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

ANDREI COSTA

Evolution of Negative Application Conditions on Second-Order Graph Rewriting

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

Advisor: Prof. D. Leila Ribeiro
Coadvisor: Prof. D. Rodrigo Machado

Porto Alegre
June 2019

CIP — CATALOGING-IN-PUBLICATION

Costa, Andrei

Evolution of Negative Application Conditions on Second-Order Graph Rewriting / Andrei Costa. – Porto Alegre: PPGC da UFRGS, 2019.

79 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; Coadvisor: Rodrigo Machado.

1. Graph Transformations. 2. Higher-Order Graph Grammars. 3. Negative Application Conditions. 4. Critical Pairs Analysis. I. Ribeiro, Leila. II. Machado, Rodrigo. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Prof[a]. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenadora do PPGC: Prof[a]. Luciana Salete Buriol
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“É chato chegar a um objetivo num instante.”

— Raul Seixas

ACKNOWLEDGMENTS

First of all, I would like to thank my advisors, Prof. Dr. Leila Ribeiro and Prof. Dr. Rodrigo Machado, for their support, persistence and patience during this work, also during my moments of doubt when not only technical advices were needed.

I would like to thank Prof. Dr. Simone Costa and Prof. Dr. Luciana Foss, who guided me in my first studies of graph grammars at UFPel, and who provided me opportunities to continue my studies.

I must thank *Conselho Nacional de Desenvolvimento Científico e Tecnológico* (CNPq), the governmental agency that provided me financial support. As well as *Programa de Pós-Graduação em Computação* (PPGC) and *Universidade Federal do Rio Grande do Sul* (UFRGS) for providing me the facilities to perform this work.

I thank with love to Karine Rui, my wife and great companion. This work is dedicated to you.

A thank-you to my colleagues who shared theirs studies with me, Jonas Bezerra, Guilherme Azzi and Leonardo Marques. The technical (and the non-technical too) discussions, the partnership, the Verigraph implementation, the *happy hours* and the (not always) good jokes were essential to me.

I would like to thank my family, my parents Inacio and Margarete, and my sister Stefani, that have always been strong bases to me.

Finally, my friends and everybody (there are a lot of people!) who I met in this journey. Thank you for the huge support, wise counsels, the moments when we shared problems, but mostly thanks you for the happiest moments outside of my research.

ABSTRACT

Graph grammars are a suitable formalism to modeling computational systems. This formalism is based on rules and data-driven transformations capable of simulating real systems, rules have application conditions and post conditions that can change the system state. Moreover the use of graphs allows an intuitive visual interface essential for the modeler. It is well known that software systems are always evolving, evolutions may range from minor refactorings or bug fixes to major interface changes or new architectural design. The formalization of these evolution processes in graph grammars is done via higher-order principles, which allows programmed higher-level rules to induce modifications on lower-level rules, the system rules. In this work, we extend the current framework of higher-order transformations for graph grammars in order to allow the evolution of rules with negative application conditions. Besides this extension, we provide the first working implementation of the whole framework of higher-order graph grammars in the Verigraph tool enabling the practical usage of this techniques.

Keywords: Graph Transformations. Higher-Order Graph Grammars. Negative Application Conditions. Critical Pairs Analysis.

RESUMO

Gramática de grafos é um formalismo para modelagem de sistemas computacionais. Este formalismo é baseado em regras e transformações de dados capazes de simular sistemas reais, regras tem pré e pós condições de aplicação que podem mudar o estado do sistema. Além disso, o uso de grafos permite uma interface visual intuitiva, que é essencial para o modelador. Se sabe que sistemas computacionais estão sempre evoluindo, essas evoluções podem varias de pequenas refatorações ou correções de problemas, até mudanças maiores em interfaces ou nova arquitetura. A formalização deste processo de evolução em gramáticas de grafos é feita com base em regras de segunda ordem, que possibilitam induzir modificações nas regras da gramática de primeira ordem. Neste trabalho, nós estendemos o *framework* atual de gramáticas de grafos de segunda ordem de forma a permitir evolução de regras com condições negativas de aplicação. Além desta extensão, nós provemos a primeira implementação do *framework* de gramáticas de grafos de segunda ordem na ferramenta Verigraph, possibilitando assim o uso na prática destas técnicas.

Palavras-chave: Transformação de Grafos. Gramáticas de Grafos de Segunda Ordem. Condições Negativas de Aplicação. Análises de Pares Críticos.

LIST OF ABBREVIATIONS AND ACRONYMS

DDD	Distributed Deadlock Detection
DPO	Double Pushout
FOGG	First Order Graph Grammar
GG	Graph Grammar
GTS	Graph Transformation System
GUI	Graphical User Interface
HLR	High-Level Replacement
LHS	Left Hand Side
MUTEX	Mutual Exclusion
NAC	Negative Application Condition
RHS	Right Hand Side
SOGG	Second Order Graph Grammar
SPO	Single Pushout
SYS	System
TG	Type Graph
TGG	Triple Graph Grammar
TR	Token Ring

LIST OF FIGURES

CONTENTS

1 INTRODUCTION

Software is always evolving, usually undergoing many changes during its life cycle. These changes can occur for many reasons: new features, fixing bugs, code reorganization, etc. An important point when dealing with a particular evolution is: how it changes the system behavior. Depending on the evolution purpose, the behavior must be affected or kept unchanged. In fact, the task of predicting the evolution effect is generally very difficult and expensive.

Instead code, it is possible to inspect the evolution over a model of the system. The use of models also allow us to represent just some important part of the overall system, making possible to study the evolution effects for specific parts. This approach can be integrated to Model-Driven Software Engineering (STAHL; VOELTER; CZARNECKI, 2006), where models have a key role in the development of the system.

Graph grammars (GG) are a well suited approach to model complex systems (ROZENBERG, 1997; EHRIG et al., 1999). It is a formalism based on data-driven transformations, and generally, it is used to model parallel and concurrent systems (TAENTZER, 1996). Graph rewriting models are commonly used to describe both model transformations and operational semantics (TAENTZER et al., 2005). Since graphs are a natural way of representing system states, a transformation (an execution) induced by programmed rules (code) that can change a particular graph (system state). Furthermore, this formalism is intuitive since graphs are commonly represented by diagrams. A tutorial introduction to graph transformations in the software-engineering perspective is given in (BARESI; HECKEL, 2002).

The support for the evolution process in GGs was studied in (MACHADO, 2012), which showed that higher-order principles for GG can be adequate in modeling and analyzing systems undergoing programmed modifications. The proposed framework allows to model an evolution as a programmed rule scheme and execute this in a GG. Besides that, a static analysis technique to predict an evolution effect is also introduced. However this theory, called second-order graph grammars (SOGG) was not implemented in any GG tool.

The theory besides GGs is vast, and many engines for graph transformation can be found. In fact, the SOGG as defined in (MACHADO, 2012) is based on a particular kind of transformation. This uses only rules with positive application conditions, which means that a programmed rule can only be applied if some structure do exist. A

very commonly used feature in many GG approaches is to program negative application conditions (NACs), allowing to specify rules that are applied only when some structures do not exist, improving the expressiveness of graph transformation as a modeling technique (LAMBERS, 2010). The lack of support for evolving rules with NACs severely limits the applicability of SOGG.

This work aims to improve second-order graph grammars, by providing a complete implementation of second-order rewriting, and also extending this theory with support for evolution of first-order rules with negative application conditions.

The text is organized in four parts: Chapter 2 introduces the fundamental idea this work refers to by means of examples. Chapters 3 and 4 give the formal foundations of this work; Chapter 5 reviews the studied concepts in the implementation point of view. Chapters 6 and 7 present related work and conclusions. Detailed informations about the chapters are presented below.

Chapter 2 presents two specifications in second-order graph grammars. Both are used to introduce the concepts used in this work. This chapter aims to provide an intuition of first-order graph transformations, graph grammars and second-order rules. This chapter is presented in the modeler point of view, therefore all concepts are given by means of examples and formal definitions are not given here. Furthermore, we present two examples to enrich this theory with new cases of use.

The first specification is from (HECKEL, 1998), where we take a GG that models deadlock detection algorithm and we propose some evolutions in a SOGG specification. This example is to present a SOGG acting over a very common problem in computer science, that deals with concurrent programming issues. The other specification models a Pacman game, it is a classical example in GG. We extended it to SOGG with rules that modify the game mechanics (COSTA; MACHADO; RIBEIRO, 2016). In this grammar we show some interesting situations in the SOGG modeling. Besides that, results of this work are already presented to these grammars in this chapter, in order to clarify its potential. In later chapters these examples and results are referenced.

Chapter 3 reviews the foundations for graph transformations in the DPO approach, giving precise definitions for the rule-based system modifications as presented in Chapter 2. It also presents the formal foundations for second-order graph grammars. The main concepts presented are the rule morphism, which defines a relation between two rules, and the rule rewriting that allows a precise transformation for rules.

Chapter 4 extends the theory of second-order graph grammars by adding support to manipulate rules with negative application conditions. The evolution with NACs concept is introduced, and we show precisely when evolution preserves the semantics of the NACs.

Chapter 5 is a general vision of the developed tool where concepts presented in this work were implemented. This chapter explains why we choose to extend the Verigraph tool (COSTA et al., 2016), and its architecture that supports first- and second-order transformation and analysis.

In the Chapter 6 we discuss the relation of the concepts presented in this dissertation with related work. Finally, Chapter 7 presents conclusions and discusses future work.

2 EXAMPLES AND RESULTS

In this chapter, we introduce some basic concepts about graph grammars (GGs) that are used throughout the text. This chapter aims to explain the modeler point of view when working with GGs, and to discuss how analysis techniques are utilized in a practical way. Thus, we avoid formal definitions and focus on examples and informal descriptions. This chapter also serves as a practical motivation for the following chapters.

Section 2.1 shows a MUTEX model. The basic definitions of GGs are explained together with examples of this grammar (a first-order graph grammar). Moreover, two analysis techniques are presented, the Critical Pairs and Critical Sequences (based on conflicts and dependencies respectively) Finally, the concept of evolution for GGs is presented describing programmed changes on the rules of a rule-based system, and a new conflict analysis technique for this process is introduced.

Section 2.2 presents a Pacman grammar. We introduce the first-order graph grammar relative to this model, and after, we detail a second-order graph grammar (SOGG) modeling a system evolution. The main SOGG concepts are detailed in examples related to this grammar. Still in this section, we explain these second-order rules format (rules that modify others rules), and show how this rules can induce evolutions on graph grammars. Also, we reveal desired operations for SOGGs that are objects of study in this dissertation.

2.1 MUTEX

Because graph grammars are good to model concurrent systems, many common problems in computer science can be instantiated with this formalism. An example that often occurs in operating systems is the resource management problem: it occurs when resources are limited or they can only be accessed by one process at same time, and there are many processes running in parallel in the system. This situation potentially generates conflicts for the resources.

As example for this section, we use a grammar that models a distributed mutual exclusion (MUTEX) algorithm (HECKEL, 1998). This grammar also simulates interactions between *processes* and *resources* in a operating system, in such way that potentially produces the mentioned conflicts. The atomic operations modeled are simple, for example, new *processes* created, a *process* requests a *resource*, a *process* takes a *resource*, etc.

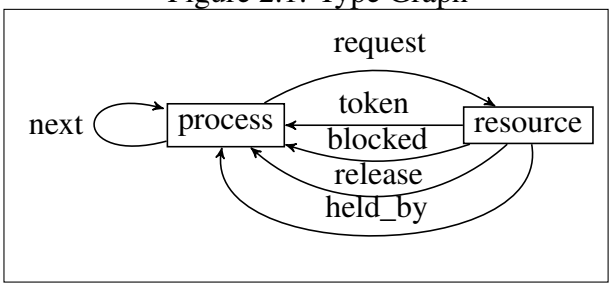
However, their interaction can be very complex, and sometimes they lead to non desirable states, as in the classical example of a deadlock situation.

This grammar was used in other works, for example as a performance benchmark for graph transformation tools in (VARRO; SCHURR; VARRO, 2005).

2.1.1 System State and Type Graph

A system state is represented as a graph, which contains nodes and directed edges linking them. In fact, we use a special kind of graph called typed graph. A typed graph respects a signature, that is a type graph. The type graph contains all potential kind of nodes and edges in that specification. A typed graph over a type graph respects these types constraints.

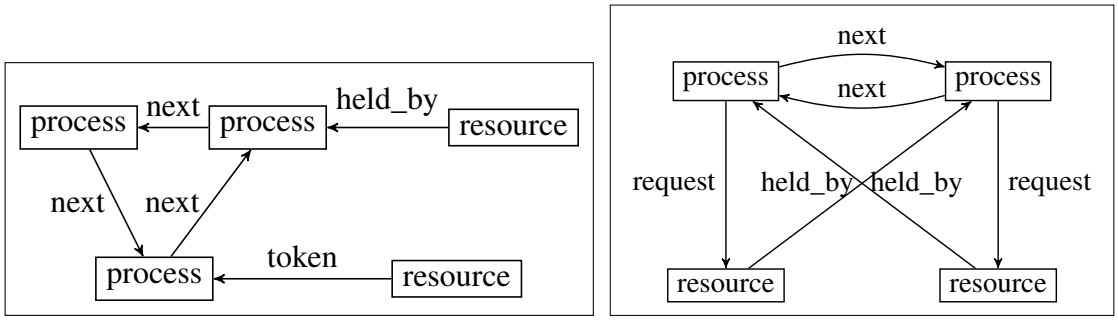
Figure 2.1: Type Graph



When modeling with graph grammars, the type graph is very important for an initial understanding of the model, as a structure diagram. It is because the components of the grammar must respect the same type graph. In our example, the type graph (in Figure 2.1) represents all potential relations between *processes* and *resources*. The only possible node types are *process* and *resource*. The edges have six possible types: *next*, *request*, *token*, *blocked*, *release* and *held_by*. They model different situations: *next*, links two *processes* indicating the next in a circular list; *request*, indicates a *process* requesting a *resource*, note that this is the unique edge type from *process* to *resource*; *token*, it indicates that the *resource* is in possession of that *process*; the other edge types are *release*, *blocked* and *held_by* which explanation is self contained.

Two possible system states are described in Figure 2.2. In 2.2(a) three *processes* form a circular list, one *process* holds the *token* of a *resource*, and another *process* is using a *resource*. In 2.2(b) the system is in a deadlock state. Two *processes*, holding two different *resources*, are requesting the *resource* of each other.

Figure 2.2: Examples of System States as Graphs



(a) Simple State (b) Deadlock State

2.1.2 Rules

This grammar is separated in three viewpoints: the *system* view (SYS) where *processes* and *resources* are created and deleted; the *token ring* view (TR) that distributes the *resources* over the *processes*; and the *distributed deadlock detection* view (DDD) that detects and resolves deadlocks. In each viewpoint the interaction with the system state is different, but all them act over the same graph. Below, we detail each of them, and we show how they are modeled in the grammar.

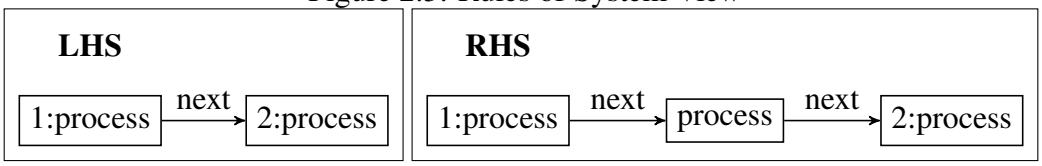
In graph grammars, the actions that potentially change states are modeled as rules. A rule contains pre and post conditions, that may be positive or negative. The positives indicate all elements that must exist in the context, and the negatives indicate all elements that must not exist. The post conditions specify the resulting state of the elements in the transformation. For this work we consider: creation, deletion and preservation of graph elements.

2.1.2.1 System View

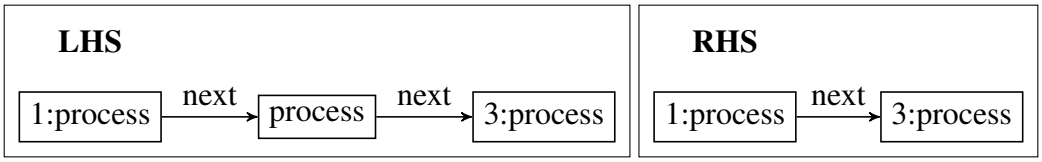
The *system* view is the simplest one, because it models *processes* and *resources* interaction situations, thus it is the best for the initial rule examples. This view models *processes* and *resources* being randomly created and deleted. A *process* always is inserted in a circular list of *processes*, and we assume that some circular list initially already exists. The rule *newRule* is shown on Figure 2.3(a), where the left hand side (LHS) determines the positive preconditions, and the right hand side (RHS), the postconditions. Note that some elements in the both sides have corresponding numerical identifiers, meaning that these elements are preserved by the rule. The elements that are only in the LHS are deleted, and those that are only in RHS are created. Finally, the rule *newRule* creates a

new *process* between two already existent *processes* and the linking of them. As a reverse of above, the rule *killRule*, in Figure 2.3(b), deletes a *process* in a list and remount the linking between the remaining ones.

