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

RODRIGO MACHADO

Higher-Order Graph Rewriting Systems

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

Prof. D. Leila Ribeiro
Advisor

Prof. D. Reiko Heckel
Coadvisor

Porto Alegre, Janeiro de 2012

CIP – CATALOGING-IN-PUBLICATION

Machado, Rodrigo

Higher-Order Graph Rewriting Systems / Rodrigo Machado. – Porto Alegre: PPGC da UFRGS,

.

180 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS,

. Advisor: Leila Ribeiro; Coadvisor: Reiko Heckel.

1. Graph Transformation Systems. 2. Double-Pushout Approach. 3. Higher-Order Functions. 4. Aspect-Oriented Modelling. 5. Formal Methods. I. Ribeiro, Leila. II. Heckel, Reiko. III. Título.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Prof. Leila Ribeiro for accepting me as her student and guiding me through this research effort. Her motivation and example during the period of this course have been essential for me, specially in the (many) moments my own doubts were troubling me the most. Thanks to her technical knowledge, ethics, guidance and (many times) patience, I was able to overcome all the issues that appeared and complete this work.

I also thank very much Prof. Reiko for accepting to be my co-advisor, and for having received me for seven months at the University of Leicester. It was a pleasure to be able to work with him and his students. I must thank Tamim Khan for his friendship and support (specially in the weekends of work and study), along with Fawad, Niaz, Daniela, and all other colleagues and friends that I was able to meet during my time at Leicester. I also thank Prof. Uwe Wolter very much for its helpfulness towards early versions of what has become my thesis. I was really lucky for being able to meet so many competent and nice researchers.

I must not forget to thank the professors of the Informatics Institute here at UFRGS, which I had the pleasure to learn from, as a student, and also to collaborate with, as a teaching assistant. In special, I would like to thank Profs. Alvaro Moreira, Luis Lamb, Paulo Blauth, Lucio Duarte, Rafael Bordini, Daltro Nunes and Tiaraju Diverio. I must also acknowledge the several friends I made here, which helped me so many times during this course. Starting with Luciana Foss and Simone Costa, for helping me while I was starting in the area of graph grammars. Also Fernando Rubbo, for the time we we took into grasping the semantics of AspectJ. For the support and also nice conversations having coffee, I would like to thank Márcia Cera, Rodrigo Kassick, Marnes Hoff, Clarissa Marquezan, Karina Roggia, Márcio Dorn, Alysson Machado, Tales Heimfart, Fabiane Dillerburg, Ramon Medrado, Germano Caumo, Cláudio Fuzitaki, among others friends I unfortunately cannot mention due to the lack of space.

It is essential to acknowledge the governmental agencies that provided me financial support during those years of study and research. I must thank CAPES, for funding my initial year during the course and also my scholarship period at Leicester, and CNPq, for supporting me during the remaining of the course. In particular, I also thank the secretary of PPGC for being so helpful in finding solutions for problems of bureaucratic nature.

Finally, I need to thank my parents for supporting me all those years, and my friends and family for understanding my absence so many times. Last but not least, a special acknowledgement goes to my girlfriend Márcia Bohrer for all her partnership and care, in particular when I was at Leicester and also during the last year of this project. I extend this acknowledgement to her parents, which have been equally supportive during all this time. Thank you all.

CONTENTS

LIST OF SYMBOLS

$A, B, X, Y, \ldots$	objects in categories (sets, graphs, …)
$f : A \to B$	morphism from A to B, usually function or graph morphism
$f = \{\ldots, x \mapsto y, \ldots\}$	function definition (by elements)
$f(a) = expression$	function definition (by expression)
$dom(f)$	domain/source of the function/morphism f
$cod(f)$	codomain/target of the function/morphism f
$img(f)$	image (range) of function/morphism f
$f(x)$	result of function application on element x
$f(X)$	result of function application on all elements of set or graph X $f(X) = \{y \mid x \in X \wedge f(x) = y\}$
$f^{-1}(y)$	reverse image on y: $f^{-1}(y) = \{x \mid x \in dom(f) \wedge f(x) = y\}$
$[x]_{\equiv}$	equivalence class of x: $[x]_{\equiv} = \{y \mid x \equiv y\}$
$p = L \xleftarrow{l} K \xrightarrow{r} R$	graph rule
$\alpha = a \xleftarrow{f} b \xrightarrow{g} c$	graph 2-rules
m, n	matches (both for rules and 2-rules)
(N, p)	graph rule with NACs
(N, α)	graph 2-rule with NACs
$m \vDash \nu$	match m satisfies NAC ν
$m \vDash N$	m match satisfies all NACs in N
$G \overset{p,m}{\Longrightarrow} H$	DPO transformation of graph G into graph H using rule p and match m
$p \overset{\alpha,m}{\Longrightarrow} q$	DPO span transformation of span p into span q using 2-rule α and span match m (until Section 4.5). After Section 4.5, it represents *second-order graph rewriting* (see Notation 82)
$p \overset{\alpha,m}{\Longrightarrow}_2 q$	DPO transformation of graph rules *avoiding* rule invalidation
$p \overset{\alpha,m}{\Longrightarrow}_\mu q$	DPO transformation of graph rules *correcting* rule invalidation
$\mathcal{S}(\alpha)$	calculated minimal rule-preserving set of NACs
$D = \coprod R$	coproduct of a collection R of objects
$D \overset{\alpha,m}{\Rightarrow} D'$	coproduct rewriting
$\mathcal{A}(D)$	active injections of coproducts
$\mathcal{G}_0, \mathcal{G}_1, \ldots$	graph grammars (any kind)
$\mathcal{G} \overset{(1,g,r,m)}{\Longrightarrow} \mathcal{G}'$	first-order derivation of a second-order graph grammar
$\mathcal{G} \overset{(2,r,\alpha,m)}{\Longrightarrow} \mathcal{G}'$	second-order derivation of a second-order graph grammar

LIST OF FIGURES

ABSTRACT

Software systems are not static entities: they usually undergo several changes along their development and maintenance cycles. Software evolution may be required for several reasons, such as the inclusion of new functionalities, the correction of errors or even as part of the system semantics, as it is the case of aspect-oriented systems. However, it is usually not trivial to foresee how structural changes can affect the system behaviour, since system components often interact in very complex ways, and even trivial modifications may introduce new problems.

Graph transformation, also known as graph rewriting, has been used throughout the years as an important paradigm for system modelling and analysis. Models based on graph transformation, such as graph grammars, allow an intuitive but formal representation of the system behaviour, allowing the usage of analysis techniques such as model checking and static analysis of rule interaction. The theory behind graph transformation is quite general, and has been studied since the 1970s. However, it still lacks a general notion of higher-order rewriting that would allow a natural definition of model transformations for graph grammars. The lack of general second-order characterization presents difficulties for employing graph grammars as targets of model transformations, and studying how model transformations affect their natural behaviour.

In this thesis we address the problem of modelling and analysing systems undergoing programmed modifications in the context of graph grammars. We use the generalization of the double-pushout approach for graph rewriting as a principle for defining simultaneously the system semantics and structural modifications. To achieve this, we introduce a notion of second-order graph rewriting that acts on graph transformation rules. Based on second-order rewriting we are able to define second-order graph grammars, models equipped with a first-order layer, representing the original system execution, and a second-order layer, representing a model transformation. Using second-order graph grammar we can encode simultaneously model transformations and system execution, allowing us to formally relate them. Moreover, we propose new techniques to investigate the effect of rule modification over their effect on graphs. As an application example, we characterize aspect-oriented constructions for graph grammars, and discuss how to relate the aspect weaving layer with the base system semantics.

Keywords: Graph Transformation Systems, Double-Pushout Approach, Higher-Order Functions, Aspect-Oriented Modelling, Formal Methods.

Sistemas de Reescrita de Grafos de Alta Ordem

RESUMO

Programas sofrem diversas modificações ao longo das etapas de desenvolvimento, implantação e manutenção. A evolução de um software pode ter várias causas: correção de erros, inclusão de novas funcionalidades ou até mesmo, como é o caso de programas orientados a aspecto, transformações estruturais podem fazer parte da semântica do sistema. Apesar de modificações serem comuns, não é tarefa trivial prever como estas afetam o comportamento dos programas, já que os componentes de software normalmente interagem de forma complexa, o que faz com que mesmo pequenas alterações possam introduzir comportamentos indesejados.

Transformação de grafos, também conhecida como reescrita de grafos, é um importante paradigma para modelagem e análise de sistemas. Modelos baseados em transformação de grafos, como gramáticas de grafos, permitem uma modelagem ao mesmo tempo intuitiva e com semântica precisa, permitindo a aplicação de técnicas de análise como verificação de modelos e análise de par crítico no estudo do comportamento de sistemas. A teoria por trás de transformação de grafos vem sendo desenvolvida a várias décadas, e atualmente está descrita de uma forma bastante abstrata. Contudo, ainda não possui uma definição natural de reescritas de alta ordem, que facilitaria a definição de evolução de especificações compostas por regras de reescrita de grafo, tais como gramáticas de grafos.

Nesta tese são abordadas a modelagem e a análise de sistemas sob modificações programadas no contexto de gramáticas de grafos. A generalização da abordagem de pushout duplo para reescrita de grafos é utilizada como o princípio geral para descrever, simultaneamente, a semântica do sistema e modificações estruturais. Para tal, introduzimos uma noção de reescrita de segunda ordem para modificar a estrutura de regras de transformação de grafos, e usando isso, definimos modelos equipados simultaneamente de regras de primeira e segunda ordem, chamados *gramáticas de grafos de segunda ordem*. Através destes modelos podemos representar simultaneamente transformações estruturais e execução do sistema, e relacionar formalmente ambos tipos de reescrita. Também propomos novas técnicas para investigar o efeito da modificação de regras sobre a aplicação destas. Finalmente, como um exemplo de aplicação da teoria, caracterizamos construções de sistemas orientados a aspectos através de gramáticas de grafos de segunda ordem, e discutimos como utilizar as novas técnicas para estudar o efeito da combinação aspectual sobre o sistema inicial.

