO_2 \arrow[dl, dashed] \arrow[d, dashed] & & \\
L_1 & K_1 \arrow[l] \arrow[r] & R_1 & L_2 & K_2 \arrow[l] \arrow[r] & R_2 & L_3 & K_3 \arrow[l] \arrow[r] & R_3
\end{tikzcd}$$

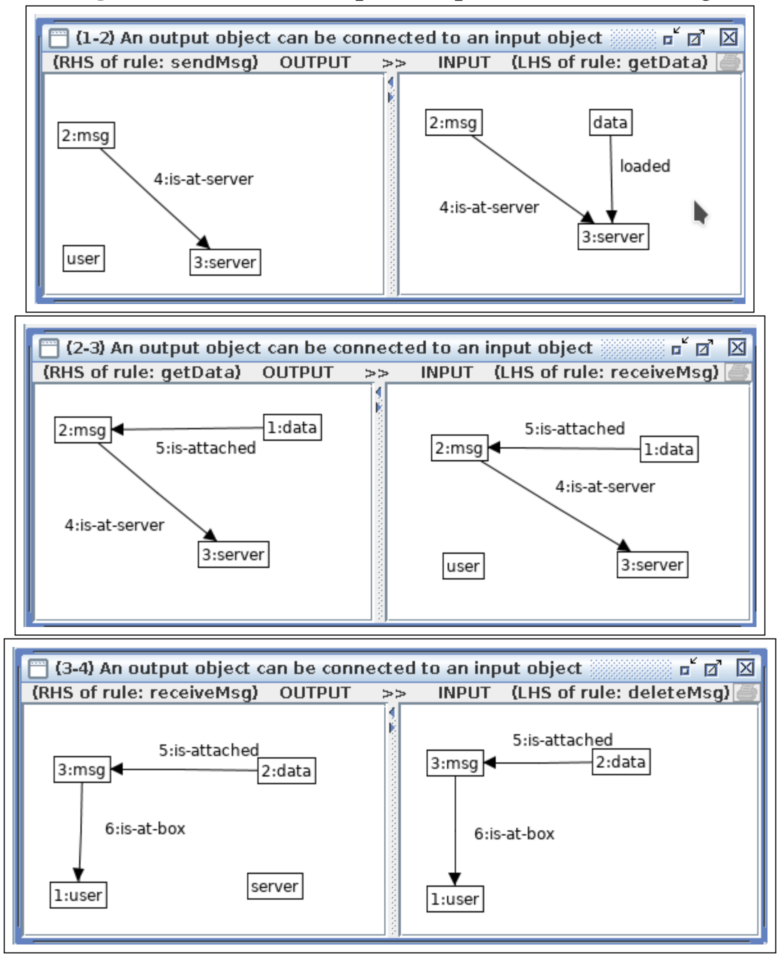
Remark. We could have also included in the *IO* relation a type of span that also connects only the left (resp. right) sides of rules together, however we would accomplish very little with this kind of span. Given that we are looking for Occurrence Graph Grammars, where the exact same element can not be deleted or created by two different rules, this particular kind of span would serve only to identify elements that are preserved by both rules, otherwise they would introduce inconsistencies, therefore preventing the creation of OGGs. □

Example 5.3 (Input-Output Relations)**.** Figure 5.2 shows one possible *IO* relation for our running example. Notice that the elements which should be the same in both rules have the same prefix number in their identification. The first *IO* object connects the rules *send message* and *get data*. In this case, we want the server, the message and the connection between them to be the same in both rules.

As a side note, the reader may notice that it is not mandatory to create an *IO* span $(R_x \leftarrow IO_i \rightarrow L_y)$ for **every** pair of rules. For example, consider that we want to analyse a scenario where the *server* node is unique. Given rules *get data* and *receive message*, we have the options of building the span that identifies the server node as $R_{sendMsg} \leftarrow IO_n \rightarrow L_{receiveMsg}$ or $R_{receiveMsg} \leftarrow IO_m \rightarrow L_{sendMsg}$, or even combining the two of them. Any option would result in the same final effect: both rules use the same *server*. ■

Currently, the generation of the *input-output relation* is a manual step in our strategy, therefore the analyst needs to decide how to better implement his/her own *IO* relation. This has not shown to be a problem, as in our analysis we usually wanted to restrict the number of possible combinations to more realistic cases. However, its fully automation has been scheduled as future work.

Figure 5.2: A basic Input-Output relation building


(1-2) An output object can be connected to an input object
(RHS of rule: sendMsg) OUTPUT >> INPUT (LHS of rule: getData)
2:msg
4:is-at-server
user
3:server
2:msg
data
loaded
4:is-at-server
3:server
(2-3) An output object can be connected to an input object
(RHS of rule: getData) OUTPUT >> INPUT (LHS of rule: receiveMsg)
2:msg
1:data
5:is-attached
4:is-at-server
3:server
5:is-attached
2:msg
1:data
4:is-at-server
user
3:server
(3-4) An output object can be connected to an input object
(RHS of rule: receiveMsg) OUTPUT >> INPUT (LHS of rule: deleteMsg)
5:is-attached
3:msg
2:data
6:is-at-box
1:user
server
5:is-attached
3:msg
2:data
6:is-at-box
1:user

5.2 Constructing the OGG

Having GG, the graph grammar model of the system; F, a collection of rules; and IO, an *input-output relation*, we proceed to the construction of an occurrence graph grammar for GG accomplished by means of an amalgamation of F over its input-output relation IO. This amalgamation is later used in the construction of a doubly-typed graph grammar, which we then check to verify whether it satisfies the conditions to be an occurrence grammar. The construction steps are specified in Definition 5.4, which is an adaptation of the construction of an occurrence graph grammar without NACs from (CORRADINI; MONTANARI; ROSSI, 1996).

Definition 5.4 (Deterministic Occurrence Graph Grammar Construction)**.** Given a grammar $GG = (TG, I, P)$, F a collection of rules from P, and IO and input-output relation over the rules in F:

1. calculate the amalgamation (colimit) Occ of the rules in F with respect to IO as presented in the following diagram, where all squares commute.

$$\begin{tikzcd}
 & & & & IO_6 \arrow[dddllll] \arrow[dddrrrr] & & & & \\
 & IO_3 \arrow[ddl] \arrow[ddrrrr] & & & & & & IO_4 \arrow[ddllll] \arrow[ddr] & \\
 & & IO_1 \arrow[d] \arrow[dr] & & IO_5 \arrow[dll] \arrow[drr] & & IO_2 \arrow[dl] \arrow[d] & & \\
L_1 \arrow[drrrr] & K_1 \arrow[l] \arrow[r] & R_1 \arrow[drr] & L_2 \arrow[dr] & K_2 \arrow[l] \arrow[r] & R_2 \arrow[dl] & L_3 \arrow[dll] & K_3 \arrow[l] \arrow[r] & R_3 \arrow[dllll] \\
 & & & & Occ & & & &
\end{tikzcd}$$

2. *retype* the rules in F over Occ: use each morphism found from each L_i, K_i, R_i to Occ as their respective new typing morphism. This step generates a set F' of doubly-typed graph rules, given that Occ is a TG-typed graph itself.

3. calculate the causal relation $\leq_c$ of the doubly-typed graph rules in F' and verify whether it is a partial order.

4. generate the initial and final graphs I and J by respectively deleting from Occ all elements ever created and deleted by the rules in F'.

5. calculate and categorize the produce-forbid conflicts and delete-forbid dependencies according to definitions 3.25 and 3.27.

   (a) use the concrete conflicts and dependencies to extend the causal relation in order to obtain the occurrence relation $\leq_o$.

   (b) use the abstract conflicts and dependencies to generate the set R of restrictions over the ordering of rules applicability.

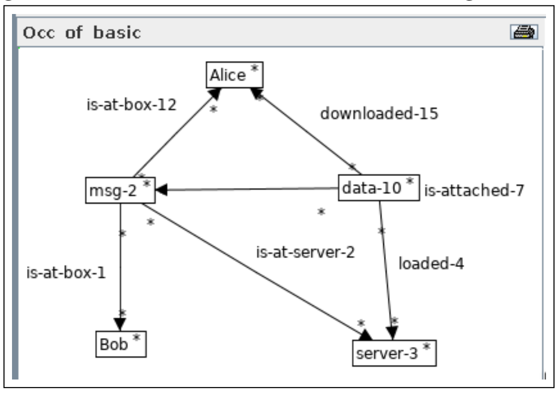
6. find one or more total orderings of rules in F' which respects both the occurrence relation and the occurrence restrictions.

□

If all the steps in such a construction can be successfully executed, specially steps 4 and 6, we have that $OGG = (Occ, I, F')$ is not only a doubly-typed graph grammar, but also a deterministic occurrence graph grammar.