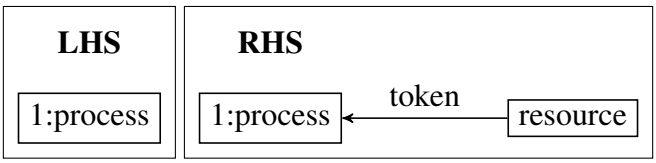
Figure 2.3: Rules of System View



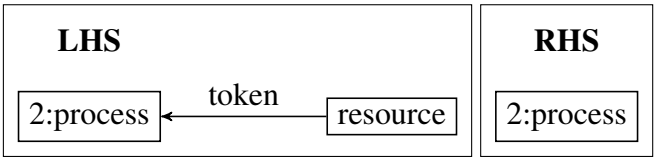
(a) newRule



(b) killRule



(c) mountRule



(d) unmountRule

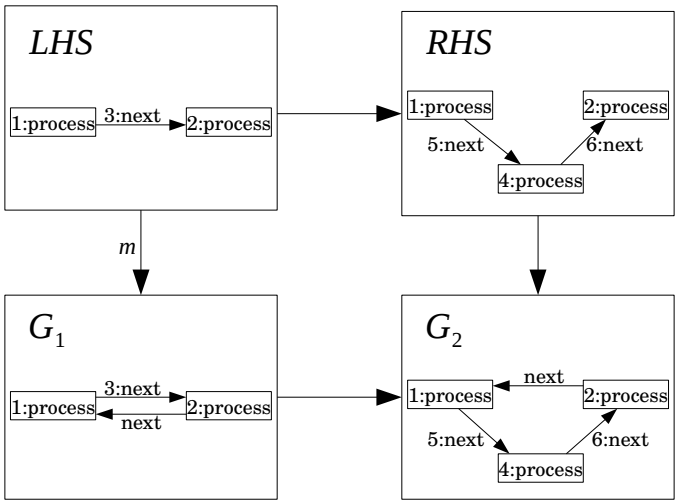
A *resource* is always linked to a *process* via a token edge, even when this *process* is not using it, that indicates just a reference. A *resource* with only one incident *token* edge is called idle. The rule *mountRule*, in Figure 2.3(c), creates an idle *resource* linked to a *process*. The rule *unmountRule*, in Figure 2.3(d), deletes an idle *resource* linked to a *process*.

This four simple rules model a working computational system, since it is used just as a basis for the MUTEX algorithm, we maintain this view in a high level of abstraction. Note that does not matter why *processes* and *resources* are being created and deleted, in this grammar it is only important that these events eventually occur.

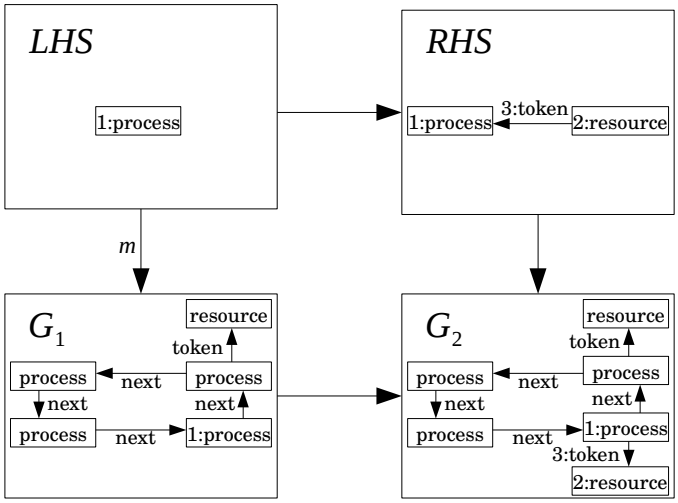
Rules model potential transformations on a graph, but the existence of a transformation depends of another element, called match. A match indicates a part of the graph where a rule will be applied. In a practical way, besides having a rule, we must find in the graph a region that satisfies the rule preconditions. In order to satisfy the positive pre-

conditions, any region of the graph which is identical to the LHS of a rule is sufficient (in fact we will explain in the next chapters that depending of the rewriting system additional constraints are needed). The set of negative preconditions are verified in such way: if the match satisfies the positive preconditions then verify if each negative precondition also is satisfied. More details are given when rules with negative conditions appear.

Figure 2.4: Transformation Examples



(a) Transformation 1



(b) Transformation 2

Figure 2.4 shows two examples of transformations. Figure 2.4(a) utilizes the rule *newRule* ($LHS \rightarrow RHS$). The match (m) relates the left hand side of *newRule* (LHS) with an arbitrary graph (G_1), where numerical identifiers were added to help the identification of the matched elements. Note that this match covers almost the graph G_1. Finally, the transformation removes the *3:next* edge and adds the new *4:process* with two *(5,6):next*

edges linking it with other nodes in graph G_2. Note that elements out of the match are always preserved[1].

Figure 2.4(b) utilizes the rule *mountRule* ($LHS \rightarrow RHS$). Only a small part of the graph G_1 is covered by the match (m). Note that this rule could be applied to any *process* in G_1, we arbitrary choose *1:process* as matched element. The transformation adds an idle *2:resource* on this node without removing anything.

Considering a space state exploration, and starting from a typed graph with an initial circular list of *processes* with at least two elements, the rules of this view have the power of generate circular lists of *processes* with arbitrary size, each *process* linked with an arbitrary number of idle *resources*. If the initial state is a typed graph with just one *process*, we will be able to add an arbitrary amount of *resources* on it. However, if the starting typed graph does not have any *process* node, no transformation can occur because no rule will have its preconditions satisfied.

2.1.2.2 Token Ring View

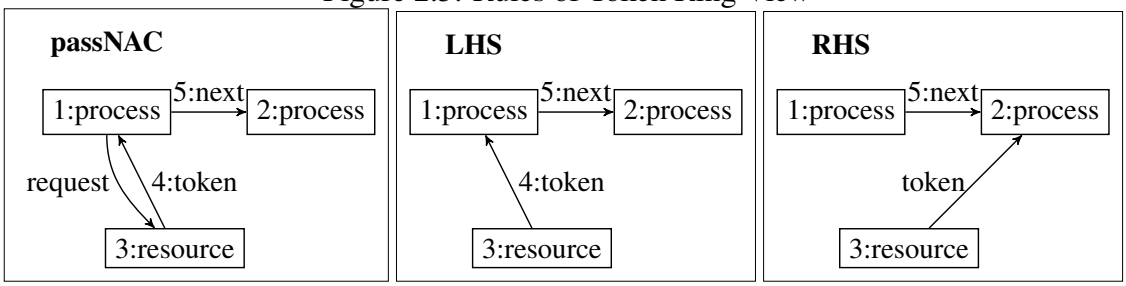
The Token Ring View is responsible for the scheduling of the *resources* over the *processes*. A *process* can request a *resource* at any time. While a *resource* is not in the requested *process*, it walks through the list of *processes*. When a *process* has a requested *resource*, it holds it for some time, disabling the movement of the *resource*. When it eventually releases it, the *resource* is able to walk by the list of *processes* again.

The passRule, in Figure 2.5(a), passes the *resource* between two linked *processes*. Although, the first negative precondition appears. It indicates that this rule is unable to be executed if the *resource* that will be passed is requested by *1:process*. It is modeled as a NAC, that is, a graph which contains all the left hand side of a rule plus the forbidden elements; in this case a *request* edge from *1:process* to *3:resource*. Note that the numerical identifiers are used in the NAC to relate their elements with the LHS elements.

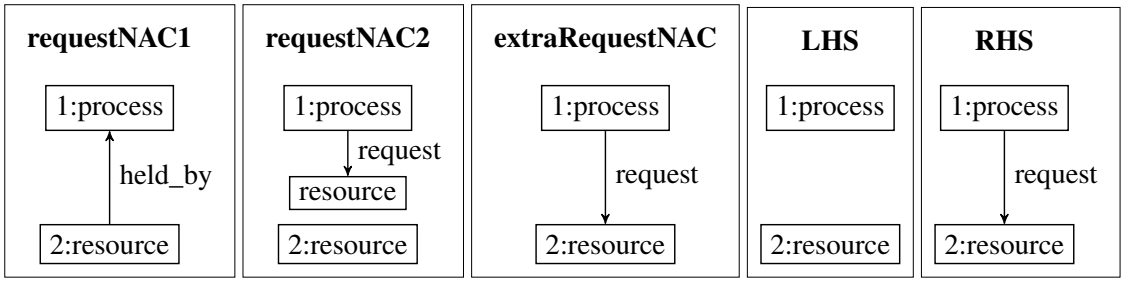
The requestRule, in Figure 2.5(b), models a *process* creating a new request for a *resource*. This rule can not be applied in three situations that are represented by their negative preconditions: when the *process* already holds the *resource* to be requested; when the *process* already requests another *resource*; and when the *process* already requests the same *resource*. Out of this three situations, a *process* can (non deterministically) request an existent *resource*.

[1] In some other approaches such as SPO and SqPO these elements could be delete in specific cases, this situation is called side effect.

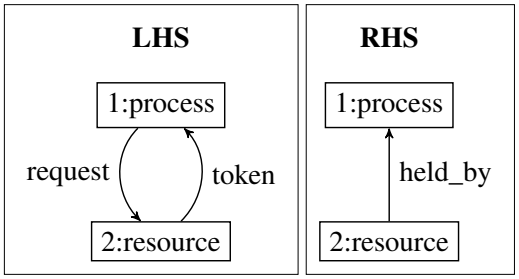
Figure 2.5: Rules of Token Ring View



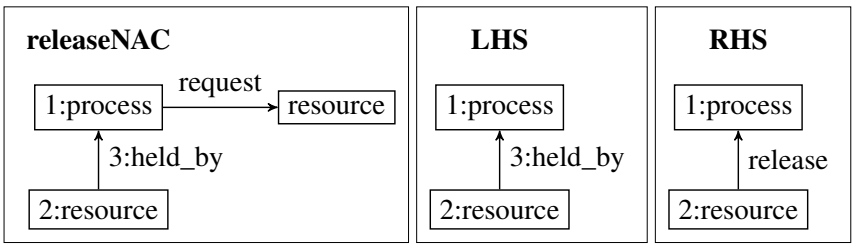
(a) passRule



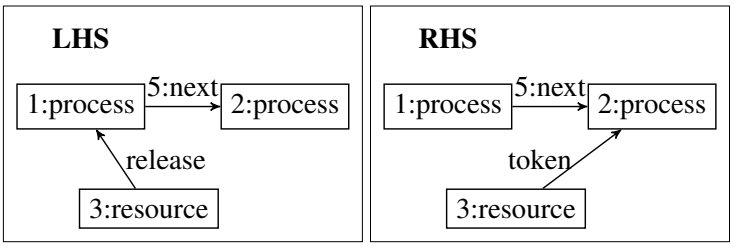
(b) requestRule



(c) takeRule



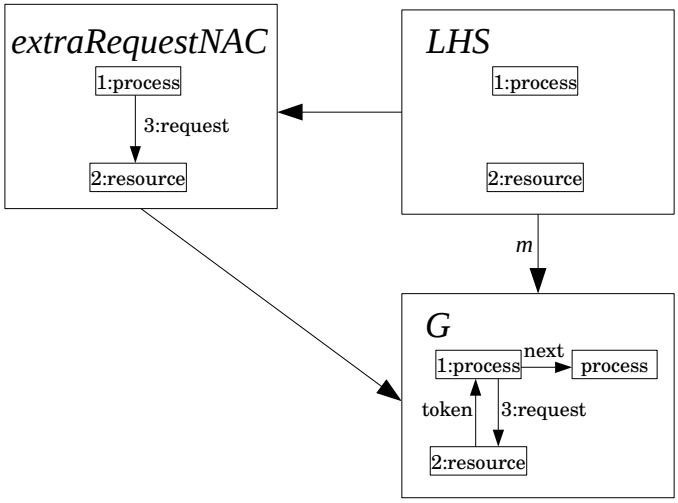
(d) releaseRule



(e) giveRule

Figure 2.6 shows an example of a NAC disabling a transformation. The rule requestRule has three NACs, but in this example we choose to focus only on *extraRequestNAC*. This rule has a match m in the graph G, however since there is in G

Figure 2.6: NAC Disabling Transformation



a *3:request* edge between *1:process* and *2:resource* then there exist a morphism from *extraRequestNAC* to G. The transformation for this match m can not occur because there is a morphism for (at least one of) the negative conditions from the *LHS* of the requestRule.

The takeRule, in Figure 2.5(c), models a *process* that requests and has token of the same *resource*. This *process* gets the *resource* as indicated by the edge hold_by. In the releaseRule, in Figure 2.5(d), a *process* holding a *resource* releases it. However, this rule is not applicable when this *process* is requesting another *resource*. The giveRule, in Figure 2.5(e), models a released *resource* being placed in the circular list of *processes*.

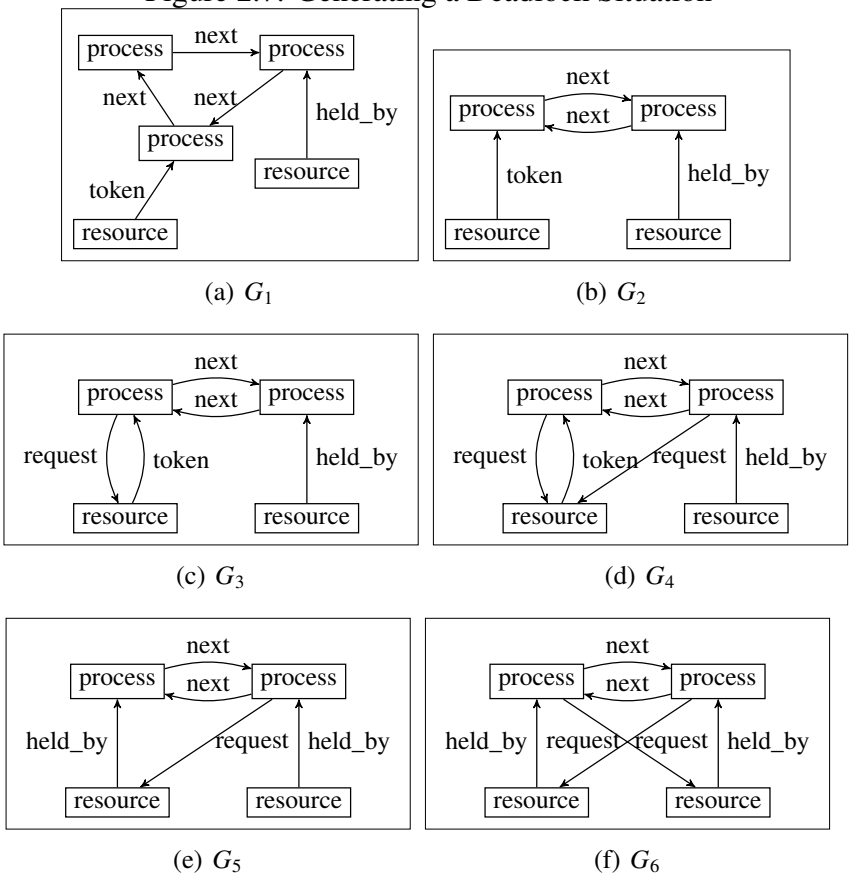
This view of the system is able to distribute *resources* over *processes* in a circular list of them. Instead of System View, the Token Ring View can generates many complex states, once many *processes* are requesting many *resources*. In particular, system states with deadlock can be generated, as detailed below.

A possible path to transform the graph of the Figure 2.2(a) in a graph with deadlock (Figure 2.2(b)), is by applying a specific sequence of rules (with specific matches) as Figure 2.7 shows. Consider the first graph as G_1, the deadlock graph as G_6 and the application of a rule r as: $\overset{r}{\rightarrow}$. A possible sequence of rule applications that leads to this transformation is $G_1 \xrightarrow{killRule} G_2 \xrightarrow{requestRule} G_3 \xrightarrow{requestRule} G_4 \xrightarrow{takeRule} G_5 \xrightarrow{requestRule} G_6$. We omit the matches due simplicity of information in the figures.

2.1.2.3 Distributed Deadlock Detection View

The Distributed Deadlock Detection View operates detecting and broking deadlocks in the system. This view adopts the following approach: it detects a potential dead-

Figure 2.7: Generating a Deadlock Situation



(a) G_1 (b) G_2

(c) G_3 (d) G_4

(e) G_5 (f) G_6

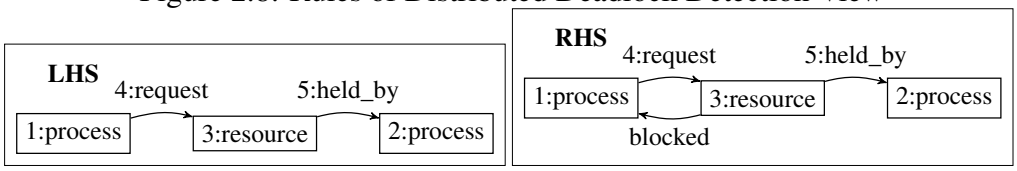
lock, this detection triggers a search operation to verify if it is a real deadlock or not and, in the positive case, the *resource* blocked is released.

This view is modeled with the four rules presented in Figure 2.8. The *blockedRule*, in the Figure 2.8(a), acts sending a *blocked* message (an edge) for any *process* that requests a *resource* with *held_by* with other *process*. However, sometimes this situation does not generates a deadlock, in fact we can discard all *blocked* edges for *processes* that do not hold any *resource*. For this reason *ignoreRule* (in Figure 2.8(b)) remove the *blocked* edge for the *process* that do not hold any *resource*. The *waitingRule* passes the *blocked* edge for another *process*. The *unlockRule* resolves the deadlock situation by releasing the *resource*.