Palavras-chave: Sistemas de Transformação de Grafos, Abordagem de Pushout Duplo, Funções de Alta Ordem, Modelagem Orientada a Aspectos, Métodos Formais.

1 INTRODUCTION

This chapter introduces the two basic ideas behind this thesis: higher-order constructions and graph rewriting. Furthermore, it discusses and provides initial answers to the following questions:

1. What is the general problem we aim to study?
2. Why use the graph rewriting framework?
3. Which kind of system can we model and analyse using graph grammars with higher-order capabilities?

Finally, it provides an overview of the structure of the text. We start by contextualizing the overall problem to be addressed.

1.1 Problem statement

Software systems are not static entities: they have a life cycle, undergoing several changes during its useful life span. Software evolution may occur for several reasons, such as the inclusion of new functionalities, the correction of errors or even to improve the system organization. In some contexts, system modifications may be part of the system behaviour rather than a component of the software development cycle. This is the case of Aspect-Oriented Programming (AOP) paradigm (KICZALES et al., 1997), which advocates that some application requirements (called crosscutting concerns) should be implemented by means of a particular kind of implicit transformation named aspect weaving. Following this view, aspects may be seen as a compact descriptions of such transformations. As it is expected from any program transformation, aspect weaving raises concerns regarding the interaction of the injected code with the original system constructions. It is well known that, while aspects increase modularity of crosscutting concerns, they also introduce unintended behaviours. In order to obtain a compromise between power and control, there is a broad research effort regarding interaction analysis of aspect-oriented weaving (DOUENCE; FRADET; SÜDHOLT, 2004; MEHNER; MONGA; TAENTZER, 2006; AKSIT; RENSINK; STAIJEN, 2009; DJOKO; DOUENCE; FRADET, 2008; MOSTEFAOUI; VACHON, 2007).

We argue that aspect-oriented systems fall into a more general category: systems equipped with two semantic layers, one describing the normal system behaviour, and other describing system transformations. We will refer the transformation layer as an “upper” or “higher-order” layer, and the original behaviour as the “base” or “lower order”. Another example could be to consider traditional systems undergoing programmed modifications

such as refactorings, refinements or other kinds of system evolutions. From now on, we will generically refer to such systems as "higher-order systems". An important question for any kind of system is how modifications in the system (the higher-order layer) affect the system observed behaviour, i.e. as viewed by an interacting external system. The behaviour can be either

- Modified: as in error correction;
- Maintained: as in the case of refactorings and the weaving of non-interfering aspects (log policies, for instance);
- Extended: mainly the addition of new functionalities.

Given a particular system, to be able to predict how the observed behaviour is affected by higher-order transformations is usually a very complex and in most cases not feasible task. It involves to relate three quite distinct abstract objects, which may be specified in very distinct ways:

- The system modification;
- The overall system execution; and
- The observed behaviour of the system (from an external observer, for instance).

As an example of such system, we could consider a web-service implemented in AspectJ (KICZALES et al., 2001) – the most popular AOP superset of Java – with observational behaviour being defined as all sequences of invocations from clients. There are several difficulties regarding the analysis of such systems:

- Mostly often, general purpose programming languages do not have a complete formal description of its semantics, or they are extremely complex for verification purposes;
- Even provided a convenient formal description of the language semantics, there is the need to relate the system observation with the system transformation, usually defined using quite distinct paradigms. For instance, in this particular situation we would be required to relate the combination semantics of AspectJ with the operational semantics of Java.

The common practical solution to guarantee that system behaviour is maintained across evolution steps is by means of programmed test suites. A given set of test cases witnesses the response of the system under several situations, and whenever the system changes, the test suite has to be rerun to assure that non-intended modifications were not introduced. Testing is very useful in practice, but may not suffice when stronger guarantees regarding the system behaviour, such as safety or security constraints, are required. This happens because tests generally do not fully cover the potentially infinite behaviour of the system.

In this thesis we address this problem at an abstract level rather than at code level. This implies to employ a sufficient expressive system model where the notion of execution, system transformation and observation may be naturally and uniformly defined. By working at specification level, there is the possibility to reduce the inherent complexity of considering the full-scale system. Furthermore, this kind of approach could be integrated into

the Model-Driven Software Engineering (STAHL et al., 2006) approach for system development, where system models have central role within the development cycle, serving as a basis for code generation. In this thesis we use models based on graph transformation as a formal framework, and the next session discusses why this is an adequate choice for such context.

1.2 Graph transformation systems

Graph rewriting refers to rule-based modification of graphs. The main advantage of this principle is that it allows the creation of visual and intuitive models endowed with a precise semantics. Graph grammars are models based on graph rewriting for which the initial state is given by a graph, and the dynamics is defined by a set of rewriting rules. System evolution is determined by applications of rules starting in the initial graph, and it is guided by the graph topology. Before introducing the basic definitions and concepts regarding graph grammars, we would like to focus abstractly on some of their interesting characteristics.

Visual and intuitive representation of states: graphs as data structures have a natural diagrammatic representation, from which we may obtain a visual and intuitive understanding of states with a (potentially) very complex structure.

Typing discipline: Graph grammars usually employ a typing discipline that allow to define constraints on the structure of graphs by means of schema (or type) graphs, in a very similar fashion as the meta-object facility (MOF) of UML diagrams (OBJECT MANAGEMENT GROUP, 2005).

Simple semantics: The behaviour of a graph grammar is determined by its set of rewriting rules and a initial condition represented by a graph. This is a very simple execution model, if compared with more detailed behavioural models such as UML activity diagrams, for instance.

Abstract notion of rewriting: The algebraic approach for graph rewriting is generalized in the sense that the same rewriting theory can be applied uniformly in distinct graph-based models. This allows an easy lifting of the rewriting mechanism, usually used to describe base model semantics, to describe higher-order constructions.

Versatility: Rule-based rewriting is a general principle which may be applied to describe distinct scenarios. For instance, in the context of visual languages we can define language generation, operational semantics, model transformation and aspect weaving through the same rewriting mechanism. This allows the usage of analysis techniques for arbitrary rewriting to be used to relate the distinct scenarios.

Tool support: There are currently several tools for graph rewriting which allow system representation and analysis. Some famous tools are AGG (TAENTZER, 2000) and Groove (RENSINK, 2004).

Models based on the graph rewriting principle are popular to describe both model transformations and operational semantics in the context of visual languages (TAENTZER

et al., 2005). In this proposal, we intend to apply graph rewriting to model higher-order systems, describing simultaneously model transformation and system execution through sets of rules, and analyze the nature of their interaction. Regarding observation, there are several semantic models for graph grammars, ranging from very concrete, sequential models to more abstract ones, where parallelism is explicitly represented. In this sense, a semantic model may be used as a characterization of the intended observation policy of the system. Concerning higher-order systems, the abstract descriptions of the rewriting principle for graphs serves as a guidance towards the development of a theory of second-order graph rewriting rules, i.e., (second-order) rules that modify (first-order) graph transformation rules. Currently analysis techniques focus only on a given level of rewriting. Besides representing higher-order rewriting, we also explore the extension of currently available analysis techniques, in particular as critical pair analysis, to compare system transformation rewriting steps with the system execution steps. Before detailing more of the proposed approach, let us clarify what we mean by the expression "higher-order".

1.3 Higher-order systems

Whenever a given system or language is referred to as "of higher-order", the common understanding is that it allows the representation and manipulation of its own constructions as its data. For instance, higher-order functions are the ones which may receive functions as arguments or return functions as results. A famous example from the functional programming scenario is *map*, an operation that receives a function f and a list of values l and returns a list of values obtained by applying f to all elements of l. The abstract definition of *map* is shown below:

$$map(f, [x_1, \ldots, x_n]) = [f(x_1), f(x_2), \ldots, f(x_n)]$$

In Logic, higher-order means to allow predicates to take other predicates as arguments, and thus can be used to formalize self-referential sentences such as "this sentence has 100 characters" or "the previous sentence is false". Such ability is particularly important in meta-logic, since one of its most famous results – Göedel's incompleteness theorem – relies on an encoding of predicates and their provability as natural numbers and arithmetic predicates. In Computer Science, the notion of higher-order is usually associated with the dual relationship between program and data. For example, in the von Neumann architecture, both data and programs are encoded in binary representation and stored in the same address space. The distinction between a number and an instruction is done only when these elements are interpreted, and depends on their expected role at the moment they are accessed.

A popular formalism where terms may represent both data and procedures is the lambda calculus (BARENDREGT, 1992), a language proposed in the 1930s by Alonzo Church. Terms in the lambda calculus follow the simple abstract syntax

$$t ::= x \mid \lambda x.t \mid t_1\ t_2$$

where $x \in X$ is a variable name drawn from a countable set X, $\lambda x.t$ is a function where the formal parameter is x and the expression it calculates is t, and $t_1\ t_2$ represents a call to t_1 as a function receiving t_2 as an argument. The interesting fact about this calculus is that the role of a given term t_0 only depends on the position it occurs within an application: if it occurs on the left (e.g. in $t_0\ t$) it is interpreted as a function; on the right (e.g. in $t\ t_0$) as data.

Hence, the formalism allows to describe several situations involving function manipulation, as, for instance, to declare the self-application $t_0\ t_0$ of term t_0. Lambda calculus serves as an idealized programming language, and it is the basis of the functional programming paradigm. In practical programming languages, encoding and decoding of expressions are handled by operators. For instance, in LISP (STEELE, 1990) the translation between data and procedures is delegated to the functions `quote` and `eval`. The function `quote` takes as input an arbitrary expression, and outputs its respective representation as data (i.e. it stops the evaluation of the term). The function `eval` takes as input a list of data elements and returns the result of evaluating it as code, which can be subsequently used. This flexible design make LISP (and derived languages such as Scheme) very adequate to develop techniques that involve program manipulation, also referred as meta-programming. Aspect weaving, due to its intrinsic effect of modifying programs, may be seen as a particular kind of meta-programming. This view is reinforced in (TUCKER; KRISHNAMURTHI, 2002) and (SANJABI; ONG, 2007), where aspects are implemented using higher-order constructions of the base language.