The first step of the construction, the amalgamation of rules in F w.r.t. IO, is responsible for "gluing" the graphs of all rules in one typed graph Occ, while identifying the items that are meant to be the same throughout the grammar execution. This can be regarded as a mapping which turns generic elements such as *a user* into concrete elements such as the user named *Bob*. It also discriminates these elements, for instance: the user *Bob* is different from the user *Alice*. Therefore, at the end of this step, Occ contains all concrete elements ever to be created, preserved or deleted by any of the rules in the collection F. Figure 5.3 shows the amalgamation of an F containing one copy of each rule in our example grammar w.r.t. the IO relation depicted on Figure 5.2.

Figure 5.3: Amalgamation (colimit) of rules according to the basic IO relation.

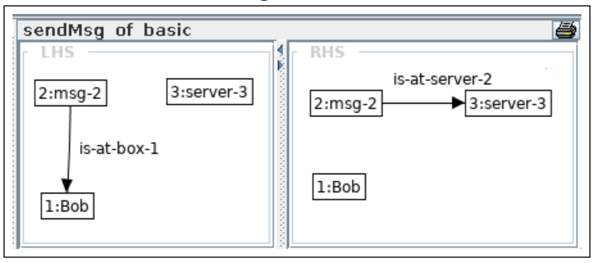


The retyping step is responsible for generating actions: "new" graph rules which, rather than being generic descriptions of system transformations, represent concrete executions of the original rules over a given context. Therefore, a rule which describes the process where a user receives from the server a message sent by another[3] user becomes a concrete action where *Alice* receives *the message* sent by *Bob*. For each graph rule $p_i^{TG} = (L_i^{TG} \leftarrow K_i^{TG} \rightarrow R_i^{TG}) \in F$, we generate a new action $q_i^{Occ} = (L_i^{Occ} \leftarrow K_i^{Occ} \rightarrow R_i^{Occ}) \in F'$, where the typing morphisms are those from the original rules in F to the colimit graph Occ. Since Occ is the type graph of the new rules and, at the same time, a typed graph over TG, the actions are now doubly-typed rules over Occ^{TG^T}. Figure 5.4 shows the actions generated for our running example[4].

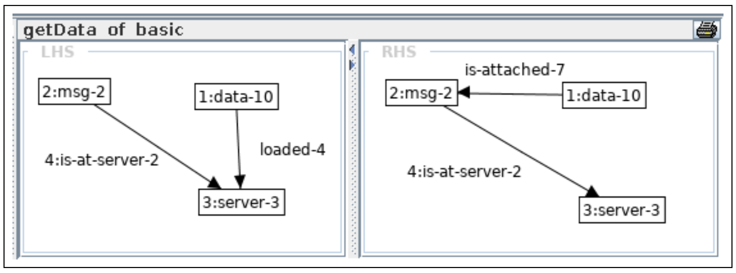
[3]Notice that the original graph grammar does not specify that the sender must be different from the receiver, therefore we would be able to choose a computation where they are the same user.

[4]Since we use AGG as our form of output visualization and it was not intended to provide support for doubly-typed graph grammars, the NACs of our actions can not be properly seen, however they are maintained and used in Verigraph internal format.

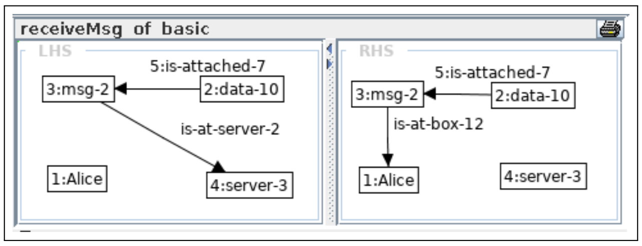
Figure 5.4: The set of actions generated w.r.t. our basic *IO* relation



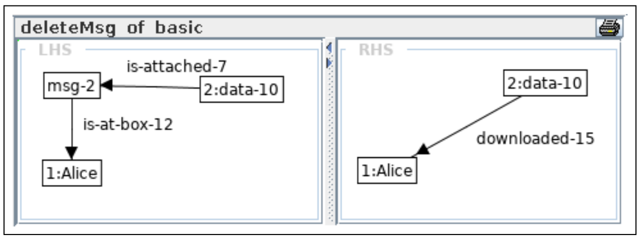
(a) Rule *send message*



(b) Rule *get data*



(c) Rule *receive message*

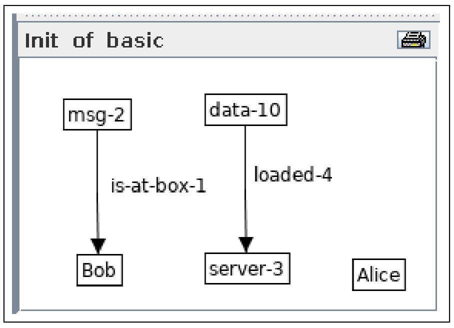


(d) Rule *delete message*

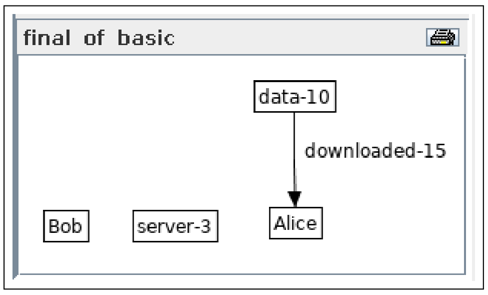
Once the set F' of actions was created, we proceed to calculating the causal relation, as described in Definition 3.18. This relation is the very first indicative of whether it is possible to construct an occurrence graph grammar for the given collection of rules. Remember that this relation must be a partial order, otherwise the totality of rules in F are not executable. Specifically, the causal relation gives us hints over the order in which the actions must be performed to accomplish the functionality aims, for example: *Bob* must sent *the message* to *the server* before it reaches *Alice*. The occurrence relation of our example is [getData < deleteMsg, getData < receiveMsg, receiveMsg < deleteMsg, sendMsg < deleteMsg, sendMsg < getData, sendMsg < receiveMsg]. It is also easy to see that this relation is a partial order. In fact, there is only one possible total order derived from it: [$sendMsg < getData < receiveMsg < deleteMsg$].

The next step consists of using the causal relation and the graph Occ in order to generate the initial and final graphs of our target grammar. These graphs correspond to the necessary input and expected output of performing F in a minimal context. In order to create the graphs, we delete from Occ the elements that are created (resp. deleted) by the rules in F' according to the causal relation. For example: *Alice* is a person who can never be created by any action of our system, no matter how advanced. As a consequence she must be present in any initial states of the actions performed with her. *The message* sent by *Bob* is never created by any action either, hence it must be present in the initial graph. However, it is deleted by the action *deleteMsg*, so it must not appear in the final graph.

Figure 5.5: Instance graphs



(a) Initial graph



(b) Final graph

In general, after simply deleting those elements from Occ, the result may be that either *Initial* or *Final* graphs are not valid, in the case that any source or target node of an edge is deleted, but not the edge itself. This means that the execution of F would need to begin in or lead to an inconsistent state, therefore no sequencing of actions in F could be performed in a real execution. However, if they are indeed valid graphs, as they are in our example, we have just found the initial and final (minimal) states of the system regarding the execution of all rules in F.

Notice that, if the steps listed so far (1 to 4) were able to be successfully performed, we have a grammar $OGG = (Occ, I, F')$ that is not only doubly-typed, but also strongly safe in the sense of Definition 3.16. Therefore it is a candidate to be an occurrence graph grammar.

In step 5, we proceed towards creating the occurrence relation and occurrence relation restrictions. In 5a, we calculate all produce-forbid conflicts and delete-forbid dependencies between the rules in F' by using the categorial algorithms presented in Definitions 3.25 and 3.27. Initially, we do not know whether these conflicts and dependencies are exercised during the execution of the underlying grammar.

In our example, we find one conflict and one dependency induced by NACs. The first is a produce-forbid between *getData* and *sendMsg* regarding the creation of the attachment between the piece of data and the message in *getData*, which is forbidden by the NAC of *sendMsg*. The second, a delete-forbid between *deleteMsg* and *sendMsg*, regarding the deletion of that same attachment. Given the information collected so far we can deduce whether any of these conditions exists in the local context. Hence, we use the information acquired in previous steps to classify those conflicts and dependencies as concrete, abstract or non-existent as specified in Definitions 3.25 and 3.27.

In this particular case we already know, by looking at the causal relation, that *sendMsg* must be executed before *getData*, therefore the later action can never trigger the NAC of an action that occurs before it, and this conflict is non-existent. Similarly, for the delete-forbid, as we know that the condition that triggers the NAC of *sendMsg* does not exist prior to any possible of its executions, it is not necessary for *deleteMsg* to remove the element triggering the NAC, therefore this dependency is also non-existent.