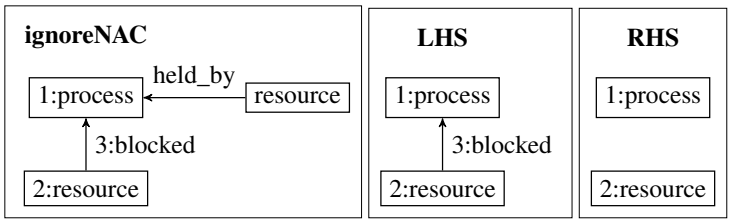
2.1.3 Analysis

The usage of graph grammars as a formal language leads to precisely defined transformations, also, the visual representation of the system modeled, as well as the system simulation though graph transformations, are important features in this area. Since

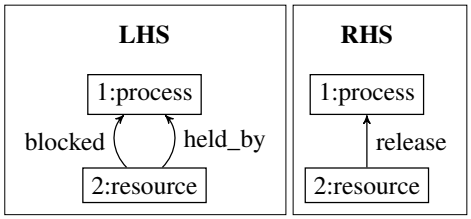
Figure 2.8: Rules of Distributed Deadlock Detection View



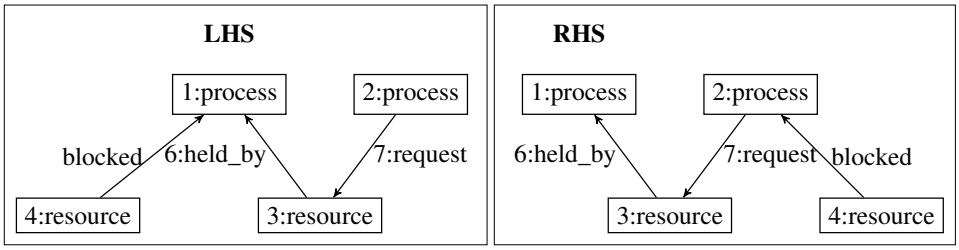
(a) blockedRule



(b) ignoreRule



(c) unlockRule



(d) waitingRule

their are based on formal foundations, formal analysis techniques can be performed with visual appeal. This analysis can be used to prove important grammar properties, which is particularly important for modeling safety-critical systems.

Many formal analysis are defined for GGs, each of them usually tries to capture specific behaviors of the system. We focus on Critical Pairs and Critical Sequences analysis, which identify if two rules are in conflict and dependency, respectively.

2.1.3.1 Conflicts

Two transformations are in conflicts when the application of one of them disables the application of the other. The conflict situation highlights a decision point in a graph transformation system, which is if a rule is applied then the other is not possible to be applied in that same context. Some transformations are desirable to be in conflict, while

others are not. It is possible to check if two rules have some potential transformations that are in conflict, and also ensure the conflict-free situation. The technique that finds all this potential conflicts is called Critical Pairs Analysis.

The first important point here is explain when two transformations are in conflict. A conflict can have many causes, however they are generally classified in two groups, that are detailed with examples below. The simplest cause is called *delete-use*, as explained in Figure 2.9(a). It occurs when an initial graph has two possible transformations, and on of them deletes an element used by the other. In this example, we have an *Initial Graph* where two rules are possible to be applied: the first (*takeRule*), intends to ensure that the *process 1* held the *resource 2*; the second (*waitingRule*), tries to pass the *blocked resource 6* through the *processes* (from *5* to *1*). This conflict cause says that, in this state (or in a bigger state, but that contains this subgraph) if we apply *takeRule* then *waitingRule* is unable to be applied, and this information must be passed to the modeler to verify the consistency.

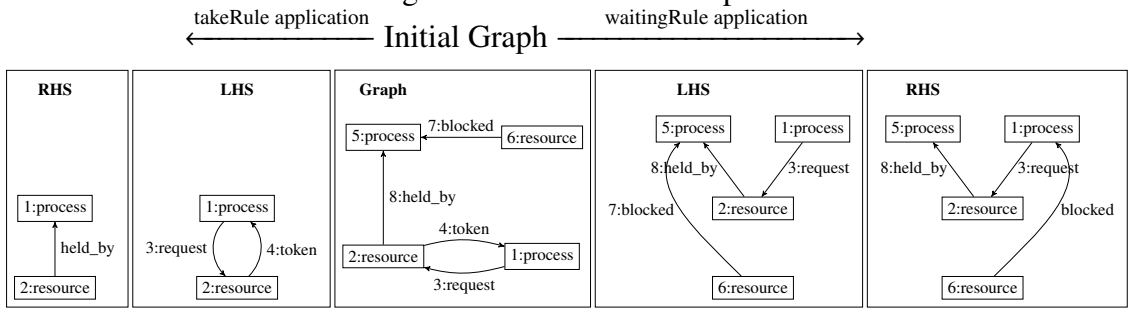
The second cause is the *produce-forbid*, as explained in Figure 2.9(b). It occurs when after a transformation, the state (middle) graph has some context forbidden by the other transformation, since this does not exist before the first transformation. In the context presented in this work, this is only possible when the first rule creates something that the NAC of the second forbids. In the example, the rule *requestRule* was applied in a graph, only the resulting state graph (*Middle Graph*) is showed. This transformation turned the application of the rule *passRule* impossible due the NAC of this rule. In fact, the *process 1* had the token of the *resource 2*, before the application of *requestRule*, it was not requesting this *resource* and then it was able to pass to the other *process*. However, when *requestRule* was applied the *process 1* turns to request the *resource 2*, and then it can not pass anymore.

The Critical Pair Analysis technique constructs all potential conflicts, which are showed in a table that contains all pairs of grammar rules, and the number of potential conflicts between them. Figure 2.10(a) shows this table for the Mutex grammar. Any real conflict that can occur in the grammar has its respective potential (and minimal) conflict in this table.

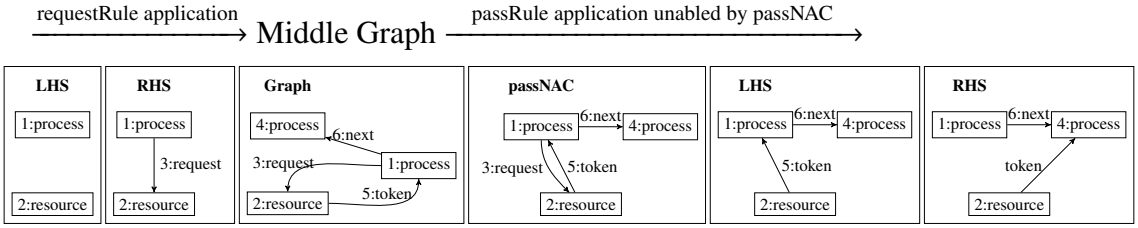
2.1.3.2 Dependencies

The dependency concept is analogous to the conflict, but considering sequential rather than parallel transformations. It captures situations where two transformations are

Figure 2.9: Conflicts Example



(a) Delete Use



(b) Produce Forbid

only possible to occur in sequence. Differently of the conflicts, the dependencies (due to sequentially) basically raise from four causes. We split them in two kinds: triggered and irreversible.

The triggered dependencies occurs when a transformation enables other transformation (that necessarily was disabled). They have two types: the produce-use, when a transformation creates an element utilized by the other transformation; and the delete-forbid, when a transformation deletes an element that was forbidden by some NAC of the other transformation, turning it applicable. The Figure 2.11(a) shows a triggered dependency, the second rule is able to be applied in that *process* because the first transformation created the *token* edge.

The irreversible dependencies occurs when after two sequential transformations, the first one can not be undone. Also, this dependency has two types: the deliver-delete, when the second transformation deletes something utilized by the first; and, the forbid-produce, when the second transformation produces something that is a NAC of the first. The Figure 2.11(b) shows an irreversible dependency, in the *Middle Graph* the first rule was applied, if the second rule also be applied then the first rule could not be applied because the second created the *request* edge.

The Critical Sequence analysis is very similar to the Critical Pairs Analysis. All potential dependencies are generated for all possible pairs of rules. Figure 2.10(b) shows the Critical Sequence table for the Mutex grammar. These dependencies can be of any type, in fact these details in actual tools can be verified one-by-one in theirs graphical

Figure 2.10: Results of Critical Pair Analysis and Critical Sequence Analysis

\	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13
R1	1	2	0	0	1	0	0	0	1	0	0	0	0
R2	2	5	1	0	1	1	0	0	1	0	0	0	0
R3	0	1	0	0	0	0	0	0	0	0	0	0	0
R4	0	0	0	1	1	2	0	0	0	0	0	0	0
R5	0	1	0	1	3	0	0	0	0	0	0	0	0
R6	0	1	0	2	1	2	0	1	0	0	0	0	0
R7	0	0	0	0	0	0	3	0	0	1	1	1	0
R8	0	0	0	0	0	0	0	1	0	1	1	0	1
R9	0	1	0	0	0	0	0	0	3	0	0	0	0
R10	0	0	0	0	0	0	0	0	0	0	0	0	0
R11	0	0	0	0	0	0	0	0	0	0	8	0	1
R12	0	0	0	0	0	0	0	0	0	0	0	1	1
R13	0	0	0	0	0	0	0	1	0	1	2	1	3

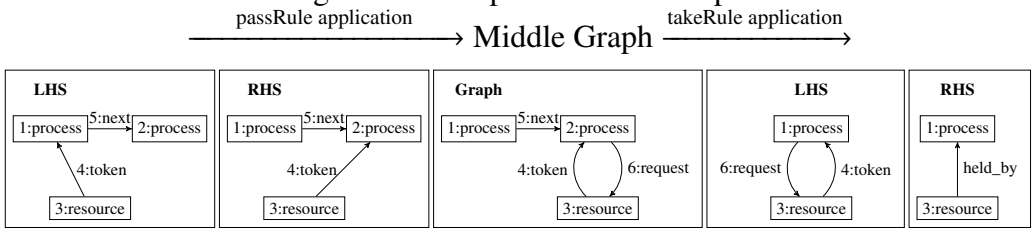
(a) Critical Pairs Analysis

\	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13
R1	4	10	2	0	2	2	0	0	2	0	0	0	0
R2	2	4	0	0	1	0	0	0	1	0	0	0	0
R3	0	0	0	2	2	4	0	0	0	0	0	0	0
R4	1	1	0	0	0	0	0	0	0	0	0	0	0
R5	1	2	0	2	4	1	2	0	0	0	0	0	0
R6	0	0	0	0	0	0	2	0	0	1	1	0	0
R7	0	0	0	0	1	1	0	3	0	1	1	0	2
R8	0	0	0	0	0	2	0	0	2	0	0	1	0
R9	1	2	0	2	4	0	2	0	0	0	0	0	0
R10	0	0	0	0	0	0	1	1	0	0	4	2	3
R11	0	0	0	0	0	0	1	1	0	0	8	2	3
R12	0	2	0	4	0	0	1	0	0	0	0	0	0
R13	0	0	0	0	0	1	0	0	2	0	0	1	0

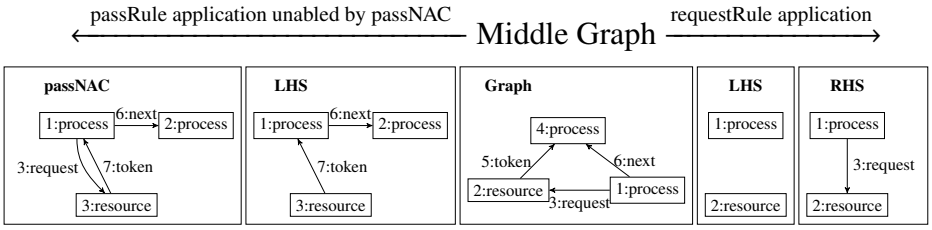
(b) Critical Sequences Analysis

R1: newRule; R2: killRule; R3: mountRule; R4: unmountRule; R5: passRule; R6: requestRule; R7: takeRule; R8: releaseRule; R9: giveRule; R10; blockedRule; R11: waitingRule; R12: ignoreRule; R13: unlockRule

Figure 2.11: Dependencies Example



(a) Triggered (Produce Use)



(b) Irreversible (Forbid Produce)

interface.

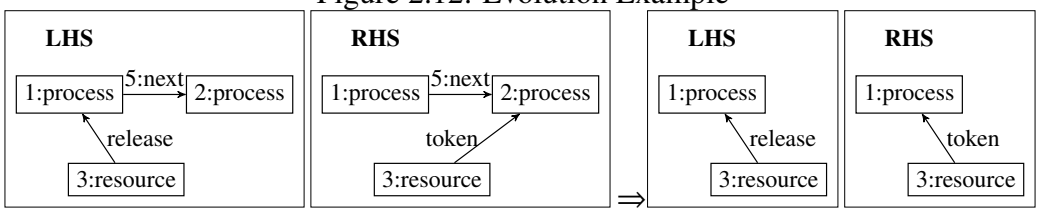
2.1.4 Evolution

The sections above follow the classical GG theory, hereinafter the concepts are part of the SOGG theory based in (MACHADO, 2012). The basic concept here is called evolution. An evolution can be any modification in a graph grammar, however we are not interested in the system state graph and type graph modifications. The evolution that we study must captures a modification in the set of grammar rules. For simplicity, we analyze modifications for each rule once. In this sense, any modification of a single rule is an evolution.

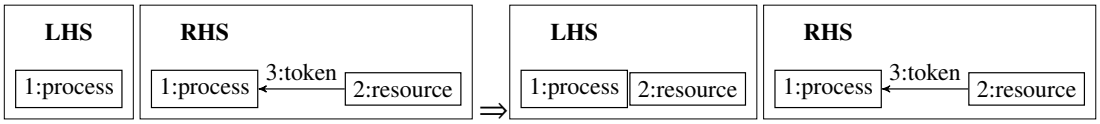
Evolutions can be done manually by a modeler in any graph grammar editor tool, however to study and model this situations opens the opportunities to develop analysis over this process. Predict desirable (or not) changes in the transformation system by analysis over evolutions is an interesting perspective. In Section 2.1.4.1 we present an analysis technique developed to capture conflicts in the evolution step.

In order to clarify this concept we give two examples of evolutions, the figures present in the left side the old rule and in the right side the evolved rule. The first, in Figure 2.12(a), could be inserted in a refactoring process of a model, because it is not trying to change the overall system behavior. This evolution modifies the rule *giveRule*, the modification inserted affects when a *process* releases a *resource*, before evolution the *resource* is released and passed to the next *process*, after the evolution the *resource* is not

Figure 2.12: Evolution Example



(a) *giveRule* Evolution



(b) *mountRule* Evolution

passed, the token remains in the *process* that performs the release.

The second, in Figure 2.12(b), changes more deeply the behavior of the rule, consequently of the grammar too. This evolution supposes a computational system where *resources* are not created randomly. In this case the rule *mountRule* can not create the node *resource*, it just attaches an existent node to a *process*. This modification aims to model a system that works with limited and pre-instantiated resources.

2.1.4.1 Inter-level Analysis

Previous sections presented graph grammars, their graph transformations induced by rules, and rule evolutions. An interesting analysis point emerges when two layers interact. The analysis over these context are called inter-level analysis (MACHADO; RIBEIRO; HECKEL, 2015). Specifically we deal with inter-level conflict analysis main purpose is to raise all situations where an evolution disables the applicability of the evolved rule in a graph. In this chapter we focus only in the results of this analysis.

In this section, we shown two examples of this analysis that are related to the evolutions in the Figure 2.12. The first is in Figure 2.12(a), this evolution is a refactoring, then we initially suppose that does not change the overall system behavior. The utilization of the inter-level conflict analysis can be used as part of this verification process. In fact, the analysis result is that the evolved rule is applicable on all situations that the non evolved rule was. Then, this evolution is conflict-free in inter-layer terms.

The *mountRule* evolution, in Figure 2.12(b), has a different behavior. It adds a new element in the pre condition of the rule, so it is potentially a conflict. In fact, this addition on the pre condition turns this rule not applicable when there is not an existing *resource*. A minimal situation of this is showed in Figure 2.13, where there is a graph

Figure 2.13: Inter-level Conflict Graph Example

that, the rule before the evolution has a match, and after the match does not exist. This cases are considered inter-level conflicts.

2.2 Pacman

The section introduces an example of second-order graph grammars: the Pacman game. Naturally, we need to present the first-order Pacman grammar, and after we explain the concepts of the second-order with the Pacman SOGG.

We start explaining the format of the Pacman rules, which are presented here in the DPO format, unlike the MUTEX rules. The DPO format requires three graphs to form a rule (L, K and R), L and R are the pre and post conditions, as in the MUTEX rules, and the K graph (also called: interface graph) has the preserved elements. This format divides the transformation in two parts, deletion (from L to K) and creation (from K to R). For this examples, this rule format does not change the behavior of the transformation, then all concepts of the section above can be reused.