Although being a powerful mechanism, higher-order constructions in general introduce hardships in system analysis. For example, code analysis techniques such as type systems or static analysis have their efficiency compromised if we allow dynamic code loading, requiring the presence of run-time machinery. Similarly, higher order is not easy to grasp in the context of system modelling and it has not been generally explored apart from a few particular contexts (such as Petri nets). This has been an issue, for instance, in the extension of aspect-oriented concepts over UML models. By working with a model which is simple enough in its execution, such as graph grammars, we are able to overcome some of the intrinsic complexity introduced by higher-order constructions. We obtained intuition regarding the representation of higher-order constructions from idealized higher-order languages, in special lambda calculus.

1.4 Example: evolving place-transition net

In this section we present a small example to illustrate the kind of situation we describe and analyse in this thesis. In the example, the base system is represented by a place-transition (PT) net, a model for concurrent systems (see (REISIG; ROZENBERG, 1998) for precise definitions). The transformation over the base model, implemented by means of graph rewriting, adds a *step counter* place to the model. Finally, we try to characterize how the semantics of the base model is affected by the transformation.

A place-transition net is defined by a set of places, visually depicted as circular nodes, a set of transitions, represented by means of square nodes, and connections between places and transitions, represented by edges. Figure 1.1 depicts the place-transition net P, where $\{p_1, p_2, p_3, p_4\}$ is the set of places and $\{t_1, t_2, t_3\}$ the set of transitions. Each transition is connected to a subset of places, named input places and drawn as edges from places to transitions, and a subset of places, named output places and drawn as edges from transitions to places. For instance, $input(t_1) = \{p_1\}$ and $output(t_1) = \{p_2, p_3\}$. A state of the system is a distribution of a finite number of tokens into places. Tokens are graphically represented as small black circles inside the nodes. The initial state of P contains only two tokens in place p_1.

System execution is given by triggering a transition, i.e. by removing one token in each of its input places and creating one token in each of its output places. If two or more transitions are enabled (i.e. all of their input places have tokens), then the choice of

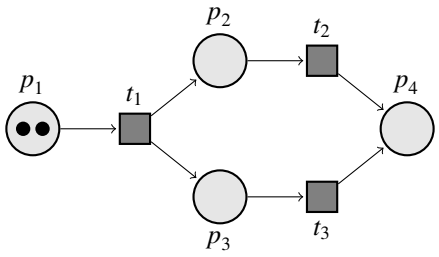


Figure 1.1: Place-transition net.

execution is nondeterministic. The transition system representing all possible sequential executions of P, one transition triggered each time, is shown in Figure 1.2. We have used the graph grammar tool Groove (RENSINK, 2004) to obtain this transition system, using the graph grammar representation of the place-transition net. States are actually multisets of place names, e.g. s1 = $\{p_1, p_2, p_3\}$. The initial state is s0 = $\{p_1, p_1\}$, and the state s13 = $\{p_4, p_4\}$ is said to be final because no transitions can be triggered from it.

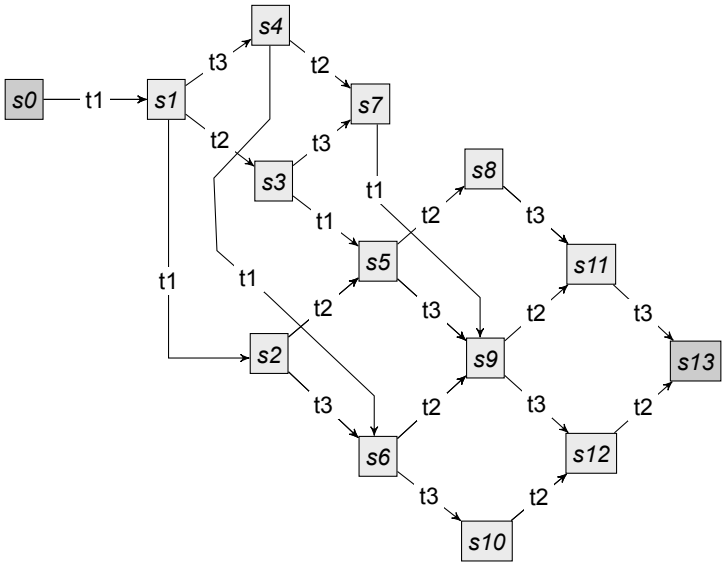


Figure 1.2: Full execution of place-transition net.

In the following analysis, we will assume from the reader some familiarity with the basic concepts regarding graph grammars. We suggest readers uncomfortable with the language to go through the initial definitions of Chapter 2 before continuing.

1.4.1 System execution as graph grammar

It is part of the folklore of the graph transformation area that place-transition nets can be seen as a special case of graph grammar. Each place-transition net corresponds to a graph grammar where the type graph is discrete, i.e., it contains only nodes. Places are represented as node types and tokens, as instance nodes. Distributions of tokens into places is given by the typing morphism of the graph being rewritten, and thus the start graph of the grammar represents the initial token distribution of the net. Transitions are represented by rules with an empty interface that delete and create nodes according to the respective input and output places. In Figure 1.3 we have the graph grammar representation of the place-transition net P shown in Figure 1.1.

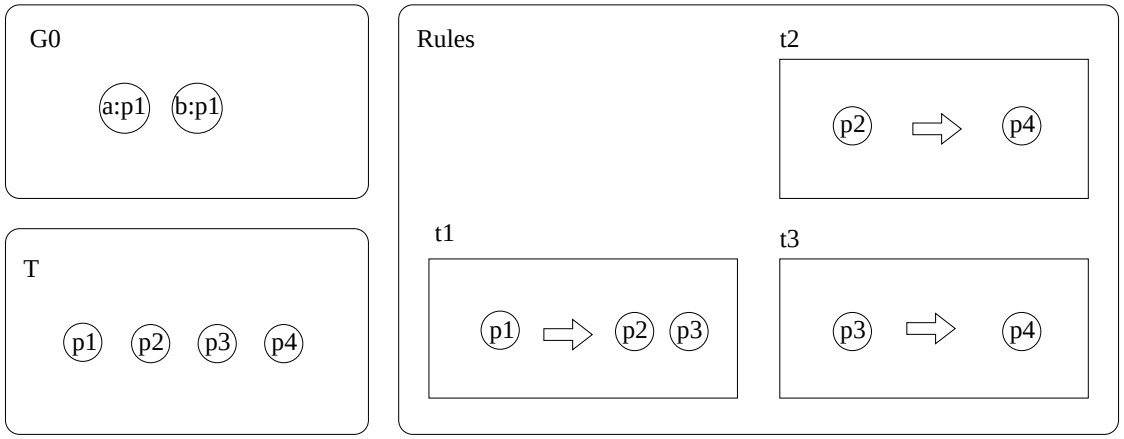


Figure 1.3: Place-transition net as graph grammar.

1.4.2 Model transformation as graph grammar

Place-transition nets are visually described by graphs, and thus it is very natural to consider the implementation of model transformations by means of graph rewriting. A simple example is the addition of a global step counter for P, i.e., the addition of a place where all transitions deposit one token every time they are triggered. At the end of the execution, this place will contain as many tokens as triggered transitions. Figure 1.4 depicts a graph grammar $\mathcal{G}$ implementing such transformation. Notice that the type graph T represents the meta-structure of place/transition nets augmented with a "control" edge type counter, and also that the initial graph G_0 encodes the net P together with its initial token distribution. The rule createPlace creates a single place in the net, being this place "marked" by counter edge. A negative application condition ensures that createPlace will be only applied if no other counter place exists in the net. The rule createOutput connects an already existing transition to the counter place, and likewise, its negative application condition ensures that each transition will have only one edge targeting the counter.

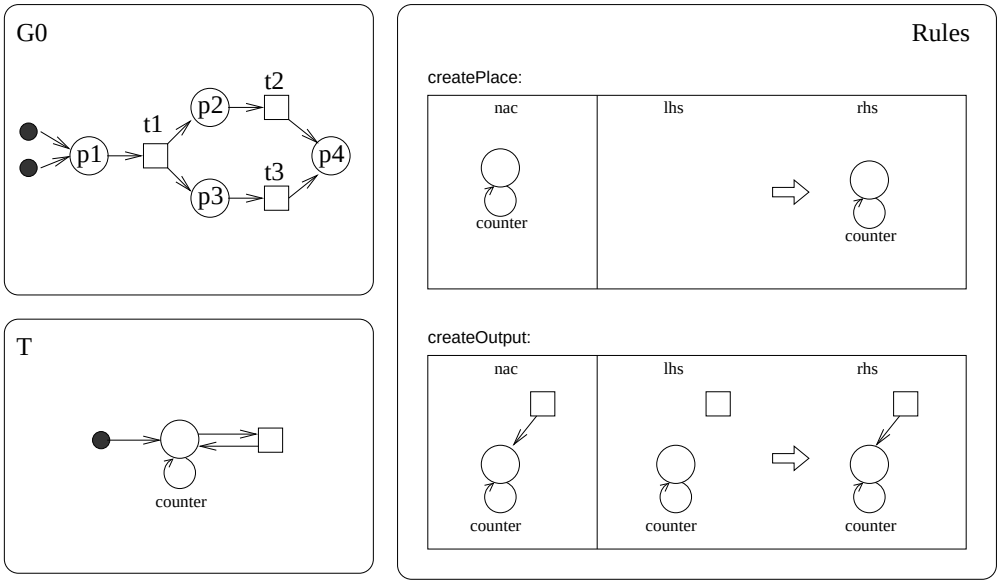


Figure 1.4: Model transformation as graph grammar.