The occurrence relation $\leq_o$ is then calculated from the causal relation together with the concrete conflicts and dependencies. In step 5b we create the set R_i of restrictions as the union of all abstract conflicts and dependencies calculated before. In our example, since no concrete conflicts or dependencies were found, the occurrence relation remains equal to the causal relation. As for the set of restrictions, since no abstract conflicts or dependencies were found, it remains empty.

Finally, we have the strongly safe grammar OGG together with its corresponding occurrence relation $\leq_o$ and a set of restrictions R. As it was shown, if $\leq_o$ is a partial order and it is possible to find a total ordering of it that respects all restrictions in R, it follows that OGG is an occurrence graph grammar according to Definition 3.32.

In our previous example, no situation with abstract conflicts or dependencies was found. This grammar and the particular execution chosen have a causal relation which forces a sequencing of actions that avoids the existence of abstract conflicts/dependencies. In our case studies, this situation has shown to be very common in graph grammars extracted from use cases, where there are clear steps that must be followed in a specific order to accomplish the completion of a functionality. Nonetheless, in graph grammars that model systems with higher parallelism or concurrency, situations with abstract conflicts/dependencies have shown to arise more often. We now introduce another graph grammar example with more independent rules to illustrate the construction of an occurrence graph grammar with NACs which has a non-empty set of restrictions.

Example 5.5. This graph grammar models a common traffic situation, where a pedestrian crosses or not a street according to the state of a traffic light. The grammar is depicted in Figure 5.6 while its rules are summarized as follows:

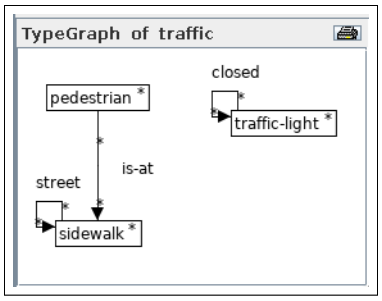
(b) *walk:* a pedestrian is on a sidewalk and crosses the street to be on another sidewalk, however they can not do so if there is a closed traffic light.

(c) *open:* turns a closed traffic light into an open one.

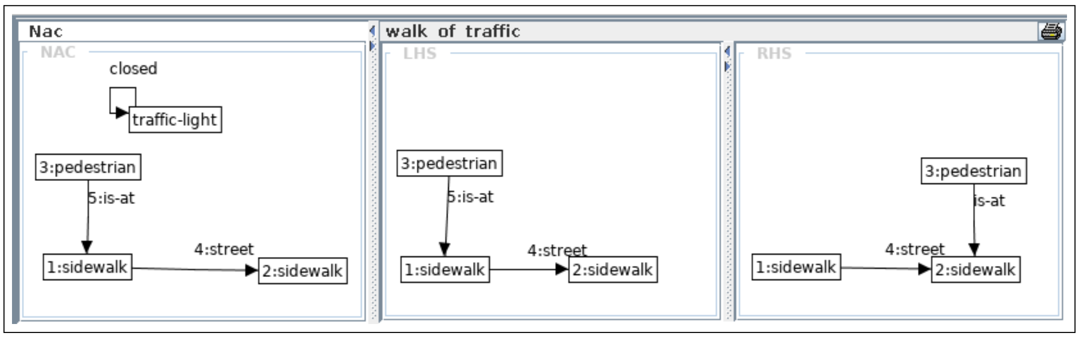
(d) *close:* turns an open traffic light into a closed one.

■

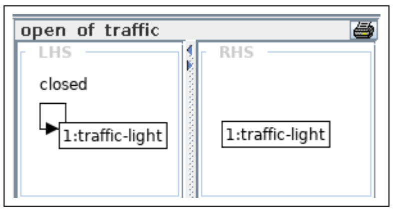
Figure 5.6: Graph Grammar of the Traffic System



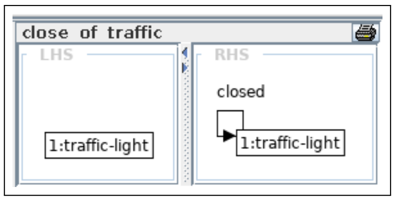
(a) Type Graph of the Grammar



(b) Rule *walk*



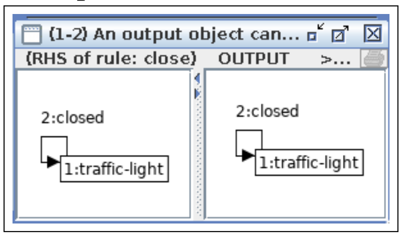
(c) Rule *open*



(d) Rule *close*

As before, to begin the construction of the occurrence graph grammar with NACs from our graph grammar, we need a collection of rules and an input-output relation. The chosen collection consists of one copy of each rule in the original grammar. The *IO*

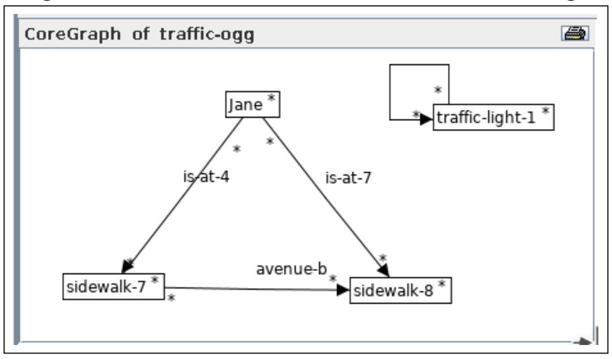
Figure 5.7: Input output relation for the traffic graph grammar



relation is also very simple: it only identifies the traffic lights and the edges that represent they are closed in rules *close* and *open*. The relation is depicted on Figure 5.7.

The first step in the construction, the amalgamation of rules, results in the core graph depicted in Figure 5.8. Notice that, differently from the type graph in Figure 5.6a, it has concrete elements, a generic pedestrian is now *Jane*, there are two specific sidewalks which are connected by the *Avenue B*, etc.

Figure 5.8: Amalgamation (colimit) of rules for the traffic grammar example.



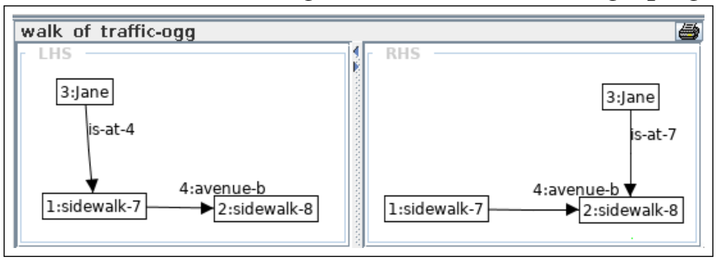
The second step retypes the rules generating the actions depicted in Figure 5.9. Once again, NACs are not shown due to limitations with AGG format compatibility for double-typed graph grammars.

The third step is to calculate the causal relation. For this particular execution of the grammar it consists of the set cointaining only the pair $close < open$. Therefore, in this context, the traffic light must be closed before it opens and, so far, the action *walk* does not relate to any of the others. This relation is a partial order, given that there are multiple options of total orderings which respect this relation: $[walk < close < open]$, $[close < walk < open]$ and $[close < open < walk]$.

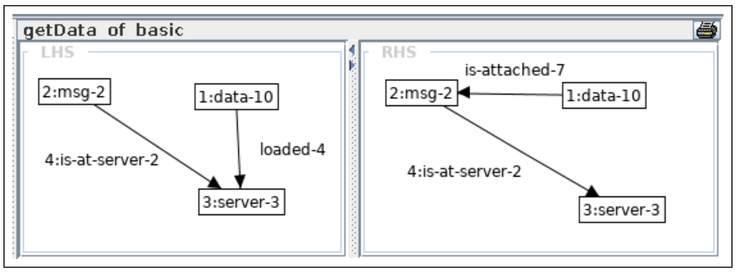
After calcultating the causal relation, we construct the *Initial* and *Final* graphs obtained by deletion of elements created (resp. deleted) by the actions in the core graph. The graphs for this execution are shown in Figure 5.10.

The characterization of conflicts and dependencies is where the behaviour of this graph grammar greatly differs from our previous example. There is one potential produce-

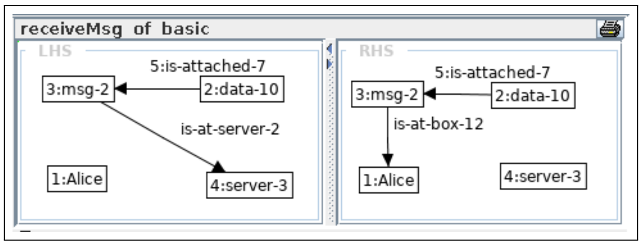
Figure 5.9: The set of actions generated for the traffic graph grammar.