2.2.1 First-Order Graph Grammar

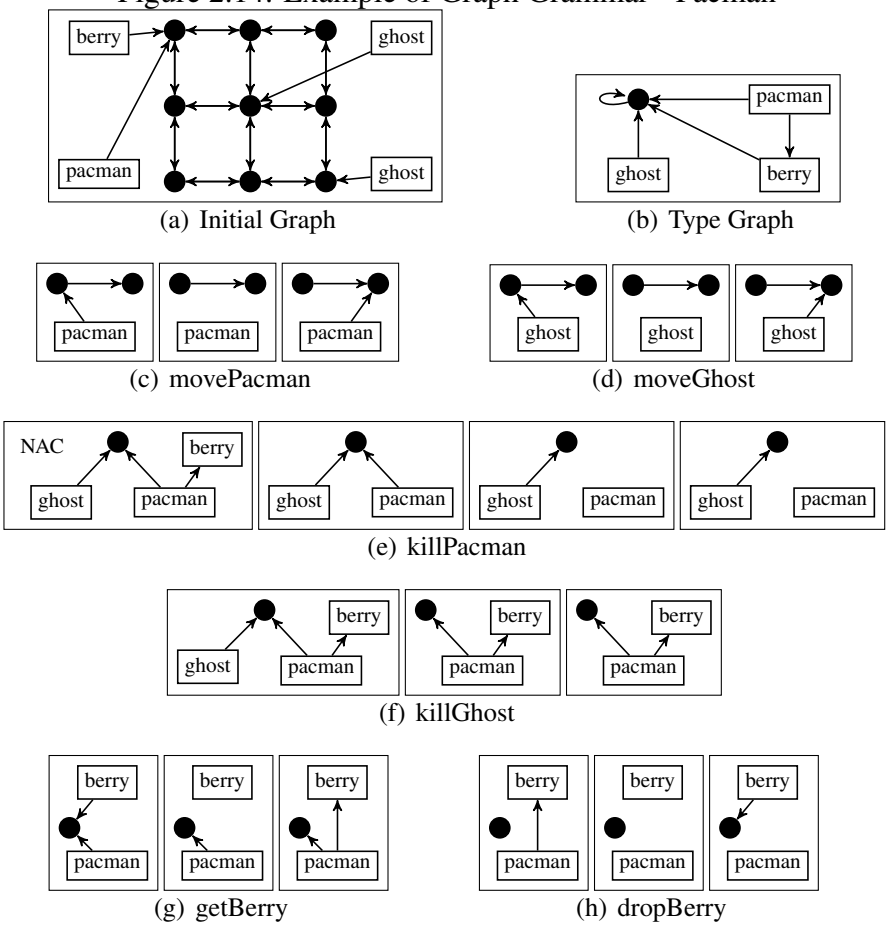
First-Order Graph Grammar is a name that we use to differentiate the "level" of the grammar, the first-order is used as synonymous of the (Typed) Graph Grammars as introduced before in this work.

A simplified version of the Pacman game is modeled with graph grammars in Figure 2.14. The type graph 2.14(b) allows four kind of nodes: *ghost*, *pacman*, *berry* and *block* (filled circle). The edges indicate the position of the elements: *ghosts*, *pacman* and *berries* can be found in *blocks*, and *pacman* can carries *berries*. The initial graph, in 2.14(a), is an arbitrary typed graph specifying the initial state. The set of rules is: {*movePacman*, *moveGhost*, *killPacman*, *killGhost*, *getBerry*, *dropBerry*} as shown in the Figures 2.14(c...h), each of them depicted by its span of typed graphs (L, K and R, in this

order).

In this simplified Pacman game, *ghosts* and *pacmans* move freely over the *blocks*, according to rules 2.14(c) and 2.14(d). The *pacman* can obtain a *berry*, and occasionally drop it in anywhere, as shown in rules rules 2.14(g) and 2.14(h). When a *ghost* and a *pacman* are on the same *block*, two rules may be applied. If the *pacman* has a *berry* then it kills the *ghost*, rule 2.14(f). Otherwise, the *ghost* removes *pacman* of the game, rule 2.14(e). Notice that rule *killPacman* requires to check if *pacman* does not have a *berry*, and therefore a NAC is necessary.

Figure 2.14: Example of Graph Grammar - Pacman



2.2.2 Second-Order Graph Grammar

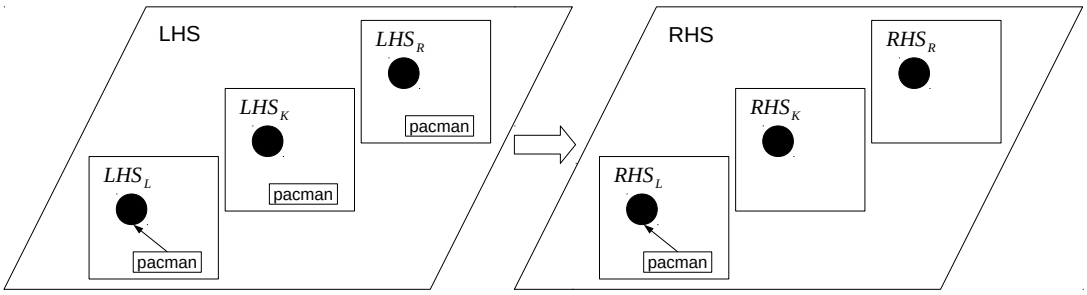
Second-Order Graph Grammars (SOGGs) models, evolutions on first-order graph grammars. A second-order state is a set of first-order rules, and a second-order rule models a potential modification in a first-order rule. For simplicity, we work with a first-order rule being modified at time. In this case, a second-order rule models a modification of a

single first-order rule.

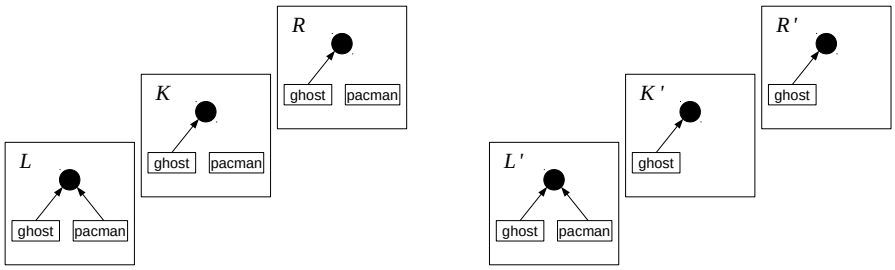
We present two examples of second-order rules, the first is used to exemplify the second-order transformation, and the second to illustrate an open problem that motivates this dissertation.

2.2.2.1 Evolution without NACs

Figure 2.15: Second-Order Examples



(a) *violence* second-order rule



(b) *killPacman* first-order rule

(c) *killPacman* first-order rule evolved

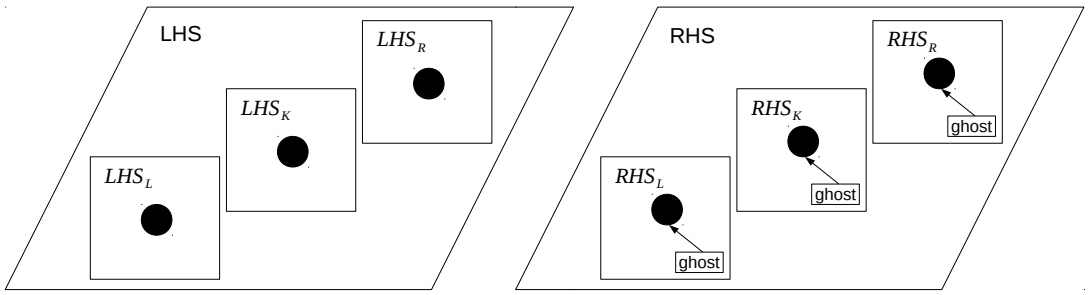
The first example is named *violence* rule in Figure 2.15(a), it acts on first order rules that removes a *pacman* of the *block*, it changes its behavior to a deletion of the *pacman*. Therefore, its pre condition is a first order rule that deletes an edge from *pacman* to *block*, as showed in the LHS rule (LHS_L, LHS_K and LHS_R graphs). The RHS adds a deletion of the node *pacman*, then it must contain this element only in RHS_L, and not in RHS_K and RHS_R graphs.

A second-order match occurs when the pattern of the rule is founded in the state, in this example for each *pacman* removed of the *block* by some rule, then there is a match. Figure 2.15(b) shows a first-order rule such that there is a match from *violence* to *killPacman* (we omitted the NAC for simplicity). This match leads to a second-order transformation, that is, an evolution. The evolved rule is showed in Figure 2.15(c), the

evolution generate a rule that deletes the *pacman* node.

The second example is a second-order rule named *addOnePreservedGhost*. This rule adds a preserved *ghost* to a *block* in any rule that already preserves this *block*. In the pre condition there is a rule that preserves a *block*, and as pos condition it adds a new preserved *ghost* on the same *block*. Figure 2.16 shows this second-order rule.

Figure 2.16: *addOnePreservedGhost* rule



As in FOGGs, a rule can be applied in different areas of the current state graph. Similarly, in SOGGs second-order rules can be applied in different areas of first-order rules. As *addPreservedGhost* rule has an easily satisfied pre condition, it can be applied to any first-order rule of the Pacman grammar, furthermore in rules *movePacman* and *moveGhost* there are two different matches on two *blocks* each where the transformation can take place.

2.2.2.2 Evolution with NACs

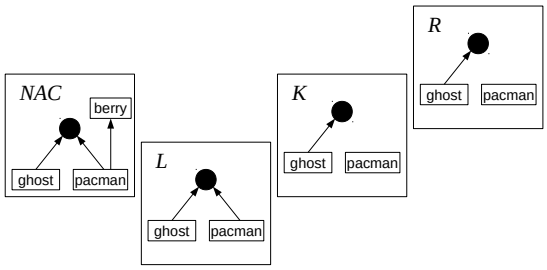
A second-order rule may trigger an evolution on a first-order rule. But if the rule to be evolved has NAC, the theory must define how evolution of the NAC occurs. In (MACHADO, 2012) NACs were not addressed in the evolution because the complexity of preserve NACs semantics, otherwise that work aimed on structure a higher grammar level.

In this work, we consider NACs starting from the left side of the rules, although it is possible to extend this process to be used for NACs from the right side of the rules. Because this, if the left side is not affected by the evolution, then the NAC will be maintained. This is the reason why in Figure 2.15 we omitted the NAC.

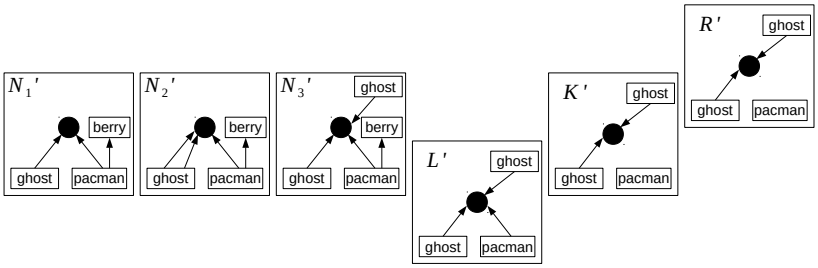
Using the second-order rule *addOnePreservedGhost*, we may evolve the rule *killPacman* with its NAC. There is just one possible match. The evolved rule will have a new preserved *ghost* node. But this rule has a NAC, that forbids its application when the

pacman has a *berry*, and once the evolution has changed the left-hand side, the NAC must be evolved too.

Figure 2.17: Example of Evolution with NACs



(a) *killPacman* with NAC



(b) *killPacman* with NAC Evolved

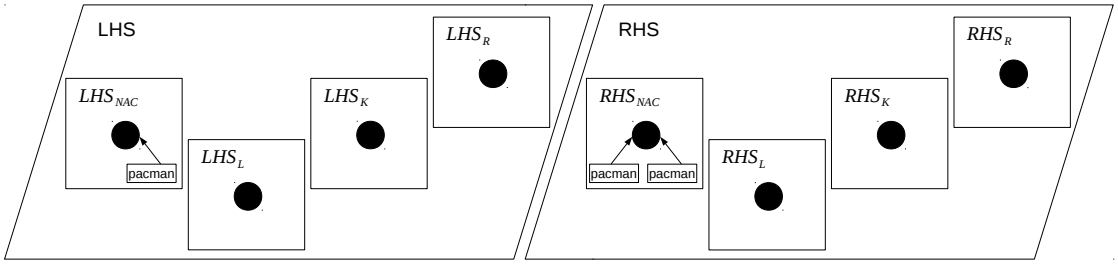
We propose to evolve the NAC using as basis the evolution of the left-hand side of the first-order rule, Figure 2.17 shows an example of this evolution. The *addPreservedGhost* rule is modifying the *killPacman* rule by adding a preserved *ghost*. As the NAC was evolved using as basis the transformation $L \rightarrow L'$, the evolved NACs carry the information of the new *ghost*. A set of NACs was generated because there is a NAC for each match possibility where the NAC N could exists. In the case of L' having less elements than L the idea is the same, these missing elements must be deleted also for the evolved NAC.

2.2.2.3 Manipulating NACs

The section above exemplified NACs transformation on rule evolutions. However the NACs was modifications are induced only by the rule evolution itself (the left-side evolution), another approach is to model arbitrary changes on the set of NACs. In this section we briefly discuss how we can model first-order NAC modifications on second-order rules.

Originally first-order rules can be modified by second-order rules, but the second-order rules have a “limited” expressiveness since they can not model changes in NACs. We wish to define second-order rules that can modify the set of NACs arbitrarily.

Figure 2.18: Example of Second-Order Rule with NACs



An example of what we propose is shown in Figure 2.18, where there is a second-order rule, the non NAC part of the rule is preserved (but potentially could be modified). However, this rule models a modification on the set of NACs, in LHS_{NAC} the NAC forbids one *pacman* in the *block*, after the evolution the new NAC (RHS_{NAC}) forbids two *pacmans* on the same *block*.

Since LHS has NACs, and it being the pre condition of a second-order rule, these NACs are now part of the match of second-order. This work does not develop support to this kind of transformations, however initial experiments was made and it is clearly a future work.

3 FIRST- AND SECOND-ORDER GRAPH TRANSFORMATIONS

In this section, we review the basic concepts of the algebraic Double Pushout Approach (DPO) for graph transformations including negative application conditions (EHRIG et al., 2006) and then present the main concepts of Second-Order Graph Transformation (SOGTs) as defined in (MACHADO; RIBEIRO; HECKEL, 2015), a framework that allows to define rules, called second-order rules, to modify other rules, called first-order rules. The current restriction of SOGTs is that first-order rules may not have NACs. In the next section an extension to handle NACs in SOGTs will be provided. To illustrate the concepts throughout the paper a simple Pacman game example is used.

3.1 The Double-Pushout Approach

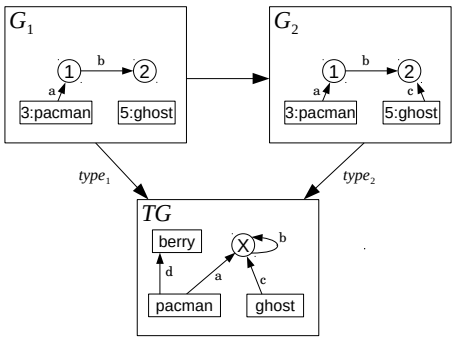
The algebraic approaches for graph transformation use categorical operations in order to perform the transformations defined by the rules (EHRIG; PFENDER; SCHNEIDER, 1973; EHRIG, 1978). The approach we follow uses two *pushouts* as gluing operations, therefore it is called Double-PushOut approach (DPO). We start from the basic definition of graph and morphism. This approach are based on graphs and graph morphisms. Graphs are structures composed of nodes and edges, which allow the description of complex situations in a visual, compact, clear, and intuitive way. In this paper we use directed edges, therefore the source and target of each edge must be defined. Graph morphisms are used in order to relate graphs, they map all elements of one graph into the corresponding elements of another graph. This mapping must preserve the source and target of each edge, i.e., if an edge e_1 is mapped to an edge e_2, the source and target nodes of e_1 must be accordingly mapped to the source and target of e_2.

Definition 1 (graph, graph morphism)**.** *A* ***graph*** $G = (V, E, s, t)$ *consists of a set V of nodes, a set E of edges, and two functions, $s, t : E \to V$, the source and target functions. Given two graphs, $G_1 = (V_1, E_1, s_1, t_1)$ and $G_2 = (V_2, E_2, s_2, t_2)$, a* ***graph morphism*** $f : G_1 \to G_2$ *is composed by two total functions $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ such that $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$. A graph morphism is injective/surjective/iso if both components are injective/surjective/iso. The category of graphs and graph morphisms is called* **Graph**.

Figure 3.1 shows three graphs: G_1, G_2 and TG. As example the graph G_1 for-

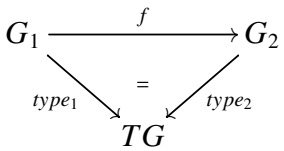
mally is: $(\{1, 2, 3{:}\text{pacman}, 5{:}\text{ghost}\}, \{a, b\}, \{(a, 3{:}\text{pacman}), (b, 1)\}, \{(a, 1), (b, 2)\})$. Figure 3.1 shows three graph morphisms: $type_1$, $type_2$ and f. The graph morphism $type_1$ is $(\{(1, x), (2, x), (3{:}\text{pacman}, \text{pacman}), (5{:}\text{ghost}, \text{ghost})\}, \{(a, a), (b, b)\})$.

Figure 3.1: Example of Typed Graph Morphism



For practical applications, it is very convenient to distinguish different types of vertices and edges in a graph. In the DPO approach, this can be achieved by the notion of typed graph. A typed graph is defined by two graphs together with a typing morphism, where the target graph is also called type graph. A typed graph morphism can only be defined over graphs typed over the same type graph. A compatibility condition ensures that the mappings of nodes and edges preserve types.