The execution of this graph grammar applies createPlace once, and then createOutput

as many times as there are transitions in the initial graph (exactly three times for P). The order the output edges are created is not particularly relevant, since all rewritings do not interfere with each other.

1.4.3 Inter-level interaction

This view of nets as graph grammars allows us to consider the whole setting by means of a higher-order graph grammar, where the base grammar rules are affected by the upper-level rewriting. Although small, the scenario comprised by P and the transformation $\mathcal{G}$ allows us to observe some facts:

1. Either the base level which carries tokens, and the model level where the transitions are modified, can be modelled by the same principle (graph rewriting).
2. In this particular example, both levels of rewriting are terminating.
3. If we consider the complete application of model transformation (MT) rules and base system operational semantics (OS) in a non-interleaved way, i.e. all rules of one level are executed before the rules of the other level, we obtain different final systems, as seen in Figure 1.5. If we allow interleaved execution between levels, a variable number of tokens in the counter place may be found when the rewriting process finishes.

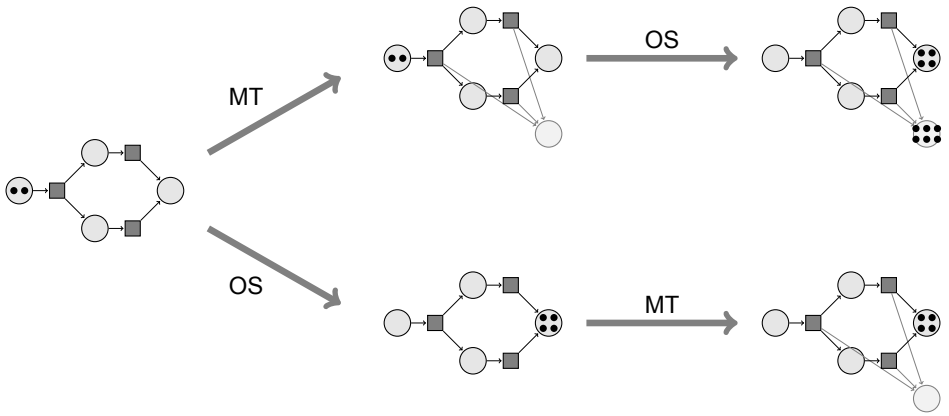


Figure 1.5: Operational semantics (OS) versus model transformation (MT).

Based of the previous observations, we can also inquire about the properties of such system. Some relevant questions are:

- In which way the model transformation affects the base system execution?
- How conflicts between the two levels of rewriting are characterized?
- Is the observable system behaviour preserved?

Whenever we consider the effect of model transformation over the execution of a base model, we must keep in mind which particular semantic model we are interested in. If we consider as semantic model the resulting graph whenever the system terminates, then the presented model transformation actually modifies the base system semantics. On the

other hand, if we consider the "shape" of the transition system representing the execution of P (as shown in Figure 1.2) as the semantic model, it may be claimed that this model transformation does not interfere with the semantics, since (intuitively) adding a new output place does not affect the applicability of rules. A precise definition for what constitute an observable behaviour of the system, or the particular semantic model used to characterize the system behaviour, is a requirement before any characterization of interference. Provided a particular observational model, we may then ask if it has been maintained, extended or subtracted. Furthermore, we may wish to obtain more information regarding what has been updated.

Such questions are starting point of this work. We propose to address them in the context of graph grammars. A notion of higher-order for graph rewriting is required, and to obtain this, we need to revise the theory of graph transformation, exploring the possibility of second-order rules transforming first-order rules and (possibly) other components of the specification such as the initial and type graphs. We extend some techniques available in graph rewriting, such as critical pair analysis, to address interference between transformations and operational semantics.

It is worth mentioning that dynamic place-transition systems like the one presented in the example, i.e. extended with dynamic modifications given by graph rewriting, have already been vastly studied in the literature (BADOUEL et al., 2003; LLORENS; OLIVER, 2004; HOFFMANN; MOSSAKOWSKI; PARISI-PRESICCE, 2005; PRANGE et al., 2008). Moreover, the idea of having rules transforming rules in graph transformation is not entirely new, since it has appeared previously in some approaches such as (GÖTTLER, 1999) and (PARISI-PRESICCE, 2001). However, there is not yet an adequate characterization of higher-order principles for the DPO approach for graph rewriting in the literature. We foresee the application of graph grammars with higher-order rules as a framework in the formal study of graph grammars with aspect-oriented capabilities. In this direction there are only few proposals such as (WHITTLE; JAYARAMAN, 2007) and (MEHNER; MONGA; TAENTZER, 2006).

1.5 Thesis aims

The main objective of this thesis is to address the following problem:

> *How to describe and analyse the interaction between model transformations and the system semantics in the context of graph grammars.*

We propose to following approach:

1. Employ graph transformation as a basic principle to describe both model transformation and base model semantics. This requires the development of a notion of higher-order for graph rewriting;

2. Define an extension of the basic graph grammar model that allows the definition of both model transformation and base system execution. Model transformations would make use of higher-order rewriting, while normal system behaviour is given by first-order rewritings.

3. Define a notion of interference between rules in two distinct rewriting levels. This provides a notion of "inter-level" conflict, which we use for analysis purposes.

4. Apply the previous results in the context of aspect-oriented graph grammars, characterizing the interaction of aspect weaving and base system execution.

Hence, the *hypothesis* we intend to confirm is

> *Higher-order principles in the context of graph rewriting can be adequate in modelling and analysing systems undergoing programmed modifications.*

The study of the interaction between second-order and first-order layers in specifications may be useful in several contexts, and here we briefly mention some of them:

1. Verification of properties across model transformation;

2. Modelling of aspect-oriented capabilities in visual models;

3. Modelling of adaptive systems, i.e., systems whereas the rules for execution can change dynamically.

1.6 Methodology and text organization

In this section, we summarize the structure of the remaining chapters, which depicts the overall structure of this thesis.

Chapters 2 and 3: technical background

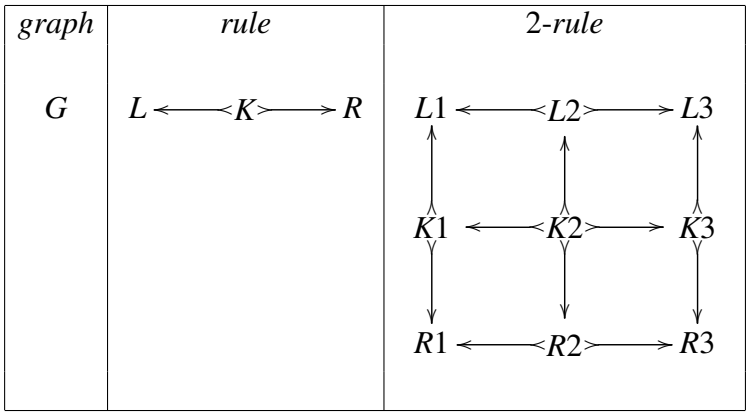
These chapters provide a review of the graph transformation and higher-order functions in the lambda calculus formalism. On Chapter 2 we focus on the DPO approach to graph transformation, introducing graph grammars and some important results of the area such as the characterization of conflicts and dependencies between rewriting steps, and also important theorems such as local Church Rosser. We mention the critical pair analysis technique for analysis of specifications.

Chapter 3 presents a very brief review of the untyped and simply typed variations of the lambda calculus formalism, focusing on the characterization of higher-order terms and types. The contribution of this chapter is a discussion providing analogies between higher-order types in the typed lambda calculus setting and rule modifications in the graph transformation setting.

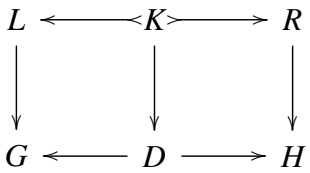
Chapters 4, 5 and 6: theoretical development

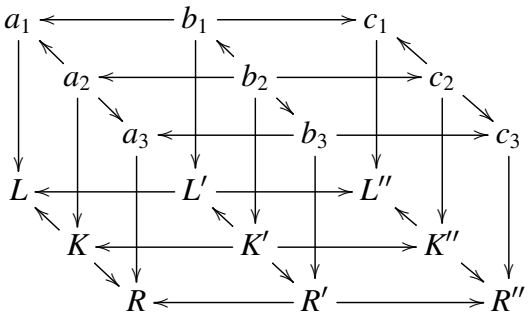
These chapters comprise the main theoretical contributions of this thesis. The main objective of Chapter 4 is to establish an adequate notion of second-order rewriting. Roughly speaking, the concept of *higher-order rule* is defined by the same mechanism employed to define rules from graphs in the DPO approach. First we need to put graph rules and respective transformations in a categorical context. For such, we explore two possibilities: the category of arbitrary spans over T-**Graph**, and its subcategory T-**Rules** of graph rules. We define a notion of second-order rules, which we name *2-rules*, formed by the same principle that constructs graph rules

from graphs. The following diagram depicts a visualization of a graph, a graph rule and a graph 2-rule as diagrams in T-**Graph**:



The next step is to ensure that T-**Span** or T-**Rules** are suitable for DPO rewriting, in other words, if they can be characterized as adhesive HLR categories. If this holds, we can characterize 2-rule-based rewritings of rules as DPO diagrams in the category in question, just like rule-based rewritings over graphs are DPO diagrams in the category T-**Graph**. We have found that T-**Span** is adequate for modification of rules, provided a mechanism to ensure that both morphisms that represent the resulting span continue to be injective after the rewriting. To enforce rule-preservation, we employ the mechanism of *span rewriting with negative application conditions*, for which we introduced the notion of *minimal rule-preserving set of NACs*. One direct advantage of this approach is that we can represent the two kinds of rewriting as diagrams in the same category T-**Graph**, as shown below, which is convenient if we want to relate both kinds of rewriting for analysis purposes.