(a) Action *walk*

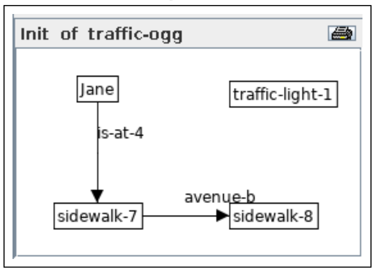


(b) Action *open*

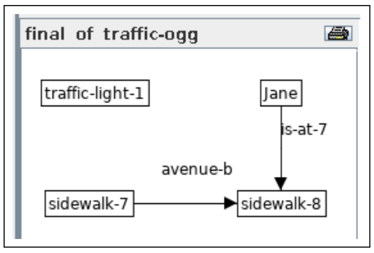


(c) Action *close*

Figure 5.10: Instance graphs for the traffic grammar



(a) Initial graph



(b) Final graph

forbid conflict between actions *close* and *walk* and one potential delete-forbid dependency between actions *open* and *walk*. Both the conflict and the dependency act over the same triggering element, the *closed* edge in the NAC of action *walk*. This element is not present in the *Initial* graph, as it is created by action *close*. Also, the element is not (causally) related to action *walk*, given that the actions which create and delete it are themselves not related to that action. It is easy to see that the action *walk* can either occur before action *close* or after action *open*, but never in between them, because *Jane* must not cross *Avenue B* while the traffic light is closed. Therefore, we have an abstract produce-forbid conflict as well as an abstract delete-forbid dependency which, in this case, are both denoted by the tuple $(walk, close, open)$. Moreover, as there are no concrete conflicts and dependencies induced by NACs, the occurrence relation remains equal to the causal relation.

Finally, we have to find at least one total ordering of the actions respecting both the occurrence relation $\leq_o = [close <_o open]$ and the set of occurrence relation restrictions $R = (walk, close, open)$. In fact, there are two such orderings: $[walk < close < open]$ and $[close < open < walk]$. Therefore, the strongly safe graph grammar constructed from this graph grammar execution is also an occurrence graph grammar with NACs.

5.3 Generating Test Cases

Occurrence Graph Grammars may be used to generate a set of tests for a Graph Grammar. The process of constructing an occurrence graph grammar may provide insights for tests even when it fails. In the case where OGGs are found, we have test cases for successful executions of the system under modelling and conditions under which the system should execute. In the case where OGGs are not found, we have test cases for executions where the system must always fail.

We use the occurrence relation and the set of abstract restrictions as test oracles, to define the acceptance of tests: any path that complies with the format imposed by them is considered valid and must always succeed. On the other hand, paths that break at least one of such restrictions are considered invalid, and their corresponding test cases must always capture them as failures.

The tests are represented by the concrete orderings of the rules execution, orderings of elements creation/deletion, and by the initial and final graphs. An ordering of rules is one of (possibly) many valid orders in which the rules can be applied according to the occurrence relation. An ordering of elements represents an ordering in which the state of the system may be constructed, whereas the initial and final graphs translate the valid/necessary data for the input and output of each test. More specific usability details can be found on the Verigraph tutorial at <https://github.com/Verites/verigraph-tutorial/releases>.

The output of Verigraph for an OGG creation and its test case generation is shown in Figure 5.11. On the first figure, Verigraph performs the basic verifications to check whether the generated output is, in fact, an occurrence grammar.

The analysis file contains a summary of the results for calculation of conflicts and dependencies among rules and among elements. For example: which conflicts and

Figure 5.11: Tool command line output

```
Testing Serialization:
[OK] Unique creations and deletions
[OK] Initial graph is valid
[OK] Final graph is valid
[OK] Concrete occurrence relation is a total order
[OK] Concrete elements relation is a total order
[WARN] There are abstract restrictions
Analysis written in ogg_execution_analysis
Test cases written in ogg_execution_test_cases
Doubly-typed grammar saved in ogg_execution.ggx
```

dependencies were found; for the conflicts and dependencies induced by NACs which are the triggering elements of each NAC; the causal relation between elements and actions that created/deleted them, etc. Figure 5.12 depicts the content of the analysis file for our traffic example.

Figure 5.12: Analysis file content

```
Conflicts and Dependencies:
[
  Interaction {firstRule = "close", secondRule = "walk",
   ↪  interactionType = ProduceForbid, nacInvolved = Just 0},
  Interaction {firstRule = "close", secondRule = "open",
   ↪  interactionType = ProduceUse, nacInvolved = Nothing},
  Interaction {firstRule = "open", secondRule = "walk",
   ↪  interactionType = DeleteForbid, nacInvolved = Just 0}
]

Creation and Deletion Relation:
[
  (Edge 1,Rule "open"),(Edge 4,Rule "walk"),(Rule "close",Edge
   ↪  1),(Rule "walk",Edge 7)
]

Conflicts and dependencies induced by NACs:
[
  (Interaction {firstRule = "close", secondRule = "walk",
   ↪  interactionType = ProduceForbid, nacInvolved = Just
   ↪  0},Edge 1),
  (Interaction {firstRule = "open", secondRule = "walk",
   ↪  interactionType = DeleteForbid, nacInvolved = Just 0},Edge
   ↪  1)
]
```

The test cases file contains information we consider relevant to a test designer, such as the rules and elements involved in that particular execution of the graph grammar represented by the occurrence graph grammar, as well as the occurrence relation, a total ordering of rules application (if found) and the set of restrictions (if found). Figure 5.13 presents the content of the test cases files for our traffic example.

Figure 5.13: Test cases file content

```
Rules involved:
[Rule "close",Rule "open",Rule "walk"]

Concrete Rules Relation:
[(Rule "close" < Rule "open")]

Elements involved:
[Node 1,Node 7,Node 8,Node 9,Edge 1,Edge 3,Edge 4,Edge 7]

Elements Relation:
[(Edge 4 < Edge 7)]

Rules Ordering: Just [Rule "close",Rule "open",Rule "walk"]

Elements Ordering: Just [Node 1,Node 7,Node 8,Node 9,Edge 1,Edge
 ↪  3,Edge 4,Edge 7]

Set of Abstract Restrictions:
[
  (AbstractProduceForbid: Rule "walk" must not occur between
   ↪  [Rule "close" < Rule "open"]),
  (AbstractDeleteForbid: Rule "walk" must not occur between
   ↪  [Rule "close" < Rule "open"])
]
```

Finally, the `.ggx` file presents a translation of the constructed occurrence graph grammar from Verigraph format to AGG format in order to support the OGG graphical visualization. It contains its actions (without NACs), initial and final graphs and core graph (which assumes the role of the type-graph in AGG) as shown in the previous section.

6 RELATED WORK

We proposed a new semantical model for Graph Grammars with Negative Application Conditions based on unfoldings (Occurrence Graph Grammars). It relies on and extends previous work that defined the semantics of Graph Grammars as one unfolding structure (describing a non-deterministic computation) or as a set of unfolding structures (describing a set of deterministic computations). The main difference is that our approach considers Negative Application Conditions, that were not tackled by any of the existing approaches.

6.1 Semantics of Graph Grammars

6.1.1 Unfolding Semantics

Unfolding techniques were initially proposed for Petri Nets (NIELSEN; PLOTKIN; WINSKEL, 1981) and later generalized to Graph Grammars (RIBEIRO, 1996) and other Adhesive Systems (BALDAN et al., 2009). The idea of Unfolding of a Graph Grammar uses that of Occurrence Graph Grammars and consists of a non-deterministic process that expresses the Graph Grammar behaviour: for a given grammar G, it is possible to build a sequence of Occurrence Grammars O_n where each O_n represents all computations up to depth n, where the depth of a concurrent computation is the length of a maximally parallel execution of the computation (BALDAN et al., 2009).

The Unfolding begins at the start graph of the grammar G and proceeds by continuously applying all rules in all possible ways to its concurrent subgraphs until depth n, then each occurrence of a rule and each new graph element created is recorded in the unfolding. As in our approach, no element is ever deleted from this structure, therefore it can be seen as a "partial application" of a rule to a match, given that it generates the new items created by the application of the rule, without actually erasing the elements that have been deleted (BALDAN; CORRADINI; KÖNIG, 2008).

6.1.2 Abstract Graph Processes

The concept of Graph Processes was introduced by (CORRADINI; MONTANARI; ROSSI, 1996) and it also uses that of Occurrence Graph Grammars. A Graph Process for a grammar $GG = (TG, I, P)$ consists of an Occurrence Graph Grammar $OGG = \left(C^T, I^{C^T}, A\right)$ together with its causal relation $\leq_c$ and a pair of morphisms (mg, mp) mapping (1) the core graph of the occurrence grammar to the type graph of its underlying grammar $mg : C^T \rightarrow T$ and (2) the actions in A to their corresponding rules in P, $mp : A \rightarrow P$.