Definition 2 (typed graph, typed graph morphism)**.** *A* ***typed graph*** *is a triple* $(G1, type_{G1}, TG)$*, denoted by* $G1^{TG}$*, where* $G1$ *and* TG *are graphs and* $type : G1 \rightarrow TG$ *is a graph morphism. Given two typed graphs over the same type graph,* G_1^{TG} *and* G_2^{TG}*, and their respective typing morphisms* $type_{G1} : G_1 \rightarrow TG$ *and* $type_{G2} : G_2 \rightarrow TG$*, a* ***typed graph morphism*** *is a pair* (f, id_{TG})*, where* $f : G_1 \rightarrow G_2$ *is a graph morphism and* id_{TG} *is the identity morphism of* TG*, such that:* $type_{G2} \circ f = type_{G1}$*. A typed graph morphism is injective/surjective/iso if* f *is injective/surjective/iso.*



The category of graphs typed over TG *as objects and typed graph morphisms as morphisms is called* $\mathbf{Graph}_{TG}$*. This category is the comma category* (**Graph** $\downarrow$ **TG**).

Figure 3.1 presents an example of a typed graph morphism. G_1 and G_2 are graphs typed over TG. The type graph defines the syntax of graph used in transformations. In the example, *pacman* vertices may be connected to *berry* nodes, but *ghost* nodes may not (because in the type graph TG there is no connection between these latter nodes). In

the graphical notation, we will use the convention that nodes and edges with the same names in the source and target of a morphism depict the same item (are mapped by this morphism). To indicate the type we use the morphisms $type_1$ and $type_2$ that maps nodes *1* and *2* to the *x* node, which is a *block* (it is a place where other game elements can be), the remaining elements are mapped to the corresponding TG elements accordling to their own names.The morphism f in this case is an injective mapping, but it is not surjective since G_2 has elements that are no in the image of the morphism (for example, the edge *c*).

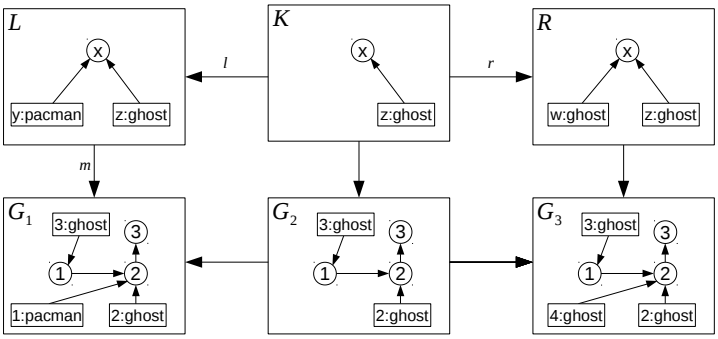
According to the DPO approach, a *rule* is a span of morphisms $p = L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$, where L, K and R are called the left-hand side, interface and right-hand side of the rule, respectively. Intuitively, a rule describes that, whenever an image of its left-hand side is found in a graph representing the state, a copy of the rule's right-hand side may replace this image. Thus, the left-hand side (or LHS) of a rule defines the items that must be present in the state to enable this rule's application, the interface defines the elements that must be present and will be preserved, and the right-hand side (RHS) defines which elements will be created by the application of the rule. The items that will be deleted are described indirectly: they are the elements that belong to the LHS and not to the interface graph.

Definition 3 (rule)**.** *A (typed graph) rule p consists of two typed graph morphisms l and r with the same typed graph as source,* $p = L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$*, where l and r are monomorphisms (in the category* $\mathbf{Graph}_{TG}$*, monomorphisms are injective mappings).*

The upper part of Figure 3.2 presents an example of rule, where mappings of elements are induced by using the same names and positions in the involved graphs. The pre-conditions to apply the rule in Figure 3.2 are: there must be a *ghost* and a *pacman* connected to the same *block* (called *x*), the elements in L that are not in K are deleted by the application, and the elements of R that are not in K are created, thus this rule deletes the matched *pacman* and creates a *ghost* in its place.

Rules can be applied to typed graphs, a candidate for an application is called *match*. A *match* as a typed graph morphism from the LHS of a rule to an arbitrary typed graph. However, in the DPO approach, the simple existence of a match is not sufficient to perform a graph rewriting. There are two situations that should be handled with care when applying a graph rule to a graph G: (i) when a node is specified for deletion and there are edges connected to it in G that are not in the image of the rule's LHS (*dangling condition*); and (ii) in the case that two distinct items of the rule, one to be deleted and the other preserved, are mapped to the same element of G, or two different deleted items

Figure 3.2: Example of Transformation



are mapped to the same in *G identification condition*). To ensure that no such situations occur when a rule is applied, the embedding of the LHS of the rule in the state graph must satisfy a condition called *gluing condition*, that encompasses both the dangling and the identification conditions. If a match satisfies this condition, it is possible to apply the rule, and the result is obtained by a double pushout construction.

Definition 4 (match, typed graph transformation)**.** *Consider a rule* $p = L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ *and a typed graph G, as in the diagram below. A* ***match*** *is an arbitrary typed graph morphism from L to G.*

A match m satisfies the ***gluing condition*** *iff both conditions below are satisfied:*

(dangling condition) *All edges of G that are connected to nodes that are in the image of m are also in the image of m;*

(identification condition) *If an element e of L is deleted (not in the image of l), no other element of L may be mapped to m(e) in G.*

Given a rule p and a match m, a ***(typed) graph transformation*** $G \overset{p,m}{\Longrightarrow} H$ *from G to H is defined as the diagram below, where (1) and (2) are* pushouts *in the category* $\mathbf{Graph}_{TG}$.

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow{\scriptstyle m'} \\ G & \xleftarrow[l']{} & D & \xrightarrow[r']{} & H \end{array}$$

Figure 3.2 presents a match m, where the nodes' mapping is { (x, 2), (y:pacman, 1:pacman), (z:ghost, 2:ghost) } and the edges' mapping is trivial.

3.2 Rules with NACs

When describing behaviour using graph rules, it is usually very convenient to be able to define a forbidden context that prevents rule application. This is called a Negative Application Condition, or NAC (HABEL; HECKEL; TAENTZER, 1996). In DPO, NACs are defined by embedding the left-hand side of the rule in the forbidden context. Then, if there is an image of the graph consisting of the LHS plus the forbidden context in the state, the rule is not applicable. A NAC may also forbid rule applications in which two different elements of the LHS of a rule are mapped to the same in the state graph. We will call the pair (LHS, NAC) of a rule as rule with NACs, or simply rule.

Definition 5 (NAC, rule with NACs, NAC satisfiability)**.** *Given a rule* $p = L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$*, a* ***negative application condition*** *NAC*(n) *for* p *is an arbitrary typed graph morphism* $n : L \rightarrow N$*. A NAC* $n : L \rightarrow N$ *is satisfied with respect to a match* $m : L \rightarrow G$ *if and only if* $\nexists q : N \rightarrow G$ *such that* q *is injective and* $q \circ n = m$*.*

A rule with NACs (p, NAC_p) *is composed by a rule* p *and a set of NACs for* p *(*NAC_p*).*

A match $m : L \rightarrow G$ *satisfies* NAC_p *if and only if it satisfies all single NACs in* NAC_p*, we denote as* $NAC_p \vDash m\,(p, G)$*.*

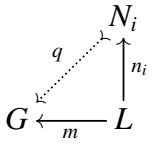
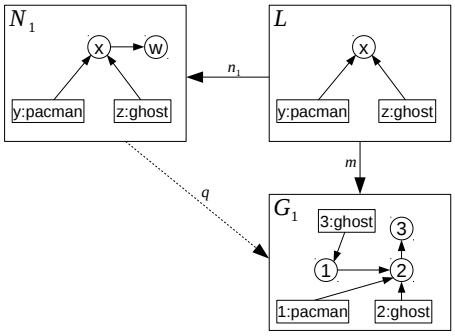


Figure 3.3 presents an example where a NAC forbids an application of a rule. The NAC n_1 forbids the application of the rule when the *block* that contains the pacman and the ghost is linked to another *block*. There is an injective morphism q that commutes with n_1 and m (mapping x to 2 and w to 3), and thus this match does not satisfy this NAC.

Figure 3.3: Example of Negative Application Condition



Definition 6 ((first order) graph transformation system)**.** *A **(first order) Graph Transformation System** consists of a set of (typed graph) rules with NACs*[1] *and a typed graph, called initial or start graph. The rules of a first-order graph transformation system are called first-order rules.*

We use the term *first-order* here because the next section introduces second-order graph transformations.

3.3 Second-Order Graph Transformations

In this section we review the notion of rule-based modification of typed graph rules without NACs presented in (MACHADO, 2012; MACHADO; RIBEIRO; HECKEL, 2015). This rewriting of rules is based on the DPO approach, thus the rule format remains : $L \leftarrow K \rightarrow R$. However, instead of rewriting graphs, rules will be used to rewrite other rules. This new rule scheme is called second-order rule, or 2-rule for simplicity, and it requires the definition of morphisms between rules.

Definition 7 (span, span morphism)**.** *A (typed graph)* span *is a diagram with shape $G \overset{l}{\leftarrow} G' \overset{r}{\rightarrow} G''$ in the category of T-typed graphs. For convenience, we refer to spans as the pair of morphisms (l, r) with common source. A **span morphism** $f : s \rightarrow s'$ between spans $s = (l, r)$ and $s' = (l', r')$ is a triple (f_L, f_K, f_R) of typed graph morphisms between the objects of the spans such that the diagram below commutes.*

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle f_L}\downarrow & = & \downarrow{\scriptstyle f_K} & = & \downarrow{\scriptstyle f_R} \\ L' & \xleftarrow[l']{} & K' & \xrightarrow[r']{} & R' \end{array}$$

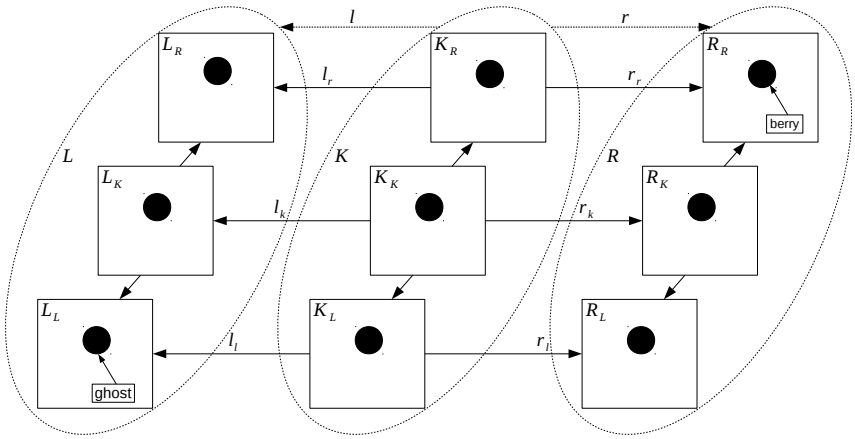
*A rule morphism is mono/epi/isomorphic if all three morphisms are also mono/epi/isomorphic. Spans and span morphisms constitute a category, named **Span**.*

Definition 8 (rule, rule morphism, span of rule morphisms)**.** *A (typed graph)* rule *r is a span $L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ such that l and r are* monomorphisms*, i.e. injective typed graph morphisms. A* rule morphism *$f : r \rightarrow r'$ is a span morphism (f_L, f_K, f_R) between rules. Notice that f_L, f_K and f_R are not always injective. A* span of rule morphisms *is a span $A \overset{a}{\leftarrow} B \overset{c}{\rightarrow} C$ such that a and c are* rule morphisms*.*

[1] In (MACHADO, 2012) this rules did not have NACs

Figure 3.4 shows two rule morphisms, l and r, between the dashed rules. These rule morphisms represent relations between rules, in this case, both are embeddings of rule K into rules L and R.

Figure 3.4: Example of Second-Order Rule



Definition 9 (second-order rule (2-rule))**.** *A second-order rule is a span of monomorphic rule morphisms. A second-order rule with (second-order) NACs is a pair* (s, NAC_s) *composed by a second-order rule* s *and a set of (second-order) NACs for* s*, where a second-order NAC is defined as a rule morphism with source on the left hand side of* s*.*

The LHS of the 2-rule presented in Figure 3.4 can be applied to any first-order rule which deletes a *ghost*. This 2-rule models the deletion of the *ghost* and the creation of a *berry*, that is, it will transform a rule that deletes a *ghost* into a rule that creates a *berry*, leaving the rest of the rule unchanged.

The transformation of a rule by means of a 2-rule is defined by means of a DPO diagram in the category **Span**, as in the case of graphs. One caveat exists, however: the resulting span may not be a valid rule because injectivity is not necessarily preserved by the rewriting. To mitigate this, it is possible to build (for each individual 2-rule) a set of *structure preserving* NACs that forbids any match that would result in a ill-formed rule. In the following, we omit this set of structure-preserving NACs because they only affect the selection of 2-rule matches, not the second-order rewriting itself. The construction of this set of NACs is detailed in (MACHADO, 2012; MACHADO; RIBEIRO; HECKEL, 2015).

Definition 10 (second-order transformation)**.** *Consider a second-order rule* $\alpha = (L_{\{L,K,R\}} \leftarrow K_{\{L,K,R\}} \rightarrow R_{\{L,K,R\}})$*, and a first-order rule* $p = (L \leftarrow K \rightarrow R)$*, as in the diagram below. The upper part of this diagram is a 2-rule as the corresponding names*

with Figure 3.4 indicating. A second-order match is a rule morphism from $L_{\{L,K,R\}}$ to p. Let $l = (L_L \leftarrow K_L \rightarrow R_L)$, $k = (L_K \leftarrow K_K \rightarrow R_K)$ and $r = (L_R \leftarrow K_R \rightarrow R_R)$. A second-order transformation $p \overset{\alpha,m_{\{1,2,3\}}}{\Longrightarrow} p''$ $(L'' \leftarrow K'' \rightarrow R'')$ is defined by the diagram below, where $L \overset{l,m_1}{\Longrightarrow} L''$, $K \overset{k,m_2}{\Longrightarrow} K''$ and $R \overset{r,m_3}{\Longrightarrow} R''$ are typed graph transformations and $(L' \leftarrow K' \rightarrow R')$ and $(L'' \leftarrow K'' \rightarrow R'')$ are valid typed graph rules.

Given a second-order transformation, the span $p \leftarrow p' \rightarrow p''$ is called rule evolution.

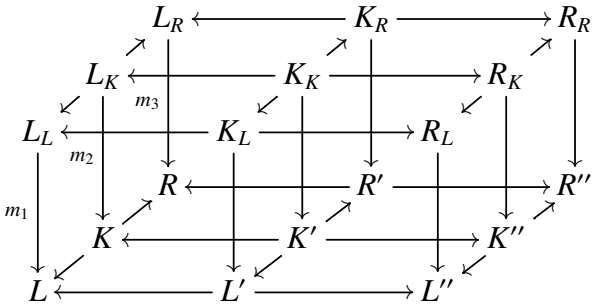
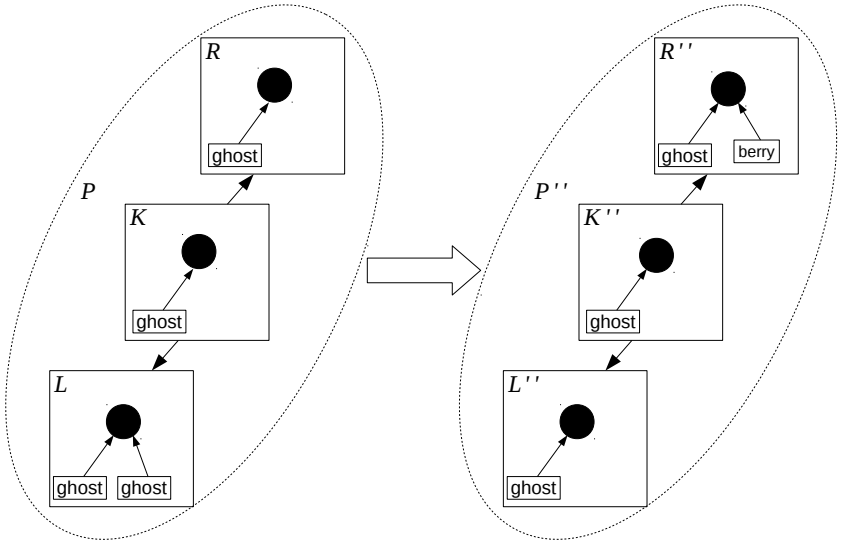


Figure 3.5 shows an example of rule evolution from a rule P to a rule P'' using the 2-rule depicted in Figure 3.4.

Figure 3.5: Example of Second-Order Transformation



4 SECOND-ORDER TRANSFORMATIONS AND NACS

NACs are widely used in practice, being very important in the modeling of real systems. The fact that second-order rewriting as defined in (MACHADO, 2012) does not provide support for modifying rules containing NACs, is an important lack for this definitions and practical usage.

Including NACs on the rule evolution can be made in many ways. We envision two scenarios in which NACs interact with second-order rewritings:

- **Evolution of first-order NACs.** In this scenario, given a rule with NACs (p, NAC_p) and a second-order transformation $p \Rightarrow p''$ transforming $p = L \leftarrow K \rightarrow R$ into $p'' = L'' \leftarrow K'' \rightarrow R''$, the question is how to obtain a set $NAC_{p''}$ to build a rule with NACs $(p'', NAC_{p''})$ maintaining, as much as possible, the same semantics of the set NAC_p with respect to p.

- **Programmed transformation of first-order NACs.** In this scenario, particular manipulation of the NACs themselves are integrated into the second-order transformation framework, allowing the second-order rule to operate on both the span $p = L \leftarrow K \rightarrow R$ and the set NAC_p.