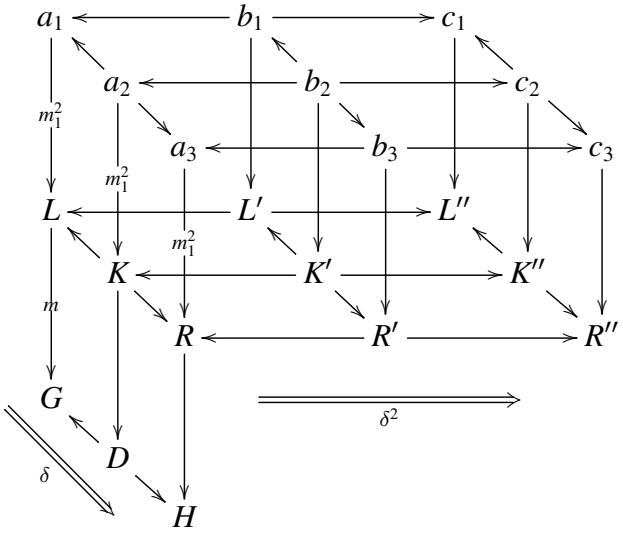




Chapter 5 discusses how to employ the notion of second-order rewriting to construct specifications. We address several details of how this realisation, such as how a single 2-rule should affects a whole collection of rules, rather than a single one. This also triggers a discussion regarding how to actually represent collections of rules, and how collections can be modified through second-order rewriting. Notions such as creation, deletion and modification of rules are defined for rule collections. Finally, we discuss how overall modifications in first-order models can be summarized through the notion of an *evolutionary span* marking what has been removed, preserved and introduced in the whole specification.

Chapter 6 presents how to represent conflicts and dependencies within second-order graph grammars, and how first-order and second-order derivations interact. Two conventional DPO rewritings $G \overset{r,m}{\Longrightarrow} H_1$ and $G \overset{r',m'}{\Longrightarrow} H_2$ from the same graph G

are said to be conflicting if the execution of one of them forbids the execution of the other on its resulting graph. In this chapter we extend this notion to evaluate the interaction between a first-order rewriting $\delta = G \overset{r,m}{\Longrightarrow} H$ and a second-order rewriting $\delta^2 = r \overset{\alpha,m'}{\Longrightarrow} r''$, by using diagrams in T-**Graph**, as show in the diagram below:



We are able to represent this interaction by means of the existence of morphisms in the diagram satisfying some essential conditions. Besides individual rewritings, we also analyse the effect of higher-order rules over the *critical pairs* of the base system. This means that a higher-order rule may affect the possible dependencies of the lower system, and thus affect its semantics. Based on such considerations, we extend the principle of critical pair analysis for second-order graph grammars, which accounts for first-order critical pairs, second-order critical pairs and inter-level critical pairs.

Chapter 7: aspect-oriented graph grammars

In Chapter 7 we provide a review of the principal concepts regarding aspect-oriented programming and aspect-oriented modelling. Then, we compare the main concepts of aspect-orientation and graph transformation, and provide a representation of aspect-related constructions for first-order systems as a second-order layer. The notion of aspect weaving is described through second-order rewritings, which allows to employ critical pair analysis for the study of interaction between aspects. Moreover, given the notion of inter-level critical pair, we can represent directly the effect of aspect weaving in the semantic of the first-order system.

Chapters 8 and 9: related work and conclusions

Chapter 8 compares our approach with others in the literature. We focus on some other proposals for modifications of rules and specifications, dynamic place-transition nets, and approaches combining aspect rewriting and graph transformation.

Chapter 9 summarizes our results, and discusses future developments.

2 GRAPH TRANSFORMATION

The research area of Graph Transformation is a discipline of Computer Science which begun in the late 1960s as a generalization of term rewriting. Methods, techniques, and results from the area have already been studied and applied in many fields of Computer Science, such as formal language theory, pattern recognition and generation, software engineering and the modelling of concurrent and distributed systems. (EHRIG et al., 2005) The main idea lies in the rule-based transformation of graphs: the system state is represented by a graph, and the application of transformation rules represent its operational behaviour. Graph transformation rules are said to be local, i.e. they describe modifications to be applied over a specific part of the graph. This allows more than one rule to be applied at the same time, being an adequate model to represent systems with non-deterministic behaviour.

There are several approaches in the literature to determine the graph rewriting process. In this chapter, we focus on the *algebraic* approach, where rules and rule applications are defined by means of constructions from Category Theory. Within the algebraic setting, there are two main variants, namely the double- and the single-pushout approach. We focus on the double-pushout approach (DPO), which inspired the current generalization of the theory, referred as *adhesive HLR systems* (EHRIG et al., 2004).

In this chapter we do not present new ideas, but rather offer an overview of the main concepts from the graph transformation field. The definitions are taken from standard references of the area, particularly (ROZENBERG, 1997; EHRIG et al., 1999, 2005). Initially we describe basic concepts such as typed graphs and then introduce the DPO approach for graph rewriting. We describe graph grammars using DPO rules, and characterize situations of conflict and parallel execution for graph transformation. Then we comment on semantic models for graph grammars, negative application conditions for rules and the usage of attributes in edges and nodes. Finally, we talk about tools, the critical pair analysis technique and adhesive HLR systems.

2.1 Basic definitions

A graph is a structure composed by a set of *nodes* with binary connections denominated *edges* (or *arcs*). When the connections between nodes do not have orientation, the graph is called *undirected*. Otherwise, it is a *directed graph* or *digraph*. In this work we will assume that all graphs are directed.

Definition 1 (Graph)**.** *A graph is a tuple $G = (V, E, s, t)$, where V and E are sets of nodes and edges, and $s, t : E \to V$ associate, respectively, a source and target node to each edge.*

Example 2. *Figure 2.1 depicts the description of a graph (on the left) together with its*

respective visual representation (on the right).

$$
\begin{aligned}
V &= \{a, b, c, d, e\} \\
E &= \{1, 2, 3, 4, 5, 6\} \\
s &= \{1 \mapsto a, 2 \mapsto b, 3 \mapsto b, 4 \mapsto a, 5 \mapsto d, 6 \mapsto e\} \\
t &= \{1 \mapsto b, 2 \mapsto c, 3 \mapsto c, 4 \mapsto c, 5 \mapsto c, 6 \mapsto e\}
\end{aligned}
$$

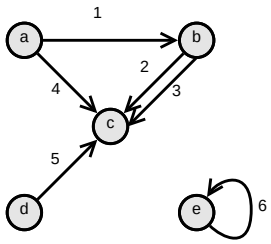


Figure 2.1: Graph.

Notation 3. We will use the term *element* or *item* to indicate either nodes or edges. By an abuse of language, we write $x \in (V, E, s, t)$ meaning $x \in V$ or $x \in E$ whenever it is not harmful to do so.

A graph morphism is a mapping of nodes and edges of one graph to, respectively, nodes and edges of another graph such that the structure of the input graph is preserved, i.e. the source and target nodes of the translation of one edge are the translations or source and target nodes of the original edge. Formally speaking:

Definition 4 (Graph morphism)**.** *Let $G = (V, E, s, t)$ and $G' = (V', E', s', t')$ be graphs. A* graph morphism *$f : G \to G'$ is a pair of functions (f_V, f_E) with types $f_V : V \to V'$ and $f_E : E \to E'$ such that $s' \circ f_E = f_V \circ s$ and $t' \circ f_E = f_V \circ t$.*

Example 5. *Figure 2.2 shows an example of graph morphism f between graphs G_1 and G_2. The mapping of nodes is represented by dashed lines, whilst the mapping of edges is represented by dotted lines. Although there are other graph morphisms between G_1 and G_2, notice that there are not morphisms between G_2 and G_1. This happens because it is not possible to associate adequately the edge* 6 *to an edge in G_1.*

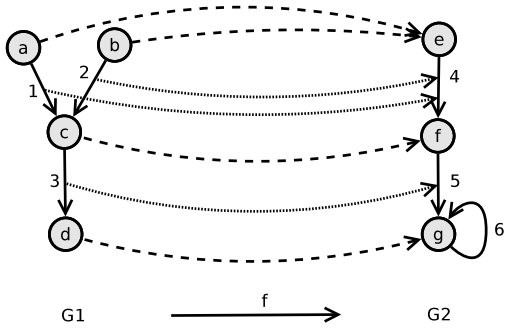


Figure 2.2: Graph morphism.

A graph morphism $m : G_1 \to G_2$ which is bijective in both source and target mappings is said to be an isomorphism, in which case the graphs G_1 and G_2 are said to be isomorphic to each other. All possible graphs related by graph morphisms constitute a category, which we call **Graph**. We assume a basic knowledge from category theory from the reader, and present all the required concepts in Appendix A.

Whenever we model situations through graphs, a useful technique is to classify both edges and vertices according to particular kinds or types. The usual approach for typing graphs is to consider a fixed graph T as a schema graph, where its elements denote the possible kinds of nodes and edges. Then, the typing of a graph G over T is represented by means of a morphism $t_G : G \to T$.

Definition 6 (Typed Graph)**.** *Let T be a fixed graph, called type graph. A T-*typed graph *$G^T = (G, t_G)$ is a graph G together with a total graph morphism $t_G : G \to T$.*

Notation 7. Whenever the type graph T is understood from the context, we may refer to G^T as simply G.

Example 8. *Figure 2.3 depicts a typed graph G^T. Nodes of T represent entities such as computers (*pc*), data servers (*server*), messages (*msg*) and data packages (*data*). The edges represent where the entities are located: data nodes may be directly stored into servers, computers or embedded in messages. Messages may be located in user computers or in servers. The nodes and edges of the instance graph G are mapped to elements of T by means of the type morphism t_G (which is depicted implicitly by the use of corresponding element shapes).*

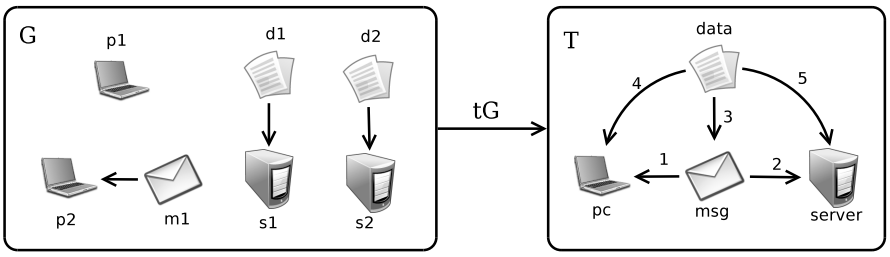


Figure 2.3: Typed graph.