Abstract Graph Processes are classes of Graph Processes which have the same structure and despite having different concrete identities for the elements and actions should still be considered equivalent. Therefore, they encompass a greater number of possible grammar executions with a smaller number of processes (or occurrence grammars).

6.2 Generating tests cases

6.2.1 Unfolding of graph transformations

In (BALDAN; KÖNIG; STÜRMER, 2004) the unfolding of graph transformation systems is used in order to generate test cases for code generators of the automotive industry. Their work is based on the category of hypergraphs and, similarly to ours, it also extends their framework to include negative application conditions.

According to the authors, code generators are widely used in the development of embedded software, however they lack the maturity and testing when compared to compilers of standard programming languages. Thus, one of the biggest problems in testing code generators is the difficulty to describe the transformation rules from a graphical model to a target language (as well as the interactions amongst the rules) in a precise and formal way. Therefore, they propose a graph transformation based approach for systematically deriving test cases in this particular scenario.

Their approach is based on the use of unfolding of graph transformation systems (RIBEIRO, 1996) over two graph grammars, a *generating grammar*, responsible for generating all possible input models, and an *optimising grammar*, which formalises specific transformation steps towards code optimisation. The main purpose is to test the optimisation phase, in an attempt to ensure no mistakes will be introduced by improving the code while preserving its behaviour.

In this environment, a test case for a subset of optimising rules R is an input model G such that all the rules in R can be computed over G. This is similar to our approach where, for a sequence to be executable, all rules in that sequence must be applicable over the occurrence graph. However, we also use the non-executable ones in order to generate test cases where the system is indeed expected to fail.

In spite of using the double-pushout approach the same way we do, they use the category of (typed-)hypergraphs with some restrictions, such as: isolated nodes[1] can never exist in the LHS or in any NAC of a rule, every node that by any means becomes disconnected is considered "garbage collected", and NACs can extend the match only with one edge (which, according to the authors, makes them weaker than general NACs). Thus, despite of several similarities, our results are not fully comparable.

Their approach is focused on testing the optimization steps of code generation, but little is presented about the code generation itself. Also, although their approach was proposed for a very practical application, no supporting tool was presented.

6.2.2 Visual Contracts

An approach proposed by (HECKEL; KHAN; MACHADO, 2011), (KHAN; RUNGE; HECKEL, 2012a), (KHAN; RUNGE; HECKEL, 2012b), (RUNGE; KHAN; HECKEL, 2013) focusing mainly on generating test cases for service-oriented or component-based systems. Given that systems of this kind often hide their implementation, the authors use interface descriptions known as visual contracts[2] in order to model the observable behaviour of the system.

Coverage criteria is defined by means of static analysis, where potential conflicts and dependencies amongst visual contracts are calculated and used to build a dependency graph. In this situation, despite of being called "a dependency graph", this structure is rather similar to our occurrence relation, summarizing the results of both conflict and dependency analysis while representing the possible orderings in which the visual contracts may be executed.

In the process of generating test cases, it is necessary to provide an initial graph, which is used to find out which visual contracts are applicable to it. One of such visual contracts is chosen as the first step and all the paths through the dependency graph in

[1]Nodes that are neither the source nor the target of any edge.

[2]Formally regarded as graph transformation rules with operation signatures.

which each rule is applied at most once are computed and stored as a set of rule sequences. Thereafter, the sequences are enriched to encompass rules with multiple dependencies and lately redundant rules contained in larger ones are removed. Afterwards, each sequence is executed (if possible), and any new edges in the dependency graph reached by them are added to coverage. The entire process is then repeated as long as the coverage shows improvement.

In comparison to our work, this approach has both advantages and limitations. As an example of the first, there are: the possibility of working with attributed typed graph transformation systems and multi-rules. As for the second: it requires more user involvement during the process of test case generation, it does not enclose negative application conditions, it was planned to work in a configuration where each rule is applied at most once and, although being an extension of AGG (TAENTZER, 2000), the tool is not available for download.

6.3 Tools

Although Verigraph (COSTA et al., 2016; BEZERRA et al., 2017; AZZI et al., 2018) is the first tool in the field to implement Occurrence Graph Grammars (with or without Negative Application Conditions) for general Graph Grammars while also having its source code closely related to Category Theory, there are other Graph Grammar tools with different approaches and analysis available. In the following, we list the ones which are to some extent closely related to Verigraph.

6.3.1 AGG

The Attributed Graph Grammar System (TAENTZER, 2000) is a graph transformation tool which supports typed graph grammars. Its main rewriting approach is the SPO, but it can configured to execute the analyses of graph grammars as in the DPO approach. AGG supports attributed graphs, thus elements of a graph can be enriched with algebraic types. This tool is focused on static analysis such as critical pair/sequence analysis and concurrent rules, but also several others like termination and consistency checking.

6.3.2 Groove

The Graphs for Object-Oriented Verification (GROOVE) Tool Set (RENSINK, 2004) aims for modelling graph grammars. As AGG, its rewriting engine also implements the SPO approach. Its main focus is the generation and exploration of state space, implementing many exploration strategies as well as an efficient search for isomorphic states. Graphs in GROOVE are untyped, however it does support labelling to simulate types in complex systems.

6.3.3 SyGrAV

The Symbolic Graph Analysis and Verification (SyGrAV) tool prototype (DECKWERTH, 2016) is a tool for static verification of attributed (symbolic) graph transformation systems. It is based on the DPO approach and implemented in Java. It shares with Verigraph its inspiration to maintain the source code as closely related to the theory as possible, for which it makes use of a series of APIs defining contracts over the behaviour of the underlying Categories.

6.3.4 Augur 2

Augur 2 is a tool for the analysis of (attributed) graph transformation systems using approximative unfolding techniques (KöNIG; KOZIOURA, 2008). It allows the analysis of Graph Transformation Systems by approximating them with Petri nets. The tool is also based in the DPO approach (with some restrictions) and its source code is written in `C++`.

7 CONCLUSIONS

An Occurrence Graph Grammar (OGG) is a way of representing the semantics of a graph grammar as a graph grammar itself. OGGs were previously defined by (RIBEIRO, 1996) and (CORRADINI; MONTANARI; ROSSI, 1996). Notwithstanding, its original definitions do not consider Negative Application Conditions, which are nowadays essential for modelling of complex, real-life systems.

In our work, we proposed an extension of the framework of occurrence graph grammars in order to include Negative Application Conditions (NACs), which has not been done so far. Additionally, the process of constructing occurrence graph grammars was implemented on Verigraph, a systems specification and verification software based on graph rewriting.

In order to implement the techniques presented in this work, we took advantage of the previous work already existent in Verigraph, mainly based on categorial notions. In this sense, one of our derived contributions was the implementation of more basic categorial operations than originally present, such as coequalizers, coproducts and colimits. This operations may now be used by different category implementations with relatively little effort.

Moreover, Verigraph is now (as far as we know) the only tool in this field that supports the construction and analysis of doubly-typed graph grammars and occurrence graph grammars, both with and without NACs.

Furthermore, Verigraph is a free and open source software, publicly available at github. Thus, we expect that any users in the community of graph grammars and category theory in general can find it and use it, besides making suggestions and even implementing new features according to their specific needs.

We also provided some insight about how the semantic of graph grammars, in the form of occurrence graph grammars, can be used in order to generate sets of test cases to the system under modelling, a strategy we intend to refine in our future work.

7.1 Open Questions and Future Work

Although our main objectives were accomplished during the work of this thesis, there are several other open paths that could (should) be investigated.

Incremental NACs: The entire process of calculating the occurrence graph grammars and the later generation of test cases depends on all the NACs of the input grammar being incremental. Despite incremental NACs being sufficient for most applications and that our case studies only used grammars which respect this restriction, there may be cases where it is needed to use grammars with general NACs.

The current implementation of Verigraph assumes the input grammars have incremental NACs only, therefore it remains as a future work to implement the algorithm that compiles arbitrary NACs to incremental ones and use it as a previous step to our main work.

NACs in strongly safe grammars: In our work, we defined NACs in strongly safe grammars that are only single-typed. However, the definition of doubly-typed NACs seems to be useful if one wants to point concretely each element in the core graph that triggers the NAC of an action.

The idea is to create a doubly-typed NAC for each concrete triggering of the original NAC over the core graph. Moreover, this translation may yield the creation of (possibly) many doubly-typed NACs for each original single-typed. Thus, it remains as future work to formally define this other kind of NAC, to verify whether and when it would really be useful, how to use it to improve the expressiveness of test cases generation and then implement it on Verigraph.

Different graph rewriting approaches: Occurrence graph grammars were originally defined for DPO and SPO approach without NACs, our extension adds NACs to the DPO approach. It remains open how to extend them for SPO or even other different approaches, such as SqPO and AGREE.