It is important to mention that NACs for 2-rules (second-order rules that manipulate the first-order ones) have already been considered in (MACHADO, 2012). Also all rule evolution definitions used are inherited and kept from that work. However, this work purposes a study of new definitions, where first-order rules are allowed to have NACs in their evolution context, in according with already defined second-order transformation.

The next section elaborates on the first of these scenarios, i.e. evolution of first-order NACs. A version of these ideas also appear in (COSTA; MACHADO; RIBEIRO, 2016), however in a very preliminar stage.

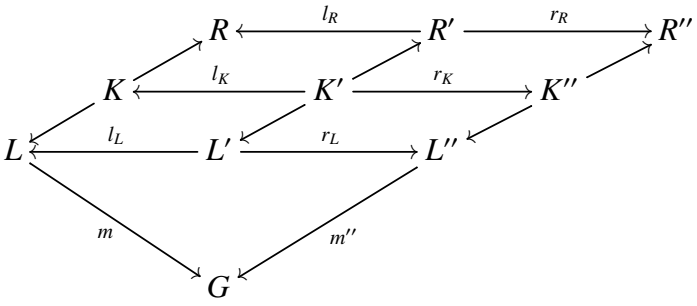
4.1 Evolution of first-order NACs

Given a second-order transformation $p \Rightarrow p''$ transforming rule p into p'' and a set of NACs for p, we wish to obtain a set of NACs for p'' keeping (as much as possible) the same semantics as the original set of NACs for p. In other words, the ideal scenario is that the original and the transformed NACs should forbid exactly the same kind of structures. However, as the preconditions (the left-hand side) of the rules can be modified, also NACs

must undergo alterations in a compatible way. In this section we formalize this notion of semantic preservation, and characterize the situations where it holds. For a fixed graph G, we start by defining when a match of p over G is related to a match of p'' over G.

Definition 11 (Related matches)**.** *Given a rule evolution $p \overset{l}{\leftarrow} p' \overset{r}{\rightarrow} p''$ (where $p = L \leftarrow K \rightarrow R$, $p' = L' \leftarrow K' \rightarrow R'$ and $p'' = L'' \leftarrow K'' \rightarrow R''$), a typed graph morphism $m\colon L \rightarrow G$ (representing a match for p in G) and a typed graph morphism $m''\colon L'' \rightarrow G$ (representing a match for p'' in G). We say that m and m'' are related matches (by means of the rule evolution) if and only if*

- *m satisfies DPO gluing conditions for p*
- *m'' satisfies DPO gluing conditions for p''*
- *the equation $m \circ l_L = m'' \circ r_L$ holds, i.e. the following diagram commutes*



Intuitively, given a match $m : L \rightarrow G$, a related match $m'' : L'' \rightarrow G$ has the same mapping as m in the part of the rule precondition (left-hand side) preserved by evolution. In this way, m and m'' are related in the sense of matching equally the same elements after and before the evolution.

Definition 12 (Preservation of NAC-behavior)**.** *Let $p \leftarrow p' \rightarrow p''$ be a rule evolution, NAC_p be a set of NACs for p and $NAC_{p''}$ be a set of NACs for p''.*

We say that

- *$NAC_{p''}$ preserves the NAC-blocking behavior of NAC_p when, for every $m : L \rightarrow G$ and related $m'' : L'' \rightarrow G$, we have*

$$NAC_p \not\models m \Rightarrow NAC_{p''} \not\models m''$$

- *$NAC_{p''}$ preserves the NAC-allowing behavior of NAC_p when, for every $m : L \rightarrow G$*

and related $m'' : L'' \to G$, we have

$$NAC_p \vDash m \Rightarrow NAC_{p''} \vDash m''$$

- *$NAC_{p''}$ preserves the NAC-behavior of NAC_p whenever $NAC_{p''}$ preserves the NAC-blocking behavior and NAC-allowing behavior of NAC_p.*

Before continuing with our analysis, it is helpful to review a concrete example of the behavior of NACs in DPO under arbitrary matches. In essence, NACs can be designed to enable or disable a match $m : L \to G$ based on characteristics encoded in the NAC itself:

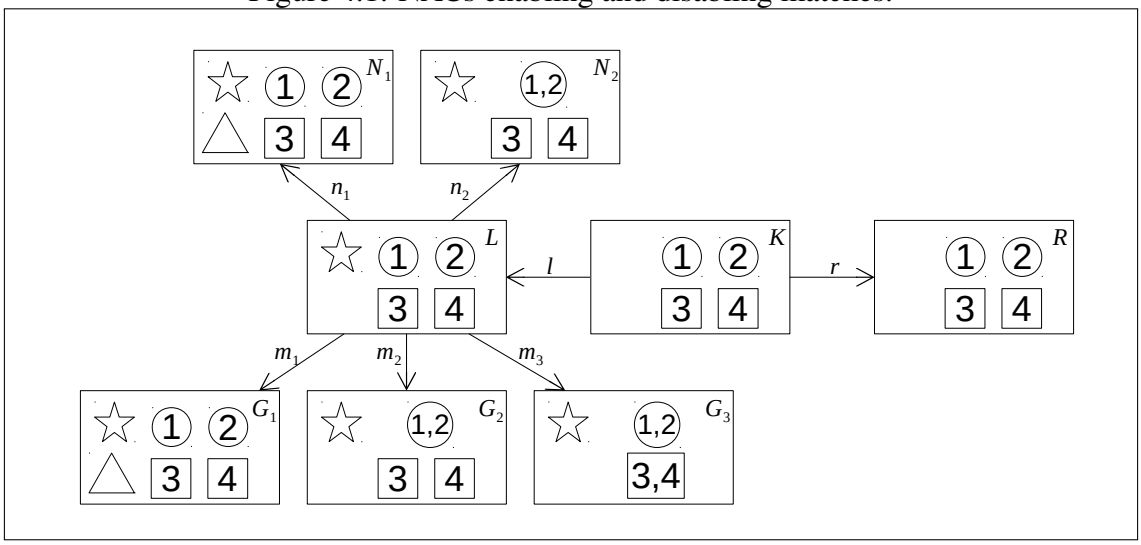
- presence of additional elements in G, which do not appear in L;
- identification of preserved elements in L (when they appear non-identified in the NAC);
- non-identification of preserved elements in L (when they appear identified in the NAC);

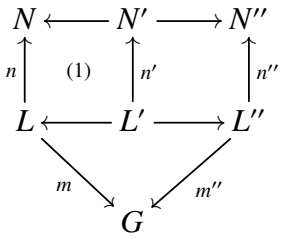
These effects can be observed in the Figure 4.1. The match m_1 is disabled because of NAC n_1, due to the presence of a star in G_1. The match m_2 is disabled because of NAC n_2 due to the identification of the circles. The match m_3 is not disable by n_2 because n_2 does not identify the squares, even considering that m_3 identify the circles as specified by n_2. This is a consequence of the definition of NACs requiring a factoring monomorphism to disable a given match.

We now show how, given a rule evolution $p \leftarrow p' \rightarrow p''$ and a set NAC_p of negative applications for p, it is possible to build a set $NAC_{p''}$ of NACs for p''. We will show that $NAC_{p''}$ preserves the NAC-blocking behavior of NAC_p. We also present a counter-example that shows that a NAC-allowing behavior preservation is not possible in general, considering arbitrary evolutions.

As first approach, let us consider the consequences of attempting to use DPO rewriting to update the NACs. Taking as first-order rule the span $L \leftarrow L' \rightarrow L''$ and $n : L \to N$ as match, by employing DPO transformation we could obtain as result the diagram below, and consider an evolved NAC the morphism $n'' : L'' \to N''$.

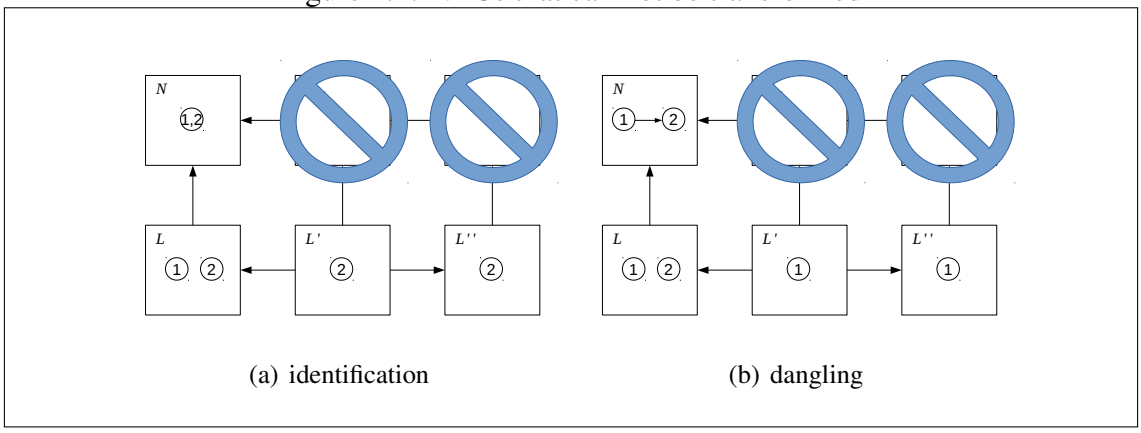
Figure 4.1: NACs enabling and disabling matches.





This definition give us a suitable transformation for NACs, since it is defined over the widely used DPO operations for typed graph transformations. It is also is well known that DPO transformations not always exist in this situation we define that the resulting NACs is *true*, which means it is always satisfied.

Figure 4.2: NACs that can not be transformed

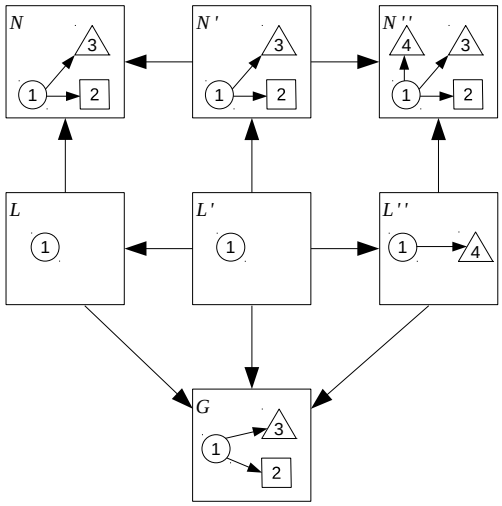


In fact, a transformation can not occur only in two cases exactly when the identification and dangling conditions are violated. In the following we show that, when this happens, the forbidden structure of the NAC before the evolution does not exist in the evolved rule, and therefore the NAC is useless and may be removed.

In the case of the identification condition, Figure 4.2(a), a NAC forbids the application when the circles 1 and 2 are mapped to the same circle $(1, 2)$. Since the evolution deletes the circle 1, this NAC does not make sense after evolution. In the case of dangling condition, Figure 4.2(b), a NAC forbids the application when the circles 1 and 2 are connected by an edge $1 \rightarrow 2$. However the evolution deletes the circle 2, therefore after the evolution there is not no need for this NAC.

However, DPO transformations fail to forbid some expected situations in this process. Suppose an evolution where the left-side of the evolved rule had a node and it changes to have another linked node to the old one. At same time, this rule has a NAC that forbidden the connection between these two nodes, as in Figure 4.3. In this specific evolution context, there must be a set of resulting NACs to forbid the same situations that the original NAC forbids. The DPO transformation is well-known that returns only one morphism.

Figure 4.3: Situation that DPO fails to generate all NACs



Since DPO does not allow to translate a set of NACs over a morphism preserving the NAC-blocking behavior, another approach is required. A solution was found in (LAMBERS, 2010) where the authors show how to construct, from a graph A with NACs, a set of equivalent NACs on a graph B via a morphism $t : A \rightarrow B$. A shifting of NACs example was shown in the evolution with NACs example, in Figure 2.17, the evolved NACs were generated by the shift operation.

The shift of a NAC along a morphism is a set of NAC-shifts, where a NAC-shift diagram can be build in some constraints.

Definition 13 (NAC-shift (over a morphism))**.** *Given a NAC $n : A \rightarrow N$, a morphism*

$A \to B$ and the diagram below, (1) is a NAC-shift if:

(i) (1) commutes.

(ii) t' and n' are jointly epi.

(iii) t' is mono.

Here $D_t(n)$ is the shift from one NAC.

Definition 14 (Shift of a NAC along a morphism)**.** *Given a NAC $n : A \to N$ and a monomorphism $t : A \to B$:*

$$\begin{array}{ccc} N & \xrightarrow{t'} & N' \\ {\scriptstyle n}\uparrow & (1) & \uparrow{\scriptstyle n'} \\ A & \xrightarrow[t]{} & B \end{array}$$

$D_t(n) = \{n' \mid n' : B \to N', t' : N \to N'\}$ where (1) is a NAC-shift.

Then it is possible to shift a set of NACs.

Definition 15 (Shift of a set of NACs)**.** $D_t(NAC) = \bigcup_{n \in NAC} D_t(n)$

Using shift of NACs it is possible to define the evolution of them. This process is a transformation in two steps, as in DPO, the first deletes NAC elements or also the entire NAC, and second the creation of the resulting NACs.

In this way, we can describe the NACs evolution process as a transformation where: first a pushout complement as in DPO transformations, and after the shift of NACs over a morphism as defined above. Note that in this process each NAC can be evolved to zero or many NACs, which is not a problem since our aim is only to preserve the forbidden structures, and not the NACs themselves.

Definition 16 (Evolution of a set of NACs)**.** *Let $p \leftarrow p' \to p''$ be a rule evolution, where $p = L \leftarrow K \to R$, $p' = L' \leftarrow K' \to R'$ and $p'' = L'' \leftarrow K'' \to R''$. Let NAC_p be a set of NACs for p. We define the* evolved set of NACs *$NAC_{p''}$ as*

$$NAC_{p''} = D_t(NAC_p)$$

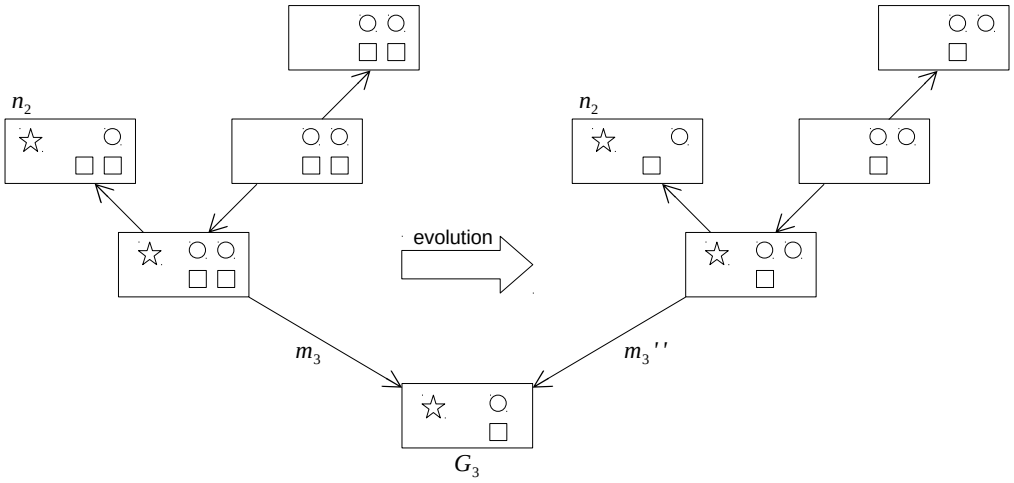
Theorem 1 (Preservation of NAC-blocking behavior)**.** *Let $p \leftarrow p' \to p''$ be a rule evolution, where $p = L \leftarrow K \to R$, $p' = L' \leftarrow K' \to R'$ and $p'' = L'' \leftarrow K'' \to R''$. Let $m : L \to G$ and $m'' : L'' \to G$ be related matches. Let NAC_p be a set of NACs for p, and $NAC_{p''} = D_t(NAC_p)$.*

$$\textit{If } NAC_p \not\models m(p, G) \textit{ then } NAC_{p''} \not\models m''(p'', G).$$

Proof. The proof of the equivalence of set of NACs in A and the set of NACs in B can be found in (LAMBERS, 2010). □

Given that the constructed set of NACs preserves NAC-blocking behavior, we now discuss the possibility of preservation of NAC-allowing behavior. Unfortunatelly, this is not possible in the general case, as the following counterexample exposes.

Figure 4.4: Evolution where NAC-allowing behavior is not preserved.



Example 1 (Invalidation of NAC-allowing behavior)**.** *Figure 4.4 depicts an example of evolution of a rule with NACs. The original rule deletes a star in the presence of two circles and two squares. Its only NAC $n_2 : L \to N_2$ forbids the application whenever the preserved circles are identified. Notice that $m_3 : L \to G_3$ is not disabled by n_2, since there is only one square, and therefore there is no injective factorization $e : N_2 \rightarrowtail G_3$. The evolution consists of removing one of the preserved squares from the rule, generating a rule that deletes a star in the presence of two circles and one square. The associated NAC evolution calculates a single evolved NAC $n_2'' : L'' \to N_2''$, without the deleted square. Notice, however, that the only thing that prevented the NAC n_2 from disabling m_3 was the impossibility of identifying the squares in a mono factorization. Given that there is now only one square, there is actually a possible factorization $e'' : N_2'' \rightarrowtail G_3$ for n_2'', and therefore n_2'' disables the match m_3'', which is related to m_3 by the evolution.*

The situation occurs because some NACs are not triggered due to the identification of some elements by the match. When an evolution deletes some of these preserved

elements, the distinction between matches that would be allowed or disabled by the NAC disappear. In this case, the NAC disables the resulting match. This is actually an effect of the pushout complement part of the NAC evolution definition.