Since the typing morphism classifies the graph elements into types, the notion of morphism between typed graphs has to ensure that each element in the source graph is mapped to an element with the same type in the target graph.

Definition 9 (Typed Graph Morphisms)**.** *A morphism of T-typed graphs $f : G^T \to {G'}^T$ is a total graph morphism $f : G \to G'$ such that $t_{G'} \circ f = t_G$.*

Similarly to untyped graphs, when a type graph T is fixed, the class of all T-typed graphs and respective typed graph morphisms constitute a category, named T-**Graph**. The typing mechanism in particularly interesting because of the natural association between meta-models (in the UML family of diagrams) and type graphs.

2.2 The double-pushout approach

The double-pushout approach (DPO), introduced by (EHRIG; PFENDER; SCHNEIDER, 1973), was the first characterization of graph rewriting by means of categorical constructions. In this approach, graph rewriting rules are represented as pairs of typed graph morphisms with the same origin. It is usual to consider both morphisms to be injective, and we follow this characterization for simplicity. We refer the reader to (HABEL; MÜLLER; PLUMP, 2001) for a comprehensive analysis of non-injective rules in the double-pushout approach.

Definition 10 (Span)**.** *A span in a given category $\mathscr{C}$ is a pair of morphisms (a, b) with the same source, as shown in the following diagram*

$$A \xleftarrow{a} X \xrightarrow{b} B$$

Dually, a co-span is a pair of morphisms (c,d) *having the same target object, as shown below.*

$$C \xrightarrow{c} Y \xleftarrow{d} D$$

A (co-)span (a,b) *is injective (monic) iff both a and b are injective (monomorphisms).*

Definition 11 (Graph rule)**.** *A graph rule is an injective span* $q : L \overset{l}{\leftarrowtail} K \overset{r}{\rightarrowtail} R$ *in the category* T-**Graph**. *The class of all graph rules is denoted* T-**Rules**.

Notation 12. We may denote the rule $q : L \overset{l}{\leftarrowtail} K \overset{r}{\rightarrowtail} R$ by (l,r) if L,K,R are understood from the context, or by $L \leftarrow K \rightarrow R$ if $K = L \cap R$ and l,r are inclusions. Graph rules may also be referred as rules, graph (transformation/rewriting) rules or graph productions.

A rule $q : L \overset{l}{\leftarrowtail} K \overset{r}{\rightarrowtail} R$ describes how to localy modify the target graph. The graph L, also known as the left-hand side (LHS) of the rule, defines a pattern to be found in the target. The graph R, also known as the right-hand side (RHS), defines the local state after rule execution. The interface graph K, which is included in both the LHS (by l) and RHS (by r), defines the elements that are preserved (read) by the rule application. The elements in $L \setminus K$ (i.e. in L but not in $l(K)$) are said to be deleted, whilst elements in $R \setminus K$ (in R but not in $r(K)$) are said to be created.

Example 13. *Figure 2.4 shows a graph rule which deletes the edge* a*, creates the edge* b *and preserves the three nodes* x*,* y *and* z *(both* l *and* r *are inclusions). Since edges are used to specify the location of messages, this production represents the act of sending a message from a computer to a server.*

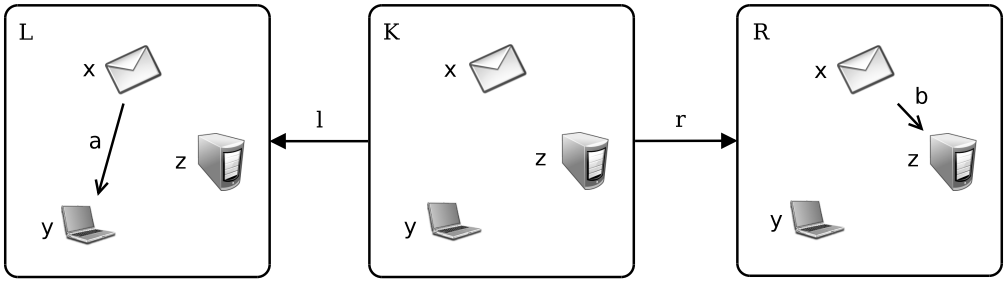


Figure 2.4: Graph transformation rule.

The application of a graph production $p : L \leftarrow K \rightarrow R$ over a target graph G depends on finding a match, i.e. an association of elements in L to elements in the target graph G. A match is defined by means of a graph morphism $m : L \rightarrow G$. Once a match is found, the rewriting of the graph is defined by means of a double pushout diagram in T-**Graph**. Before we introduce the formal definition for rule application, we will introduce the concept of pushout in **Set**, the category of sets and total functions. This is done to illustrate the construction while avoiding the complete (and rather dense) categorial definition. A pushout of two functions $f : C \rightarrow A$ and $g : C \rightarrow B$ is usually referred as an amalgamated sum. The intuition is that first we calculate the disjoint union $A + B$ of sets A and B. After this, elements of $A + B$ are combined into a single element (amalgamation) if and only if they have a common origin in C. Definition 14 presents the calculation of a pushout in **Set**, and Figure 2.5 depicts an example of pushout construction for two particular functions f and g.

Definition 14 (Pushout in **Set**). *Consider the span* $A \overset{f}{\leftarrow} C \overset{g}{\rightarrow} B$*, and the following constructions:*

- *the disjoint union* $A + B = \{a_0 \mid a \in A\} \cup \{b_1 \mid b \in B\}$*. The tags* 0 *and* 1 *to mark for each element its original set.*
- *the least equivalence relation* $\equiv$ *on* $A + B$ *such that for all* $c \in C$*,* $f(c)_0 \equiv g(c)_1$*.*
- *the quotient set* $(A + B)/\equiv \; = \{[x]_\equiv \mid x \in A + B\}$

Then, the co-span $A \overset{g^*}{\rightarrow} (A + B)/\equiv \overset{f^*}{\leftarrow} B$*, where* $g^*(a) = [a_0]_\equiv$ *and* $f^*(b) = [b_1]_\equiv$ *is a pushout of* $A \overset{f}{\leftarrow} C \overset{g}{\rightarrow} B$*.*

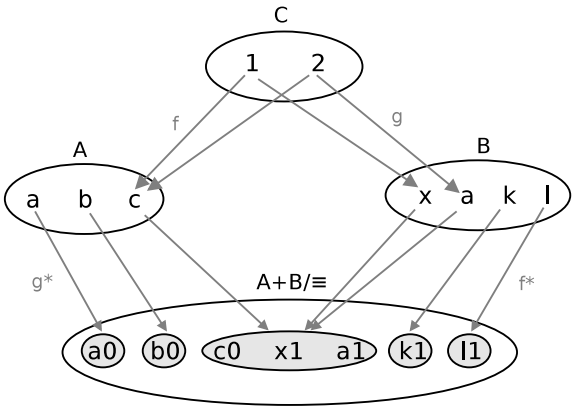


Figure 2.5: Pushout in **Set**.

An important remark is that, similarly to other categorial constructions, pushouts are unique only up-to-isomorphism. This means that we may obtain other pushouts for the same diagram by taking distinct sets with the same cardinality as $(A+B)/\equiv$. For example, another pushout for the diagram shown in Figure 2.5 is the set $\{1, 2, 3, 4, 5\}$ together with functions $\beta = \{a \mapsto 1, b \mapsto 2, c \mapsto 3\}$ and $\alpha = \{x \mapsto 3, a \mapsto 3, k \mapsto 4, l \mapsto 5\}$.

The same intuition of amalgamated sum holds for pushouts in the context of graphs and typed graphs, where they have the role of a gluing procedure to compose graphs identified by common nodes and edges. The definition of one-step rewriting, or direct derivation, is based on a diagram composed of two simultaneous pushouts in the category of typed graphs.

Definition 15 (Direct derivation). *Given a graph rule* $q : L \overset{l}{\leftarrowtail} K \overset{r}{\rightarrowtail} R$ *a graph* G*, and a match* $m : L \rightarrow G$*, a* direct derivation *(or direct graph rewriting) from* G *to* H *using* q *and* m *exists if and only if the diagram below can be constructed in* T-**Graph**, *where both squares (1) and (2) are pushouts.*

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

The direct derivation is referred using the notation $\delta : G \overset{q,m}{\Longrightarrow} H$ *or* $\delta : G \overset{q}{\Longrightarrow} H$ *if we do not make explicit* m*.*

The intuition why this particular kind of diagram models the effect of deleting, creating and preserving elements is as follows. The square (1) is a pushout iff $l^*(D)$ is isomorphic to $G \setminus m(L \setminus K)$, i.e., G without the image (via m) of the elements deleted by the rule. The square (2) is a pushout iff H is the disjoint union of D and R, amalgamated via K. This means that H has all elements of D, plus the elements of R that are outside of $r(K)$, i.e., the elements created by the rule.