Complexity of finding a total ordering: To this point, it is not clear to us what is the complexity of an algorithm to find (or to check if it is possible to find) a total ordering of actions of a strongly safe grammar that respects both the concrete occurrence relation and arbitrary abstract restrictions.

In our practical applications so far we did not find strongly safe graph grammars with a prohibitive number of constraints (for many chosen executions there was no constraint at all) in the set of occurrence relation restrictions. We believe this happened because most of our graph grammars were extracted from use cases which provided "natural" orderings under which the actions must be applied, thus limiting the possible cases in which abstract conflicts/dependencies might occur. However, in graph grammars which were not extracted from use cases and where the system execution had more possibilities to execute in parallel than sequentially, those situations appeared more frequently. Therefore, more study is necessary regarding this aspect, specifically, we want to be sure whether it is possible to find total orderings for arbitrary strongly safe graph grammars or at least under which conditions (besides an empty set of abstract restrictions) would it still be feasible. After that, we will know if this process is suitable to generate tests for grammars that model arbitrary systems and not only those described by use cases.

Concurrent Graphs: In the original definitions of Occurrence Graph Grammars, there exist the concept of concurrent graphs, which consist of all reachable graphs while executing the underlying grammar. Such a concept was not used, therefore not defined in our extension for DPO approach with NACs. Refining our work to include this concept would both complete our definitions regarding the previous works and improve the generation of tests, by allowing us to know which (intermediary) states the system can or can not assume.

Graphical Interface: Regarding input and output, Verigraph does not have an operational graphical user interface (GUI) yet, using the AGG tool and its `.ggx` file format for providing this operations. However, AGG does not support doubly-typed graph grammars nor NACs under this framework, thus the graphical visualization of the output of the test generation process is not completely possible, which makes the development of a GUI dedicated to Verigraph a necessary step to address this issue[1].

[1]the development of a responsive, dedicated GUI is currently one of the main focus of the Verites group

REFERENCES

AZZI, G. G. et al. The verigraph system for graph transformation. In: ____. **Graph Transformation, Specifications, and Nets: In Memory of Hartmut Ehrig**. Cham: Springer International Publishing, 2018. p. 160–178. ISBN 978-3-319-75396-6. Disponível em: <https://doi.org/10.1007/978-3-319-75396-6_9>.

BALDAN, P. et al. Unfolding grammars in adhesive categories. In: KURZ, A.; LENISA, M.; TARLECKI, A. (Ed.). **Algebra and Coalgebra in Computer Science**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 350–366. ISBN 978-3-642-03741-2.

BALDAN, P.; CORRADINI, A.; KÖNIG, B. Unfolding graph transformation systems: Theory and applications to verification. In: ____. **Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 16–36. ISBN 978-3-540-68679-8. Disponível em: <https://doi.org/10.1007/978-3-540-68679-8_3>.

BALDAN, P.; KÖNIG, B.; STÜRMER, I. Generating Test Cases for Code Generators by Unfolding Graph Transformation Systems. In: **Graph Transformations, Proceedings**. [S.l.: s.n.], 2004. v. 3256, p. 194–209. ISBN 3-540-23207-9.

BECKER, T. R. **VeriGraph: A Tool For Model Checking Graph Grammars**. Tese (Bachelor) — Universidade Federal do Rio Grande do Sul, 2014.

BEZERRA, J. S. et al. **Verigraph: Software specification and verification system based on graph rewriting**. 2017. Disponível em: <https://github.com/Verites/verigraph>.

BEZERRA, J. S. et al. Formal Verification of Health Assessment Tools : a Case Study. **Electronic Notes in Theoretical Computer Science**, v. 324, n. WEIT 2015, the Third Workshop-School on Theoretical Computer Science, p. 31–50, 2016. ISSN 15710661.

BEZERRA, J. S.; RIBEIRO, L. Calculation and Applications of Concurrent Rules. In: CAVALHEIRO, S. A. d. C. et al. (Ed.). **Anais da 1ª Escola de Informática Teórica e Métodos Formais**. Porto Alegre/RS: Sociedade Brasileira de Computação – SBC, 2016. p. 135–144. ISBN 9788576693574.

CORRADINI, A. et al. Agree – algebraic graph rewriting with controlled embedding. In: PARISI-PRESICCE, F.; WESTFECHTEL, B. (Ed.). **Graph Transformation**. Cham: Springer International Publishing, 2015. p. 35–51. ISBN 978-3-319-21145-9.

CORRADINI, A.; HECKEL, R. Canonical Derivations with Negative Application Conditions. In: . [S.l.: s.n.], 2014. p. 207–221.

CORRADINI, A. et al. Transformation Systems with Incremental Negative Application Conditions. In: . [S.l.: s.n.], 2013. p. 127–142.

CORRADINI, A. et al. Sesqui-pushout rewriting. In: **Proceedings of the Third International Conference on Graph Transformations**. Berlin, Heidelberg: Springer-Verlag, 2006. (ICGT'06), p. 30–45. ISBN 3-540-38870-2, 978-3-540-38870-8. Disponível em: <http://dx.doi.org/10.1007/11841883_4>.

CORRADINI, A.; MONTANARI, U.; ROSSI, F. **Graph Processes**. 1996. 241–265 p.

CORRADINI, A. et al. **Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach**. [S.l.], 1996.

COSTA, A. et al. Verigraph: A System for Specification and Analysis of Graph Grammars. In: **Brazilian Symposium on Formal Methods**. [S.l.: s.n.], 2016. p. 78–94. ISBN 978-3-642-10451-0.

COSTA, A.; MACHADO, R.; RIBEIRO, L. Evolving Negative Application Conditions. In: CAVALHEIRO, S. A. d. C. et al. (Ed.). **Anais da 1ª Escola de Informática Teórica e Métodos Formais**. Porto Alegre/RS: Sociedade Brasileira de Computação – SBC, 2016. p. 74–82. ISBN 9788576693574.

COTA, É. et al. Using formal methods for content validation of medical procedure documents. **International Journal of Medical Informatics**, apr 2017. ISSN 13865056.

DECKWERTH, F. **Static Verification Techniques for Attributed Graph Transformations**. 220 p. Tese (Doutorado) — Technischen Universität Darmstadt, 2016.

EHRIG, H. et al. **Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN 3540311874.

EHRIG, H. et al. Handbook of graph grammars and computing by graph transformation. In: ROZENBERG, G. (Ed.). River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1997. cap. Algebraic Approaches to Graph Transformation. Part II: Single Pushout Approach and Comparison with Double Pushout Approach, p. 247–312. ISBN 98-102288-48. Disponível em: <http://dl.acm.org/citation.cfm?id=278918.278930>.

EILENBERG, S.; MACLANE, S. General Theory of Natural Equivalences. **Transactions of the American Mathematical Society**, v. 58, p. 231–294, 1945. ISSN 0002-9947.

GIESE, H.; VOGEL, T.; WATZOLDT, S. Towards Smart Systems of Systems. In: SIRJANI, M.; MARJAN, D. (Ed.). **Fundamentals of Software Engineering**. Springer, 2015. v. 5961, p. 1–29. ISBN 0133056996. Disponível em: <http://link.springer.com/10.1007/978-3-319-24644-4_1>.

HABEL, A.; HECKEL, R.; TAENTZER, G. Graph grammars with negative application conditions. **Fundam. Inf.**, IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 26, n. 3,4, p. 287–313, dez. 1996. ISSN 0169-2968.

HABEL, A.; PENNEMANN, K.-H. Nested constraints and application conditions for high-level structures. In: ____. **Formal Methods in Software and Systems Modeling: Essays Dedicated to Hartmut Ehrig on the Occasion of His 60th Birthday**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 293–308. ISBN 978-3-540-31847-7. Disponível em: <https://doi.org/10.1007/978-3-540-31847-7_17>.

HECKEL, R.; KHAN, T. A.; MACHADO, R. Towards Test Coverage Criteria for Visual Contracts. In: GADDUCCI, F.; MARIANI, L. (Ed.). **Proceedings of the Tenth InternationalWorkshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2011)**. [S.l.: s.n.], 2011. v. 41.

JUNIOR, M. O. et al. **Use Case Analysis Based on Formal Methods: An Empirical Study**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015. 110–130 p. (Lecture Notes in Computer Science). ISBN 978-3-540-25327-3.

KHAN, T. A.; RUNGE, O.; HECKEL, R. Testing against visual contracts: Model-based coverage. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. [S.l.: s.n.], 2012. v. 7562 LNCS, p. 279–293. ISBN 9783642336539. ISSN 03029743.

KHAN, T. A.; RUNGE, O.; HECKEL, R. Visual Contracts as Test Oracle in AGG 2 . 0. **Electronic Communications of the EASST. Proceedings of the 11th International Workshop on Graph Transformation and Visual Modeling Techniques**, v. 47, p. 0–13, 2012.