If, however, we restrict ourselves to injective matches, these distinctions which are dependent on identifications are not allowed, and it is actually possible to have preservation of NAC-allowing behavior.

Theorem 2 (Preservation of NAC-allowing behavior for injective matches)**.** *Let* $p \leftarrow p' \rightarrow p''$ *be a rule evolution, where* $p = L \leftarrow K \rightarrow R$, $p' = L' \leftarrow K' \rightarrow R'$ *and* $p'' = L'' \leftarrow K'' \rightarrow R''$. *Let* $m : L \rightarrowtail G$ *and* $m'' : L'' \rightarrowtail G$ *be related injective matches. Let* NAC_p *be a set of NACs for* p, *and* $NAC_{p''} = D_t(NAC_p)$.

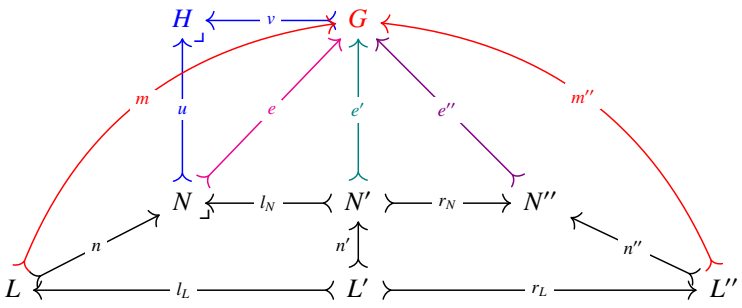
$$\textit{If } NAC_p \vDash m\,(p, G) \textit{ then } NAC_{p''} \vDash m''\,(p'', G).$$

Proof.

- (by definition)
  If for all $n : L \rightarrow N \in NAC_p$ there is no monomorphism $e : N \rightarrow G$ such that $e \circ n = m$, then for all $n'' : L'' \rightarrow N'' \in NAC_{p''}$ there is no monomorphism $e'' : N'' \rightarrow G$ such that $e'' \circ n'' = m''$.

- (contrapositive)
  If exists $n'' : L'' \rightarrow N'' \in NAC_{p''}$ and monomorphism $e'' : N'' \rightarrow G$ such that $e'' \circ n'' = m''$, then exists $n : L \rightarrow N \in NAC_p$ and monomorphism $e : N \rightarrow G$ such that $e \circ n = m$.

- (existence of monomorphism e) consider the diagram below:



  – assume monomorphisms $m : L \rightarrowtail G$, $m'' : L'' \rightarrowtail G$ and $e'' : N'' \rightarrowtail G$ in the diagram above. Notice that each $n'' \in NAC_{p''}$ was created from some

$n \in NAC_p$ by means of a pushout complement and a NAC shift commutative square, as shown in the diagram;

- $n'' : L'' \to N''$ is mono because $m'' = e'' \circ n''$ and m'' is mono. By a similar argument, note that n' and n are also mono.
- let $e' : N' \to G$ be the composition of monos $e'' \circ r_N$;
- let (u, v) be the pushout of (e', l_N). Because the category of graphs is adhesive, pushouts preserve monomorphisms and, therefore, both $u : N \to H$ and $v : G \to H$ are mono;
- let $e : N \to G$ be the unique arrow from pushout square (n', l_L, L_N, n) towards the cospan (m, e');
- $e : N \to G$ is mono because $u = v \circ e$, and u is mono.

□

As Theorems 1 and 2 show, NACs preservation is constrained by the kind of morphism allowed as a rule match:

- with general matches, only preservation of blocking behavior is possible.
- with injective matches, preservation of allowing and blocking behavior is possible.

Considering this scenario, we use Definition 16 as appropriated algorithm for evolving a set of NACs. This is the basis of our implementation of a second-order transformation model for first-order rules with NACs.

5 IMPLEMENTATION

In the graph transformation area, there are many tools that support simulation and analysis of various kinds of graph grammars models. We highlight below three tools that are important to our work since they implement related notions of first-order graph transformations.

The Attributed Graph Grammar System (AGG) (TAENTZER, 2000) is a tool that supports typed graph grammars. The rewriting engine is based on single pushout (SPO) approach, that differs from DPO used in this work, however SPO with NACs can simulate DPO adding some configurations. This tool supports attributed graphs, that is the graph elements can have algebraic types, in this case Java types. AGG focuses on static analysis, critical pairs and sequences analysis very closed to the defined in this work is implemented, besides that concurrent rules, termination and consistency checking are also available.

The Graphs for Object-Oriented Verification (GROOVE) Tool Set (RENSINK, 2004) is another tool for modeling graph grammars, its rewriting operation also follows the SPO approach. GROOVE graphs are typed over a system of labeling, that simulates types. The focus of this tool is generation and exploration of state space, and it is very efficient in the search for isomorphic states.

A more recent tool is the Verigraph (COSTA et al., 2016). It implements a rewriting system based on the DPO approach. Static analysis as critical pairs/sequences and concurrent rules are implemented. At the moment, there is an initial version of a state space exploration with model checking through CTL expressions. Nonetheless the main contribution of Verigraph is its architecture, where high-level algorithms can be defined using categorical operations in the sense of adhesive HLR systems (EHRIG et al., 2004). This approach allowed the development of the first SOGG implementation of second-order rewriting.

The development of Verigraph from a graph manipulation tool to a transformation system was made along with this work. Many features were made to support this work, such as: (i) implementation of second-order rewriting and related analysis techniques, as in (MACHADO, 2012); (ii) implementation of the framework to handle modifications in rules with NACs. The next section follows from (COSTA et al., 2016), it details the structure of this tool, after we present how the second-order transformation was implemented.

5.1 Verigraph

The Verigraph[1] tool comes from the need of the Verites[2] research group of a tool that supports a quick prototyping of graph transformation theoretical concepts. Developed in Haskell, an initial version of this tool was published in (BECKER, 2014). From this initial study, this tool developed in various directions. In (BEZERRA; RIBEIRO, 2016) the calculation of concurrent rules in Verigraph is presented, at the same symposium (COSTA; MACHADO; RIBEIRO, 2016) presented the initial ideas for this dissertation that was already implemented in this tool. The main contributions of Verigraph are published in (COSTA et al., 2016), where the internal architecture is detailed and the main results are summarized.

The next sections review these structures, we present the needed code in order to present, at the final of this chapter, the implementation of the NAC manipulation as proposed in this dissertation.

5.1.1 Architecture Overview and Data Structures

Verigraph's architecture is based on abstract classes that map important theoretical notions, such as morphism, DPO transformation, Adhesive HLR system among others. High level operations are described over these abstract notions, for example, a DPO transformation is automatically available for a class that implements pushout, pushout complement, search of morphisms and others. The concrete classes of the tool are based on integer sets, which are enriched in order to simulate graphs, and then graph morphisms and others in a incremental way, as text below describes.

The first structure in Verigraph is the `Graph` type, in Figure 5.1, which consists of a list for nodes and another for edges. Nodes and edges have numeric identifiers that are unique within a graph. They may also carry a *payload* information, which is not used by now but allows to adding additional information, which should be used in the future to implement attributed graphs. In the figure, basic functions for manipulating graphs were omitted, such as `insertNode`, `insertEdge`, `removeNode`, `removeEdge`, `incidentEdges`, `neighbourNodes`, among others.

The next natural structure for graph transformation are graph morphisms. As de-

[1] <https://github.com/verites/verigraph>
[2] <http://www.ufrgs.br/verites>

Figure 5.1: Implementation of graphs

```
data Node a = Node
 { nodePayload :: Maybe a
 } deriving (Show, Read)

data Edge a = Edge
 { source      :: NodeId
 , target      :: NodeId
 , edgePayload :: Maybe a
 } deriving (Show, Read)

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

fined in the formal definition, it consists of domain and codomain graphs, as well as two relations, mapping nodes and edges identifiers. Figure 5.2 shows the corresponding code. We implement a polymorphic `Relation` type, it has functions such as `compose`, `inverse`, `domain`, `image` and `apply`. The usage of graph morphisms demands only the functional relations for this type.

Figure 5.2: Implementation of graph morphisms

```
data GraphMorphism a b =
 GraphMorphism {
    domain     :: Graph a b
  , codomain   :: Graph a b
  , nodeRelation :: Relation NodeId
  , edgeRelation :: Relation EdgeId
  } deriving (Read)
```

Typed graphs, in Figure 5.3, are implemented as graph morphisms whose codomain is a type graph. In this way `TypedGraph` is a synonym for `GraphMorphism`. Typed graph morphisms (TGMs) also follow the formal definition: they consist of a domain and codomain typed graphs, as well as a mapping (a graph morphism) between them.

We have presented the implementation of morphisms of two different categories, **Graphs** (graphs as objects and their morphisms as arrows) and $\mathbf{Graphs}_{TG}$ (introduced in Definition 2). Therefore, a whole class of operations relative to morphisms is relevant for both data types. In order to uniformly deal with both morphism types, the `Morphism` type class was defined. It may be seen in Figure 5.4, note that the type morphism depends of the type *m* to be a concrete type. Since every morphism type has an associated type for representing objects of that category, standard Haskell would not suffice to express these

Figure 5.3: Implementation of typed graphs and their morphisms

```
type TypedGraph a b = GraphMorphism a b

data TypedGraphMorphism a b =
  TypedGraphMorphism {
      domain   :: TypedGraph a b
    , codomain :: TypedGraph a b
    , mapping  :: GraphMorphism a b
  } deriving (Show, Read)
```

ideas We therefore employed a compiler (GHC) extension allowing type families, which are essentially functions at the type level (SCHRIJVERS et al., 2008).

Figure 5.4: Type class for morphisms

```
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
```

Besides the basic operations for morphisms of general categories, many of the algorithms implemented in Verigraph depend on operations available in Adhesive High-Level Replacement (HLR) categories (EHRIG et al., 2004), such as pushouts and pull-backs. Morphisms of such categories must therefore implement the operations of the `AdhesiveHLR` type class, which may be seen in Figure 5.5. These type classes decouple the implementation of DPO rewriting from the implementation of the category in which it occurs.

Having an abstract notion of morphism, the implementation of *productions* may be polymorphic on the morphism type, as may be seen in Figure 5.6. The operations related to productions, such as checking the applicability of a match (existence of gluing condition and satisfiability of NACs) and doing the actual transformation, are implemented in terms of the `AdhesiveHLR` class.

Also, two different notions for checking the NAC satisfiability were implemented: the classical one (as presented in Chapter 3) and the partial injective version as defined in (LAMBERS, 2010) and implemented in AGG.

The last type is `DPO`. This abstract class has two needed functions: `inverse` to

Figure 5.5: AdhesiveHLR Class

```
class (Morphism m) => AdhesiveHLR m where

  calculateInitialPushout :: m -> (m, m, m)

  calculatePushout :: m -> m -> (m, m)

  -- Tests if a pushout complement exists
  hasPushoutComplement :: (MorphismType, m) -> (MorphismType, m) -> Bool

  -- Assumes a pushout complement exists
  calculatePushoutComplement :: m -> m -> (m, m)

  calculatePullback :: m -> m -> (m, m)
```

Figure 5.6: Data Production

```
data Production m =
 Production
  { left :: m
  , right :: m
  , nacs :: [m]
  }
```

invert a production, and `shiftNac` to shift a set of NACs over a production. The implementation of these presented functions for a type allows the usage of abstract functions such as `findAllMatches`, `findApplicableMatches`, `nacDownwardShift`, `satisfiesRewritingConditions`.

Note that the `Graph`, `GraphMorphism`, `TypedGraph` and `Production` types allow malformed instances (e.g. a `Graph` with an edge that references an undefined node as source or target). Therefore, a *well-formedness* condition must be implemented, which was done by defining the type class `Valid`, instantiated for those types, providing a single predicate `valid` for its verification. In order to reduce the runtime overhead of such checks, the algorithms implemented in Verigraph are designed to construct and preserve well-formedness. Thus, it must only be checked when values are obtained from external sources.

This implementation is sufficient to perform first-order graph transformations. Besides that, a generic structure to support other kinds of transformations is defined. This allows us to easily adapt Verigraph for transformation on other similar Adhesive HLR categories.

5.1.2 Analysis Techniques

In this section we aim to explain how are implemented the analysis techniques in the Verigraph tool. A set of analysis is currently implemented, such as critical pairs and sequences, concurrent rules, inter-level analysis and graph processes. For this work, inter-level analysis is particularly relevant.

In order to illustrate how these methods were implemented, we present the code that detects delete-use and produce-dangling conflicts.

This algorithm in Figure 5.7 checks if a pair of transformations are in one of these conflict types. A delete-use occurs when a transformation deletes something used by other, in this sense there is not exists a morphism from the left side of a production to the interface of the transformed object. The produce-dangling occurs when a transformation produces some structure that unable (via dangling condition) other transformation, in the diagram it is reflected when there is a match for the left side of a production on the transformed object of the other production, but it is not a valid match.

Figure 5.7: Delete-Use and Produce-Dangling Verification Algorithms

```
deleteUseDangling :: DPO m =>
  Production m -> Production m -> (m, m) -> Maybe ConflictType
deleteUseDangling p1 p2 (m1,m2) =
  case (null h21, dangling) of
    (True,_)      -> Just (Left (m1,m2))   -- delete use case
    (False,True) -> Just (Right (m1,m2)) -- produce dangling case
    _             -> Nothing               -- free overlap case
  where
    (k1,d1) = calculatePushoutComplement m1 (getLHS p1)
    (_,e1) = calculatePushout k1 (getRHS p1)
    l2TOd1 = findMorphisms (domain m2) (domain d1)
    h21 = filter (\x -> m2 == compose x d1) l2TOd1
    h21_e1 = compose (head h21) e1 --h21 is unique if it exists
    dangling = not (satisfiesGluingConditions p2 h21_e1)
      && satisfiesNACs p2 h21_e1
```

This function receives two transformations (two productions p_1 and p_2, and theirs matches (m_1, m_2) on a same graph) and returns: *Just DeleteUse* if there is not the h_{21} morphism; *Just ProduceDangling* case there is h_{21} and *dang* is true; and returns *Nothing* otherwise. The proximity of code and theory can be verified with this example. Note that all operations are in a high-level of abstraction, such as *calculatePushoutComplement*, *findMorphisms*, *compose*, *calculatePushout*, among others.

As discussed above, this function works for any morphism type that implements these basic operations. We already presented that Typed Graph Morphism implements

these operations, but in the next section another kind of morphism that implements the `AdhesiveHLR` class is presented.

5.1.3 Second-Order Transformation

Second-Order Graph Grammars (SOGG) as proposed in (MACHADO, 2012) are a new kind of transformation that can model evolution of typed graph grammars. As presented in Chapter 3, the new rewriting system is based on *rule morphism* concept.

In Verigraph we implemented directly the `RuleMorphism` type as showed in Figure 5.8. This type is composed by two rules and three morphisms between their elements, note that the well-formedness is guaranteed only in the `Valid` implementation for this type. The `RuleMorphism` type also implements others classes such as `Morphism`, `FindMorphism`, `EpiPairs` and `Cocomplete`, more details in the Verigraph repository.

Figure 5.8: Implementation of Rule Morphism

```
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
  } deriving (Eq, Show, Read)
```

Due to the fact that SOGG form an adhesive HLR category with NACs, we also implemented the functions in `AdhesiveHLR` and `DPO` classes for the `RuleMorphism` type. In this way, in the theory and in the implementation the analysis proposed for adhesive HLR categories are also valid for this type, for example the algorithm showed in Figure 5.7.

5.1.3.1 Pushout Complement

The pushout complement operation is responsible for the deletion process in the DPO approach. For this reason it is also performs the NACs deletion in the second-order transformation. A NAC is deleted when it does not fulfills the gluing conditions in the transforming process, as discussed in Section 3.1.

The algorithm for the pushout complement for second-order transformations in the

Verigraph is presented in Figure 5.9. It receives two `RuleMorphism` instances *m* and *l*, and returns theirs respective morphisms *k* and *l'*. According to diagram in the figure, usually *l* is a left morphism of a 2-rule, *m* is the second-order match and *H* the rule after the deletion process.

Figure 5.9: Implementation of Pushout Complement with NACs