Example 16. *Figure 2.6 shows the application of production q (from Figure 2.4) over a graph G using match m. The image of match m is represented by a dotted oval region within G. Notice that there are other two possible matches for this same rule, each covering a distinct message node.*

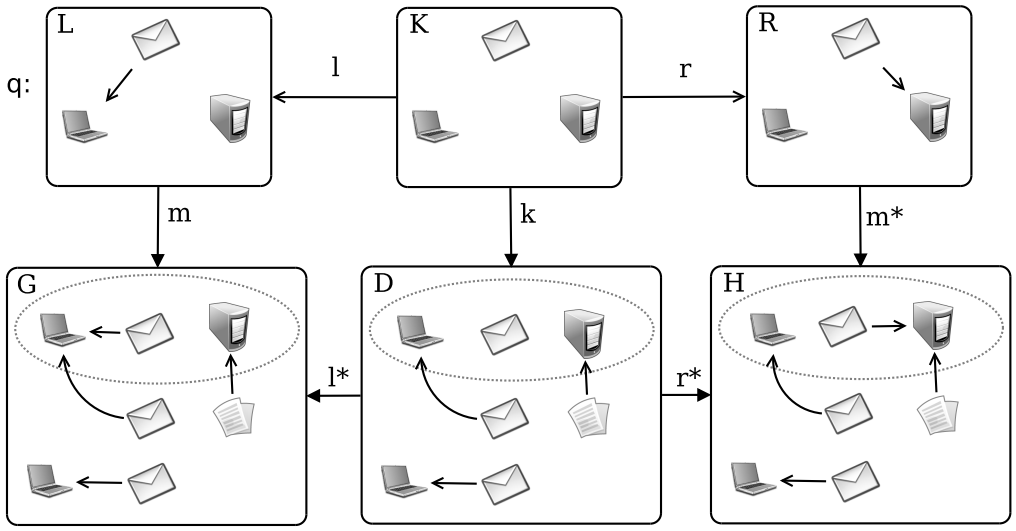


Figure 2.6: Direct derivation representing message passing.

In the DPO approach, the existence of a match for a rule does not assure the construction of the DPO diagram. The first pushout depends on the existence of a pushout complement for $K \overset{l}{\rightarrow} L \overset{m}{\rightarrow} G$, i.e. a complementary diagram $K \overset{k}{\rightarrow} D \overset{l^*}{\rightarrow} G$ making the first square a pushout. This complement only exists if two conditions, named application conditions, are satisfied:

1. *identification condition*: the match $m : L \rightarrow G$ does not map both a deleted item and a preserved item in L to the same item in G.

2. *dangling condition*: the match m does not map a deleted node d to a node $m(d) \in G$ if $m(d)$ is connected to an edge outside the image of m.

If any of the two conditions is not satisfied by a match, the rewriting cannot occur. This is a very particular characteristic of the DPO approach. To constrast, in the single-pushout (SPO) approach (LÖWE, 1993), the existence of a match suffices to obtain a direct derivation. On the other hand, application conditions ensure some interesting characteristics to the rewriting as, for instance, reversibility. A discussion about the differences between the two approaches is out of our scope, and we refer to (ROZENBERG, 1997) for more on this subject.

Now that rules and rule rewriting have been introduced, we can talk about systems built from such concepts. The intuition behind graph transformation systems is to use sets of graph productions to describe the behaviour of a visual model. Formally:

Definition 17 (Graph Transformation System)**.** *A* graph transformation system *(GTS) is a tuple* $\langle T, P, \pi \rangle$*, where T is a type graph, P is a set of rule names and* $\pi : P \to T$*-***Rules** *maps every rule name to is correspondent production.*

Notice that GTSs do not model a particular system, but actualy a set of possible rules that guide its rewriting. If specific initial conditions need to be considered, we have a graph grammar.

Definition 18 (Graph Grammar)**.** *A* graph grammar *(GG) is a tuple* $\mathcal{G} = \langle T, G_0, P, \pi \rangle$*, where* $\langle T, P, \pi \rangle$ *is a graph transformation system and* G_0 *is a T-typed graph named the* initial graph*.*

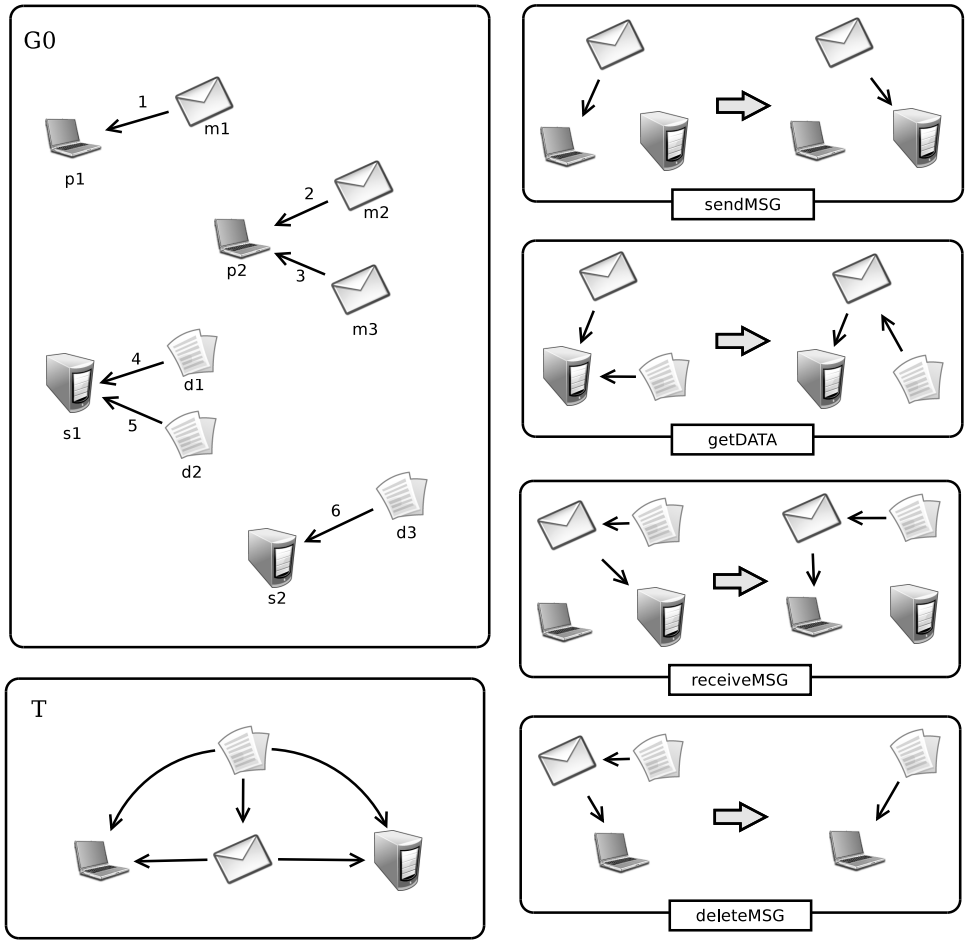


Figure 2.7: Graph grammar for clients and servers

Example 19. *Figure 2.7 shows a graph grammar thats models a client-server scenario. There are four kinds of transitions in this system: clients sending a message to servers (*sendMSG*), obtaining data elements from the server (*getDATA*), servers returning the messages to the clients (*receiveMSG*) and clients obtaining data from returned messages (*deleteMSG*). The interface of all rules is implicitly considered to be the intersection of the LHS and the RHS. The starting situation of the system is represented by the initial graph* G_0*.*

A derivation of a graph grammar is defined as a sequence of subsequent direct derivations using rules in the set *P*.

Definition 20 (Derivation)**.** *Given graph grammar* $\mathcal{G} = \langle T, G_0, P, \pi \rangle$*, a* derivation ρ *is a (possibly infinite) sequence of direct derivations* $\delta_i : G_i \overset{p_i,m_i}{\Longrightarrow} H_i$*, where* $G_{i+1} = H_i$ *and* $i \geq 0$*. If a derivation* $\rho : G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} \cdots \overset{p_n,m_n}{\Longrightarrow} G_n$ *is finite, we call* G_n *the final graph. We denote the class of all derivations of* $\mathcal{G}$ *by* $Der(\mathcal{G})$*.*

The sequential behaviour of a graph grammar , i.e. its behaviour assuming that only one rule can be applied at each step, is given by all derivations with rules in P that start at graph G_0. Operationally, the execution of a graph grammar $\mathcal{G} = \langle T, G_0, P, \pi \rangle$ may be described by the following steps:

1. set the initial graph as the current graph.
2. find in the current graph all possible matches satisfying the application conditions for all rules in P.
3. if there is no suitable match then STOP. Otherwise, non-deterministically choose a rule and a match to be applied.
4. delete from G all matched elements that occur in the LHS but not in the RHS. This will generate a graph D.
5. create in D all matched elements that occur in the RHS but not in the LHS. This will generate a graph H.
6. set H as the current graph. Return to step 2.

Example 21 (Derivation)**.** *Figure 2.8 shows a three-step derivation of the example grammar.*

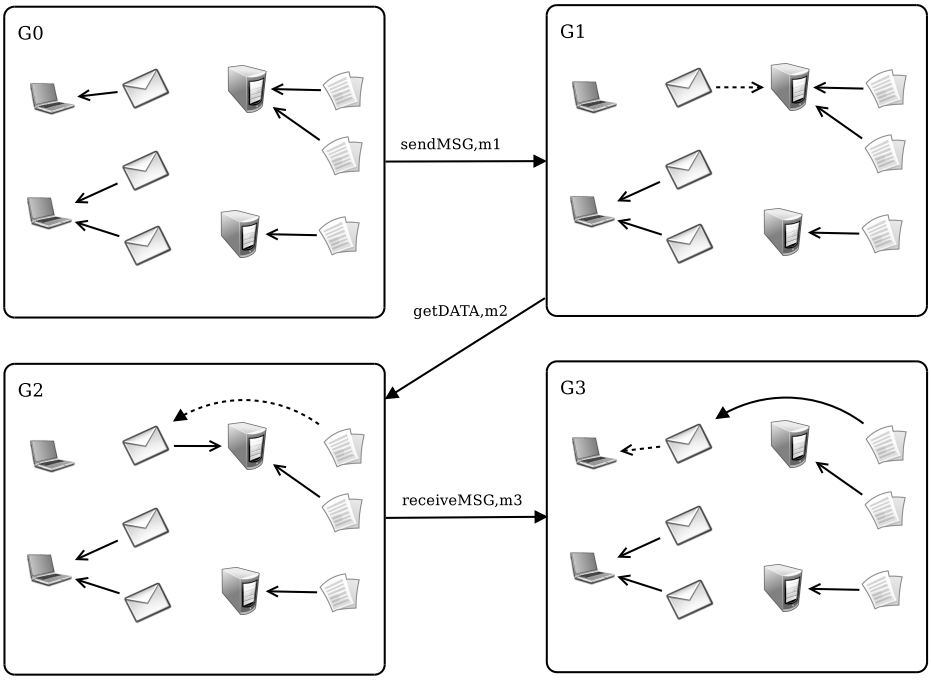


Figure 2.8: Example of derivation.