KöNIG, B.; KOZIOURA, V. Augur 2 — a new version of a tool for the analysis of graph transformation systems. **Electronic Notes in Theoretical Computer Science**, v. 211, p. 201 – 210, 2008. ISSN 1571-0661. Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006). Disponível em: <http://www.sciencedirect.com/science/article/pii/S1571066108002570>.

LAMBERS, L. **Certifying Rule-Based Models using Graph Transformation**. 258 p. Tese (Doutorado) — Elektrotechnik und Informatik der Technischen Universitä Berlin, 2010.

LAMBERS, L.; EHRIG, H.; OREJAS, F. Conflict Detection for Graph Transformation with Negative Application Conditions. In: CORRADINI, A. et al. (Ed.). **Graph Transformations: Third International Conference, ICGT 2006 Natal, Rio Grande do Norte, Brazil, September 17-23, 2006 Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. (Lecture Notes in Computer Science, v. 4178), p. 61–76. ISBN 978-3-540-38872-2.

LAMBERS, L. et al. Parallelism and Concurrency in Adhesive High-Level Replacement Systems with Negative Application Conditions. **Electronic Notes in Theoretical Computer Science**, Elsevier B.V., v. 203, n. 6, p. 43–66, 2008. ISSN 15710661.

LAMBERS, L.; EHRIG, H.; TAENTZER, G. Sufficient Criteria for Applicability and Non-Applicability of Rule Sequences. **Electronic Communications of the EASST**, v. 10, 2008.

MACHADO, R.; RIBEIRO, L.; HECKEL, R. Rule-based transformation of graph rewriting rules: Towards higher-order graph grammars. **Theoretical Computer Science**, v. 594, p. 1–23, 2015.

NIELSEN, M.; PLOTKIN, G.; WINSKEL, G. Petri nets, event structures and domains, part i. **Theoretical Computer Science**, v. 13, n. 1, p. 85 – 108, 1981. ISSN 0304-3975. Special Issue Semantics of Concurrent Computation. Disponível em: <http://www.sciencedirect.com/science/article/pii/0304397581901122>.

PENNEMANN, K.-h. **Development of Correct Graph Transformation Systems**. Tese (PhD) — Carl von Ossietzky Universität Oldenburg –, 2009.

PIERCE, B. **Basic Category Theory for Computer Scientists**. [S.l.]: MIT Press, 1991. (Foundations of computing series: research reports and notes). ISBN 9780262660716.

RENSINK, A. The GROOVE Simulator: A Tool for State Space Generation. In: PFALTZ, J. L.; NAGL, M.; BÖHLEN, B. (Ed.). **Applications of Graph Transformations with Industrial Relevance: Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. (Lecture Notes in Computer Science, v. 3062), p. 479–485. ISBN 978-3-540-25959-6.

RIBEIRO, L. **Parallel composition and unfolding semantics of graph grammars**. 215 p. Tese (PhD) — Technischen Universität Berlin, 1996.

ROZENBERG, G. **Handbook of Graph Grammars aand Computing by Graph Transformation**. [S.l.: s.n.], 1997. v. 1. ISBN 9810228848.

RUNGE, O.; KHAN, T. A.; HECKEL, R. Test Case Generation Using Visual Contracts. In: TICHY, M.; RIBEIRO, L. (Ed.). **Proceedings of the 12th InternationalWorkshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2013)**. [S.l.: s.n.], 2013. v. 58, p. 1–12.

RYDEHEARD, D. E.; BURSTALL, R. M. **Computational Category Theory**. [S.l.]: Prentice-Hall, 1988. v. 152. 263 p. ISBN 0131627368.

TAENTZER, G. AGG: A Tool Environment for Algebraic Graph Transformation. In: NAGL, M.; SCHÜRR, A.; MÜNCH, M. (Ed.). **Applications of Graph Transformations with Industrial Relevance: International Workshop, AGTIVE'99 Kerkrade, The Netherlands, September 1–3, 1999 Proceedings**. [S.l.]: Springer Berlin Heidelberg, 2000. (Lecture Notes in Computer Science, v. 1779), p. 481–488. ISBN 978-3-540-45104-4.

APPENDIX A — CATEGORY THEORY

Category theory is a powerful mathematical framework, defined by (EILENBERG; MACLANE, 1945), that provides an abstract way to reason about mathematical structures and the relationships between them.

Categories are particularly useful in Computer Science, having several applications such as design of programming languages, implementation techniques, semantic models, concurrency models, type theory, among others (PIERCE, 1991).

Category theory is also the basis for graph transformation systems, which is central to the work proposed in this thesis. Therefore, we present a brief introduction of the field and basic categorial constructions that are used throughout this work.

Definition A.1 (Category)**.** A category **C** consists of a collection of *objects* and a collection of *arrows* between objects (also called *morphisms*) such that:

1. for all arrows $f : A \to B$, $g : B \to C$ and $h : C \to D$, with objects A, B, C, D not necessarily distinct, the composition of arrows is associative:

   $h \circ (g \circ f) = (h \circ g) \circ f$;

2. for every object A there is an *identity* arrow $id_A : A \to A$ such that for any arrow $f : A \to B$:

   $id_B \circ f = f$ and $f \circ id_A = f$.

□

Example A.2 (Category of Sets)**. Set** is the category whose objects are *sets* and the arrows are *total functions* between sets. The composition operator is given by function composition, while the identity arrow is given by the identity function.

■

Definition A.3 (Diagram)**.** Given a category **C**, a diagram in **C** is a collection of vertices and directed edges such that, if an edge in the diagram is named with an arrow f and f has domain A and codomain B, then the outgoing vertex of the edge must be named A and the incoming vertex B.

A diagram is said to *commute* if, for every pair of objects A, B, all the paths in the diagram from A to B are equal. In other words, each path in the diagram determines an arrow and these arrows are equal in **C**. If the following diagram commutes than we can say that $g' \circ f = f' \circ g$.

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & Y \arrow[d, "g'"] \\
X \arrow[r, "f'"'] & B
\end{tikzcd}$$

□

Definition A.4 (Monomorphism, Epimorphism and Isomorphism)**.** An arrow $f : B \longrightarrow C$ in a category **C** is said to be a *monomorphism* if, for any pair of arrows $g : A \longrightarrow B$ and $h : A \longrightarrow B$, we have that $f \circ g = f \circ h \Rightarrow g = h$.

$$\begin{tikzcd}
A \arrow[r, shift left, "g"] \arrow[r, shift right, "h"'] & B \arrow[r, "f"] & C
\end{tikzcd}$$

An arrow $f : A \longrightarrow B$ is said to be an *epimorphism* if, for any pair of arrows $g : B \longrightarrow C, h : B \longrightarrow C$, we have that $g \circ f = h \circ f \Rightarrow g = h$.

$$\begin{tikzcd}
A \arrow[r, "f"] & B \arrow[r, shift left, "g"] \arrow[r, shift right, "h"'] & C
\end{tikzcd}$$

An arrow $f : A \longrightarrow B$ is an *isomorphism* if there is an arrow $f^{-1} : B \longrightarrow A$, the *inverse* of f, such that $f^{-1} \circ f = id_A$ and $f \circ f^{-1} = id_B$

$$\begin{tikzcd}
A \arrow[r, shift left, "f"] & B \arrow[l, shift left, "f^{-1}"]
\end{tikzcd}$$

□

Categorial Constructions

Here we present basic categorial constructions that are used in this thesis. Notice that this is not an extensive list, we present only the constructions necessary to our scope. For a more in-depth explanation of Category Theory and its application in Computer Science refer to (PIERCE, 1991).

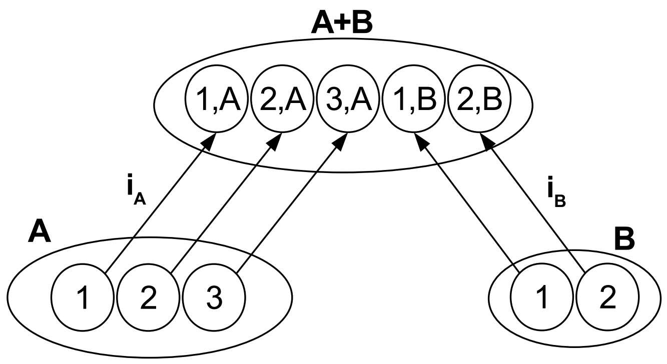
Definition A.5 (Coproduct)**.** Given two objects A and B, their *coproduct* (also called *categorical sum*) is an object $A + B$ and two injection arrows $i_A : A \longrightarrow A + B$ and $i_B : B \longrightarrow A + B$ such that, for any other object X and pair of arrows $f : A \longrightarrow X$ and $g : B \longrightarrow X$, there is one unique arrow $! : A + B \longrightarrow X$ such that the following diagram commutes:

$$\begin{tikzcd}
A \arrow[r, "i_A"] \arrow[dr, "f"'] & A + B \arrow[d, dotted, "!"] & B \arrow[l, "i_B"'] \arrow[dl, "g"] \\
& X &
\end{tikzcd}$$

□

Example A.6 (Coproducts in **Set**)**.** Coproducts can be used to generalize the notion of disjoint union. Figure A.1 shows an example of it in the category **Set**[1]. Having the sets $A = \{1, 2, 3\}$ and $B = \{1, 2\}$ as objects, we have that the set $A+B$ together with morphisms i_A and i_B is their coproduct: all elements of A and B are mapped to $A+B$, no elements from the source objects are identified in the target, and it is possible to find a unique function from $A + B$ to any other candidate satisfying the commutative restriction.