```
instance AdhesiveHLR (RuleMorphism a b) where

  -- Pushout Complement for second-order with deletion and transposing of NACs.
  -- It runs the pushout complement without NACs,
  -- filters the NACs in the matched rule (ruleG) selecting the non deleted,
  -- and updates the rule H with the transposed NACs.
  --
  --        l
  --     L<------K
  --     |       |
  --   m |       | k          Pushout Complement Diagram
  --     V       V
  --     G<------H
  --        l'
  calculatePushoutComplement
    m@(RuleMorphism _ ruleG matchL matchK matchR)
    l@(RuleMorphism ruleK ruleL leftL leftK leftR) = (k,l')
     where
       (matchL', leftL') = calculatePushoutComplement matchL leftL
       (matchK', leftK') = calculatePushoutComplement matchK leftK
       (matchR', leftR') = calculatePushoutComplement matchR leftR

       leftH = commutingMorphismSameCodomain
             (compose leftK' (getLHS ruleG)) leftL'
             matchK' (compose (getLHS ruleK) matchL')
       rightH = commutingMorphismSameCodomain
             (compose leftK' (getRHS ruleG)) leftR'
             matchK' (compose (getRHS ruleK) matchR')

       validNACs = filter (satisfiesNACRewriting leftL') (getNACs ruleG)

       newRuleNACs =
        map (\nac -> fst (calculatePushoutComplement nac leftL')) validNACs

       ruleH = buildProduction leftH rightH newRuleNACs
       k = RuleMorphism ruleK ruleH matchL' matchK' matchR'
       l' = RuleMorphism ruleH ruleG leftL' leftK' leftR'
```

This algorithm constructs the *ruleH* with typed graph morphisms $leftH$ and *rightH* obtained of the first-order pushout complement plus a commuting constraint. This part is complete for the transformation without NACs. Furthermore, in *ruleH* are added the *newRuleNACs*, these NACs can be obtained from a filtering of application conditions on *left'* of *ruleG* NACs.

5.1.3.2 Pushout

The operation used for the creation step is the pushout, consequently it is also used by the creation of NACs in the second-order transformation. This algorithm must create NACs when there is some NAC on the received rule, that is a NAC that is being transposed as defined in Section 4.1.

This pushout in Verigraph is implemented as in Figure 5.10. It receives two `RuleMorphism` instances g and f, usually a right morphism of a 2-rule and a resulting morphism of the pushout complement, respectively. It returns f' and g' that are morphisms with codomain in the resultant *ruleP*.

Figure 5.10: Implementation of Pushout with NACs

```
instance AdhesiveHLR (RuleMorphism a b) where
  ...
  -- Pushout for second-order with creation of NACs.
  -- It runs the pushout without NACs (from cocomplete),
  -- generates all NACs (considering arbitrary matches) for the rule H,
  -- and updates the morphisms f and g to get the new NACs.
  --
  --       g
  --    K------>R
  --    |       |
  --  f |       | f'        Pushout Diagram
  --    V       V
  --    D------>P
  --       g'
  calculatePushout
    f@(RuleMorphism _ ruleD _ _ _)
    g@(RuleMorphism _ ruleR _ _ _) = (f',g')
    where
      (RuleMorphism _ preRuleP f'L f'K f'R,RuleMorphism _ _ g'L g'K g'R) =
        Abstract.Cocomplete.calculatePushout f g

      ruleP = buildProduction (getLHS preRuleP) (getRHS preRuleP) transposedNACs

      f' = RuleMorphism ruleR ruleP f'L f'K f'R
      g' = RuleMorphism ruleD ruleP g'L g'K g'R

      transposedNACs = concatMap (nacDownwardShift g'L) (nacs ruleD)
```

Since the `RuleMorphism` type implements `Cocomplete`, we can define the pushout without NACs from this class. In this code the rule *preRuleP* is obtained in this operation. The NACs part is added by *transposedNACs*. The transposed are obtained directly by the first-order shift NACs over *ruleD* NACs.

Finally, according to Definition 16 second-order transformations can be represented in Verigraph tool based on above operations. It is important to note that it is the

first tool to implement these second-order transformations, moreover according tool architecture all DPO properties are inherited, conflicts and dependencies analysis for example.

The extension of the Verigraph tool towards second-order transformation with NACs is one of the major contributions of this work. The development was mainly guided by the need for an implementation of second-order rewriting: at the start, Verigraph was in a simple state, with only first-order rewriting implemented. Most analysis techniques for first-order layer and support for NACs in rules were implemented within the scope of this work. Verigraph was initially used for proof of concepts, to assist the theoretical study, but quickly the tool started to become useful for analysis of graph transformation in general. As an example, it was used in (CORRADINI et al., 2018) to perform experimental evaluation of distinct ways of calculating parallel independence for the double-pushout and sesqui-pushout approaches.

6 RELATED WORK

In this chapter, we compare our work with other approaches found in literature that deal with evolution on rule-based systems considering negative application conditions. The works listed in this section are not restricted to the classical graph grammars scope, however we list the closest results from the works founded using NACs evolution strategies.

We searched related works in literature for evolution in rule-based systems, and we apply a filter to select the ones with some NACs evolution concepts, however we faced difficulty to find works with these constraints. Sections 6.1 and 6.2 review for two well-known techniques of rule-based transformations for works that deal with NACs evolution.

6.1 Petri-nets

A Petri-net (PETRI, 1962) is a formal language usually applied to describe distributed systems. As in graph grammars, Petri-nets use a visual notation to represent the rule-based models. Petri-nets have been widely studied in the literature, with different approaches to modeling different aspects of the systems. One propose that shares similarities with SOGGs is called *reconfigurable nets* (HOFFMANN; EHRIG; MOSSAKOWSKI, 2005), in addition to the petri-nets classical place-transition system, also there is a set of graph grammars rules modifying the petri-net place-transitions. This graph grammar layer can be considered as a second-order layer of the system transformation. Algebraic foundation for these systems in the framework of adhesive HLR rewriting systems are given in (PRANGE et al., 2008).

In (REIN et al., 2008) is proposed Petri-nets together with a set of graph rules with negative application conditions that also allow dynamic changes in a net. Since these rules have compatibility with adhesive HLR systems with NACs, their results are inherited from HLR theory such as local Church-Rosser, critical pairs and sequences, concurrency, etc.

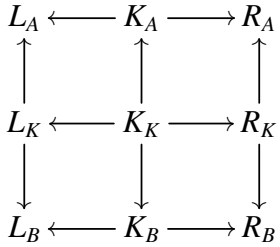
The usage of a graph grammar layer over a place transition system remind us the second-order graph grammars, where a rule-based system is acting over other rule-based system. However, we note that this “second-order” modeling is more related to our first-order graph transformation than our second-order, since in the second-order there are two adhesive HLR rewriting systems with NACs acting inter-leaved, and in these works with petri-nets there are a place-transition and an adhesive HLR rewriting systems with

NAC running. Still besides these differences, the NACs used in these papers are more related to our second-order transformations, since they act forbidding their own order rules, differently from the NACs proposed in our work where the NACs in the higher level affects the NACs of the lower level.

6.2 Triple Graph Grammars

The concept of triple graph grammars (TGGs) arises in 1994 from (SCHÜRR, 1995). They introduced TGG as a formalism to specify translations between different graph languages. The key is that each graph language is representing a model, and the translation between these languages are model transformations but formalized. Another point is that a bi-directionality of the graph relation is needed in order to maintain the verifiability of the consistency between the models.

Figure 6.1: Triple Graph Rule



TGGs use triple graph rule as basis to related two graph models, its format is described in Figure 6.1. The model relationship from a part of their specification A ($L_A \leftarrow K_A \rightarrow R_A$) to other specification of other model B ($L_B \leftarrow K_B \rightarrow R_B$) is done by an intermediate common language K ($L_K \leftarrow K_K \rightarrow R_K$) that unify concepts of the two languages. The diagram is very similar to the 2-rules, however TGGs are used in a different context such that their constraints are also different. For example, in this diagram there is no restriction to the morphism type, but in 2-rules they must to be injective, this small difference changes deeply the interpretation of these rules.

In TGG theory also the NACs are considered, some works in this area discuss them usage, specially in (SCHÜRR; KLAR, 2008; HERMANN et al., 2010; ANJORIN; LEBLEBICI; SCHÜRR, 2016) where the authors argue that NACs are real needed for TGGs in practice and they discuss the difficulties to translate them appropriately. They also emphasize that most approaches restrict the usage of NACs in some way.

The addition and verification of negative application conditions in TGG is discussed by many papers:

- A series of papers work on formal definitions of model transformations based on triple graph rules with NACs (EHRIG et al., 2009; HERMANN et al., 2010; GOLAS; EHRIG; HERRMANN, 2011; HERMANN et al., 2014). The usage of NACs appear as necessarily due their expressiveness but at same time it is difficult to maintain crucial properties needed in TGGs. Still these NACs are given in a second-order transformation sense, since they are only forbidden situations on the same level that they are modeled.

- In (SCHÜRR; KLAR, 2008) the authors focus on open problems in the interpretation and the expressiveness of TGGs. They propose a precise formalization of compulsory properties in TGG translation with NACs. These properties are very important in TGG transformations, usually they are consistency, completeness, efficiency and expressiveness. In this work the addition of NACs obviously increased the expressiveness however the completeness can not be guaranteed, which is interesting in comparison with our work when the translation of NACs also are not complete once not any NAC can be evolved.

It is important to note that the NACs semantic are added to the theory depending of the specific objective of that work, this is the main contact point between that works and ours. As the graph rules, the NACs also are utilized to modeling in specific domains, its semantic is given according to different aims.

7 CONCLUSIONS

This dissertation is about evolution and manipulation of negative application conditions in first-order graph grammars, by means of second-order graph grammars productions. Besides formal foundations, this work addresses also the implementation of the covered concepts in the Verigraph tool.

The purpose of this work emerged from future work section of (MACHADO, 2012), in which there are several questions about transformation of NACs and which effects it triggers. We believe that some of theses questions have an initial answer in this dissertation, specially the question: "how to update first-order NACs to conform them to an arbitrary rule rewriting?".

We overview the two main contributions of this work:

- *Rule evolution with NACs:* we defined evolution for rules with NACs, an extension of rule evolution. We face the problem of evolve NACs in the Section 3, we proposed the NACs evolution according to DPO rewriting, and defined that the evolved NAC may not exist when the transformation is infeasible. Thus we discovered that this evolution, unlike the older definition, is not always reversible, this result occurs because some NACs in a context do not have a correspondent in the evolved context. We expected this behavior since rules with NACs are not always reversible too.

- *Implementation of SOGG rewriting and analysis techniques:* an extension of the Verigraph tool to support classical and extended second-order transformations was presented in Section 5.1.3. All concepts covered by this dissertation were implemented in this tool, which also implements concepts of others areas of graph grammars. The second-order grammars are implemented under the `AdhesiveHLR` typeclass, in this way many operations for these categories are automatically available for SOGGs such as critical pairs and sequences, concurrent rules, etc.

7.1 Future Work

This dissertation presented a new rewriting system, it is an upgraded version of the second-order system (MACHADO, 2012). There are many investigations worth pursuing since this kind of graph transformation is not usual on graph grammars works. We list

some of open ideas below:

- *NACs creation/deletion operations:* beyond the proposes of this work, it is possible to provide algorithms to create and delete NACs programmatically. It is an extension of SOGGs where 2-rules could also have first-order NACs, as this work added NACs on the evolved rules.

- *Adhesive HLR category:* a great result will be a proof that this whole new rewriting system form an adhesive HLR category. In this case many results will be inherited from this framework such as parallelism, critical pairs, concurrence, etc. However we doubt that this extension with NACs as exactly as proposed here fits in this category, but we believe that a slightly restricted version of this grammars can form an adhesive HLR category.

- *A tool with graphical user interface:* the current tool used for second-order graph transformations has been very reliable, besides it is the only one that computes this transformations as defined here. However it fails on have a GUI, up to now this part for whole verigraph tool is in development stage. It is possible to have as result of this development, a GUI focused on second-order transformations, which is very important due the difficulty of modeling second-order rules in the current system.

REFERENCES

ANJORIN, A.; LEBLEBICI, E.; SCHÜRR, A. 20 years of triple graph grammars: A roadmap for future research. **Electronic Communications of the EASST**, v. 73, 2016.

BARESI, L.; HECKEL, R. Tutorial introduction to graph transformation: A software engineering perspective. In: SPRINGER. **International Conference on Graph Transformation**. [S.l.], 2002. p. 402–429.

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

BEZERRA, J. S.; RIBEIRO, L. Calculation and applications of concurrent rules. **Anais da 1ª Escola de Informática Teórica e Métodos Formais**, p. 135, 2016. Available from Internet: <http://etmf2016.imd.ufrn.br/Anais_ETMF.pdf>.

CORRADINI, A. et al. On the essence of parallel independence for the double-pushout and sesqui-pushout approaches. In: **Graph Transformation, Specifications, and Nets - In Memory of Hartmut Ehrig**. [s.n.], 2018. p. 1–18. Available from Internet: <https://doi.org/10.1007/978-3-319-75396-6_1>.

COSTA, A. et al. Verigraph: A system for specification and analysis of graph grammars. In: RIBEIRO, L.; LECOMTE, T. (Ed.). **Formal Methods: Foundations and Applications: 19th Brazilian Symposium, SBMF 2016, Natal, Brazil, November 23-25, 2016, Proceedings**. [S.l.]: Springer International Publishing, 2016. p. 78–94.

COSTA, A.; MACHADO, R.; RIBEIRO, L. Evolving negative application conditions. **Anais da 1ª Escola de Informática Teórica e Métodos Formais**, p. 73, 2016. Available from Internet: <http://etmf2016.imd.ufrn.br/Anais_ETMF.pdf>.

EHRIG, H. Introduction to the algebraic theory of graph grammars (a survey). In: SPRINGER. **International Workshop on Graph Grammars and Their Application to Computer Science**. [S.l.], 1978. p. 1–69.

EHRIG, H. et al. **Fundamentals of Algebraic Graph Transformation**. [S.l.]: Springer Berlin Heidelberg, 2006. (Monographs in Theoretical Computer Science. An EATCS Series). ISBN 9783540311881.

EHRIG, H. et al. **Handbook of Graph Grammars and Computing by Graph Transformation: Volume 2: Applications, Languages and Tools**. [S.l.]: world Scientific, 1999.

EHRIG, H. et al. On-the-fly construction, correctness and completeness of model transformations based on triple graph grammars. In: SPRINGER. **International Conference on Model Driven Engineering Languages and Systems**. [S.l.], 2009. p. 241–255.

EHRIG, H. et al. Adhesive High-Level Replacement Categories and Systems. In: EHRIG, H. et al. (Ed.). **Graph Transformations: Second International Conference, ICGT 2004, Rome, Italy, September 28–October 1, 2004. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. (Lecture Notes in Computer Science, v. 3256), p. 144–160. ISBN 978-3-540-30203-2.

EHRIG, H.; PFENDER, M.; SCHNEIDER, H.-J. Graph-grammars: An algebraic approach. In: **Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on**. [S.l.: s.n.], 1973. p. 167–180. ISSN 0272-4847.

GOLAS, U.; EHRIG, H.; HERRMANN, F. Formal specification of model transformations by triple graph grammars with application conditions. **Electronic Communications of the EASST**, v. 39, 2011.

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

HECKEL, R. Compositional verification of reactive systems specified by graph transformation. In: SPRINGER. **International Conference on Fundamental Approaches to Software Engineering**. [S.l.], 1998. p. 138–153.

HERMANN, F. et al. Formal analysis of model transformations based on triple graph grammars. **Mathematical Structures in Computer Science**, Cambridge Univ Press, v. 24, n. 04, p. 240408, 2014.

HERMANN, F. et al. Formal analysis of functional behaviour for model transformations based on triple graph grammars. In: SPRINGER. **International Conference on Graph Transformation**. [S.l.], 2010. p. 155–170.

HOFFMANN, K.; EHRIG, H.; MOSSAKOWSKI, T. High-level nets with nets and rules as tokens. In: SPRINGER. **International Conference on Application and Theory of Petri Nets**. [S.l.], 2005. p. 268–288.

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

MACHADO, R. **Higher-order graph rewriting systems**. Thesis (PhD) — Ph. D. thesis, Instituto de Informática - Universidade Federal do Rio Grande do Sul, 2012.

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. ISSN 0304-3975.

PETRI, C. **Kommunikation mit automaten**. Thesis (PhD) — PhD thesis, Institut für Instrumentelle Mathematik, Bonn, 1962.

PRANGE, U. et al. Transformations in reconfigurable place/transition systems. In: **Concurrency, Graphs and Models**. [S.l.]: Springer, 2008. p. 96–113.

REIN, A. et al. Negative application conditions for reconfigurable place/transition systems. In: **Proc. GT-VMT**. [S.l.: s.n.], 2008. v. 2, p. 14.

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.

ROZENBERG, G. **Handbook of graph grammars and computing by graph transformation**. [S.l.]: World Scientific, 1997.

SCHRIJVERS, T. et al. Type checking with open type functions. In: **Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming**. New York, NY, USA: ACM, 2008. (ICFP '08), p. 51–62. ISBN 978-1-59593-919-7. Available from Internet: <http://doi.acm.org/10.1145/1411204.1411215>.

SCHÜRR, A. Specification of graph translators with triple graph grammars. In: SPRINGER, HEIDELBERG. **Graph-Theoretic Concepts in Computer Science**. [S.l.], 1995. p. 151–163.

SCHÜRR, A.; KLAR, F. 15 years of triple graph grammars. In: SPRINGER. **International Conference on Graph Transformation**. [S.l.], 2008. p. 411–425.

STAHL, T.; VOELTER, M.; CZARNECKI, K. **Model-driven software development: technology, engineering, management**. [S.l.]: John Wiley & Sons, 2006.

TAENTZER, G. **Parallel and distributed graph transformation: Formal description and application to communication-based systems**. Thesis (PhD) — Technische Universitat Berlin, 1996.

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.

TAENTZER, G. et al. Model transformation by graph transformation: A comparative study. In: **MODEL TRANSFORMATIONS IN PRACTICE WORKSHOP AT MODELS 2005, Montego Bay, Jamaica**. [S.l.]: Springer, 2005. (Lecture Notes in Computer Science, v. 3713).

VARRO, G.; SCHURR, A.; VARRO, D. Benchmarking for graph transformation. In: IEEE. **2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)**. [S.l.], 2005. p. 79–88.