Figure A.1: A coproduct in **Set**



Notice that $(A + B)' = \{(1, 0), (2, 0), (3, 0), (1, 1), (2, 1)\}$ or $(A + B)'' = \{a, b, c, d, e\}$ or any other set with five elements would be equally valid as coproducts for this case. This is due to the fact the categories deal with their objects up to isomorphism, i.e. all this objects have the same format regardless of their internal representations. ■

Definition A.7 (Coequalizer)**.** Given two objects A and B with two parallel morphisms $f : A \longrightarrow B$ $g : A \longrightarrow B$, the coequalizer of the diagram is an object X together with a morphism $h : B \longrightarrow X$ such that $h \circ f = h \circ g$ and, for any other such objects X' with a morphism h', there is a unique morphism $! : X \longrightarrow X'$ such that the following diagram commutes.

$$\begin{tikzcd}
A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B \arrow[r, "h"] \arrow[dr, "h'"'] & X \arrow[d, dotted, "!"] \\
& & X'
\end{tikzcd}$$

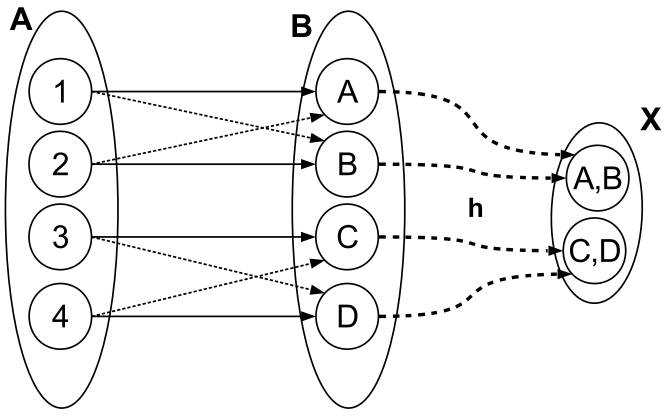
□

Example A.8 (Coequalizers in **Set**)**.** Coequalizers generalize the notion of smallest equivalence relation. Figure A.2 shows the coequalizer for two functions from A to B, let f be the one represented with a solid line and g the one with a dashed line. It is easy to see that the function h from B to X corresponds to the equivalence relation that glues together the items that are identified by the functions f and g. Notice that X does not contain any other

[1]The morphisms are represented in an expanded notation to explicitly show how the mappings were done.

element which is not mapped from B and no element in X was glued together without respecting f and g.

Figure A.2: A coequalizer in **Set**



■

Definition A.9 (Pushout)**.** Given a span of arrows $B \overset{f}{\longleftarrow} A \overset{g}{\longrightarrow} C$, its *pushout* is an object X together with a pair of arrows $f' : C \longrightarrow X$ and $g' : B \longrightarrow X$ such that (1) $f' \circ g = g' \circ f$ and (2) for any other object X' with morphisms $i : B \longrightarrow X'$ and $j : C \longrightarrow X'$ such that $i \circ f = j \circ g$ there is a unique morphism $! : X \longrightarrow X'$ such that $i = ! \circ g'$ and $j = ! \circ f'$.

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & B \arrow[d, "g'"] \arrow[ddr, bend left, "i"] & \\
C \arrow[r, "f'"'] \arrow[drr, bend right, "j"'] & X \arrow[dr, dotted, "!"] & \\
 & & X'
\end{tikzcd}$$

□

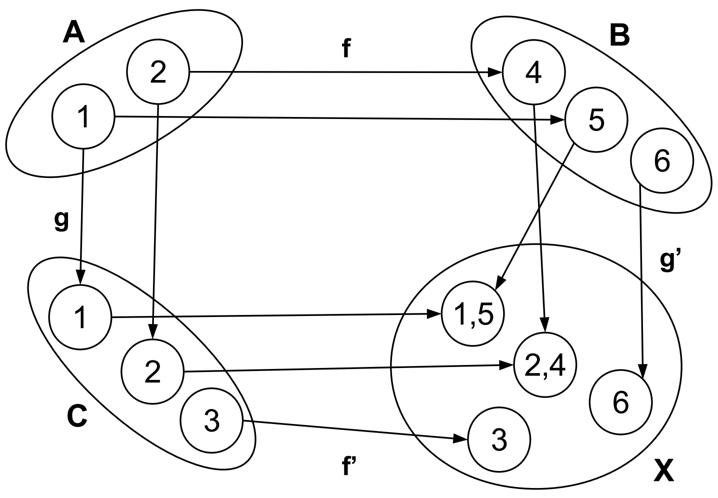
Example A.10 (Pushouts in **Set**)**.** A pushout in **Set** can be seen on Figure A.3. Notice that a pushout maps all elements of sets B and C into set X, "gluing" the ones that are identified via the morphisms $f : A \longrightarrow B$ and $g : A \longrightarrow C$.

■

Definition A.11 (Colimit)**.** Given a diagram D in a category **C**, a *cocone* for D is an object X and a family of morphisms $f_i : D_i \longrightarrow X$ (one for each object D_i in D), such that for each morphism g in D the outer part of the following diagram commutes.

$$\begin{tikzcd}
D_i \arrow[rr, "g"] \arrow[dr, "f_i"'] & & D_j \arrow[dl, "f_j"] \\
 & X &
\end{tikzcd}$$

Figure A.3: A pushout in **Set**



A *colimit* for a diagram D is a cocone $\{f_i : D_i \to X\}$ such that for any other cocone $\{f_i' : D_i' \to X'\}$ there exists a unique morphism $! : X \to X'$ such that the following diagram commutes for every D_i in D.

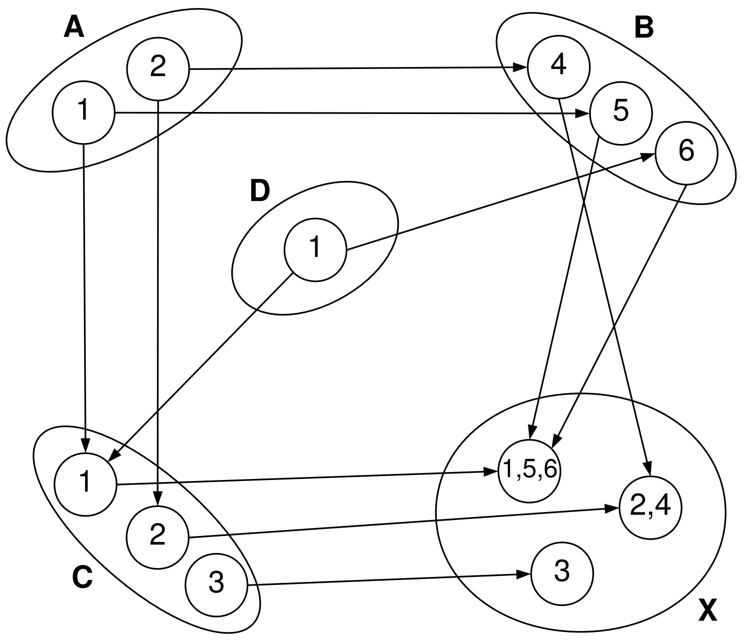
$$\begin{tikzcd}
D_i \arrow[rr, "g"] \arrow[dr, "f_i"'] \arrow[ddr, bend right, "f_i'"'] & & D_j \arrow[dl, "f_j"] \arrow[ddl, bend left, "f_j'"] \\
& X \arrow[d, dotted, "!"] & \\
& X' &
\end{tikzcd}$$

□

Example A.12 (Colimits in **Set**)**.** Colimits generalize several constructions such as disjoint unions, direct sums, coproducts, pushouts and others, where different objects of a diagram are "glued" together in one single object respecting commutativity. All previous examples of coproduct, coequalizer and pushout are special cases of colimits. Figure A.4 shows a colimit for a diagram that can not be calculated in (one step) by any of the previous constructions.

■

Figure A.4: A colimit in **Set**


A
2
1
B
4
5
6
D
1
1
2
3
C
1,5,6
2,4
3
X