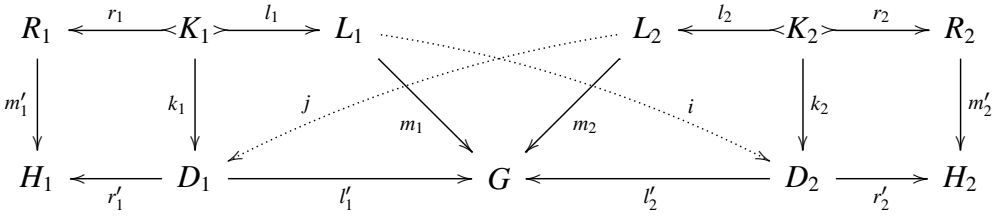
Graph grammars provide a natural and visual way to represent distributed and nondeterministic systems. Distribution is naturally represented by the graph topology, since the semantics of graph grammars is based on production applications. If there are matches for more than one production in one state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

Concerning efficiency, the act of finding a match for the LHS of a rule is critical, specially if the graph being rewritten is very large. This is usually referred as the *subgraph homomorphism problem*, and it is known for being intractable in the general case (RUDOLF, 2000). In practical situations, however, the search space of the algorithm for finding graph matches can be reduced by exploring additional information about the graphs, such as the typing mechanism, or results from previous matches (BERGMANN et al., 2008). Some implementations of the graph rewriting mechanism such as GrGen.NET (GEIS et al., 2006) are highly optimized in this sense, obtaining considerable performance in several graph transformation benchmarks.

2.3 Conflicts, dependencies and parallelism

Now we focus on the notions of conflict and dependency in graph rewriting. Moreover, we review a important result from the graph rewriting theory, regarding confluence and parallelism. We start by characterizing when two derivations are in conflict. Let $\delta_1 : G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $\delta_2 : G \overset{p_2,m_2}{\Longrightarrow} H_2$ be two direct derivations from the same graph G. We say that δ_2 disables δ_1 whenever δ_2 deletes some graph element that is required by the match of δ_1, i.e. if $m_2(L_2 \setminus K_2) \cap m_1(L_1) \neq \varnothing$. If δ_1 disables δ_2 or δ_2 disables δ_1, we say that they are *conflicting* or *parallel dependent*. Notice that if δ_2 disables δ_1 but not the opposite, i.e. δ_1 does not disable δ_2, then the conflict may have an asymmetric nature. If two derivations $\delta_1 : G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $\delta_2 : G \overset{p_2,m_2}{\Longrightarrow} H_2$ are not conflicting, then they can (potentially) be applied over the graph simultaneously and we say that they are parallel independent or conflict-free. Equivalently, we can characterized parallel independence for rewritings in T-**Graph** by testing the existence of some morphisms, as shown in the next definition.

Definition 22 (Parallel independence)**.** *Two derivations $\delta_1 : G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $\delta_2 : G \overset{p_2,m_2}{\Longrightarrow} H_2$ are parallel independent iff there are morphisms $i : L_1 \to D_2$ and $j : L_2 \to D_1$ such that $i; l'_2 = m_1$ and $j; l'_1 = m_2$, as shown in the diagram below.*



Example 23. *Figure 2.9 depicts a conflict between two rewritings of the rule* sendMSG*. Each rewriting disables the other because both attempt to delete the same edge. In this case, neither morphisms $i : L_1 \to D_2$ and $j : L_2 \to D_1$ exist (cf. Definition 22).*

Now we focus on the concept of sequential dependence. Given a derivation $\rho = \delta_1, \ldots, \delta_n$, sequential dependence occurs between two direct derivations δ_i and δ_j whenever δ_j is forced to occur after δ_i. Element-wise, the following situations entail dependencies:

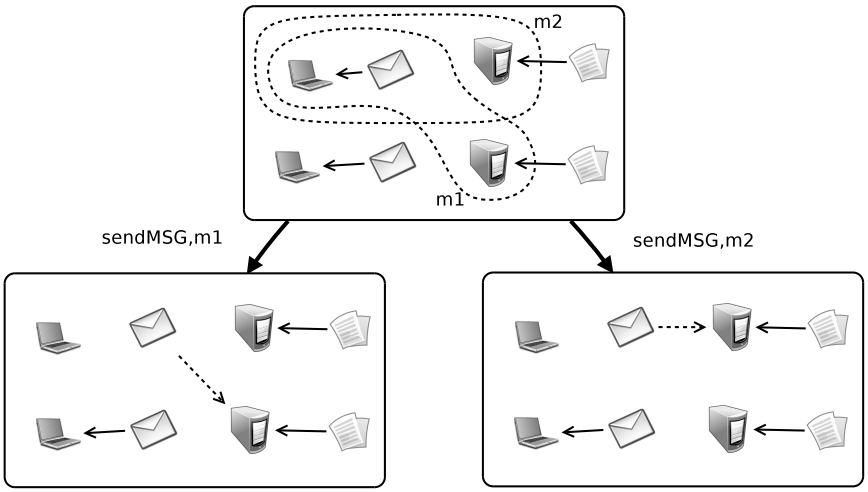


Figure 2.9: Parallel dependent derivations.

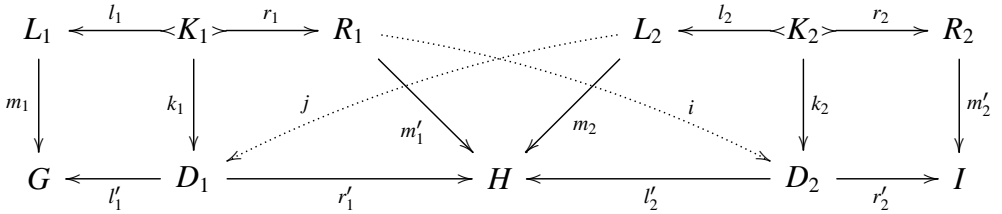
1. *create-delete:* an element is created by the first rewriting and deleted by the second one;

2. *create-read:* the latter rewriting preserves an element created by the former;

3. *read-delete:* the first rule preserves and element that is deleted by the second one.

Dependencies play a important role in graph rewriting since they block parallel execution, and also disallow the exchanging of rewriting steps, i.e. they fix the relative order of execution between two rewritings within a larger derivation. We say that two rewritings are sequentially independent iff they are not sequential dependent. Similarly to parallel independence, sequential independence can also be characterized by means of the existence of morphisms.

Definition 24 (sequential independence)**.** *Consider two direct derivations* $G \overset{p_1,m_1}{\Longrightarrow} H$ *and* $H \overset{p_2,m_2}{\Longrightarrow} I$ *where* $p_1 = (l_1, r_1)$ *and* $p_2 = (l_2, r_2)$*. The derivations are said to be* sequential independent *iff there are morphisms* $i : R_1 \to D_2$ *and* $j : L_2 \to D_1$ *such that* $m'_1 = j; r'_1$ *and* $m_2 = i; l'_2$*, as shown below:*



Example 25. *Figure 2.10 depicts a rewriting of* getDATA *and a rewriting of* receiveMSG *such that they are sequentially dependent. In this case, the dependency arises because the arrow connecting the message to the server is preserved by the first derivation and deleted by the second one. Considering the existence of i and j (cf. Definition 24), there is actually j, since the mentioned arrow has a pre-image along* r'_1*, however there is not i because there is not a pre-image along* l'_2 *of the same arrow.*

The formal characterization of sequential independence follows from the useful observation that graph rules and rewritings in the DPO approach are invertible. In other words,

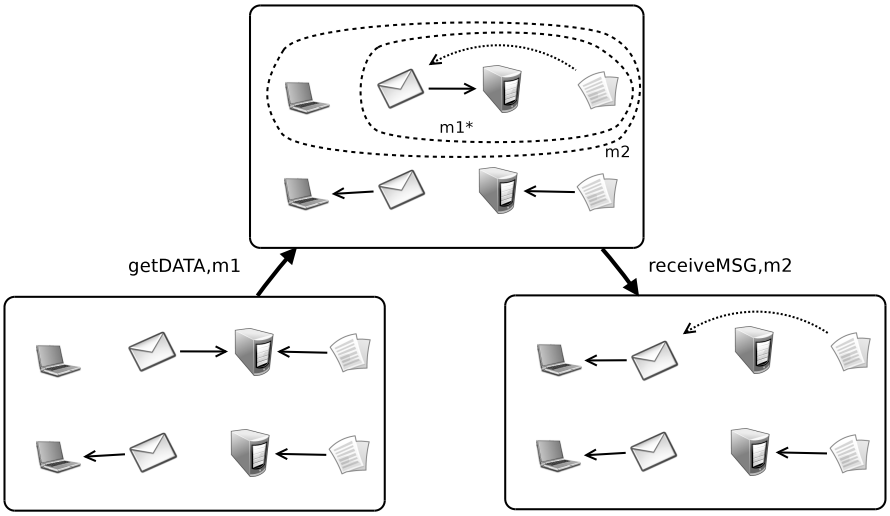


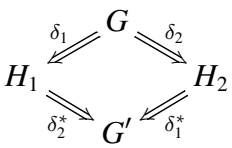
Figure 2.10: Sequentially dependent derivations.

from rule $p : L \leftarrow K \rightarrow R$ we can obtain the reverse rule $p^{-1} : R \leftarrow K \rightarrow L$ which has the opposite effect of p, i.e. it obtains the LHS from the RHS. The same idea is used for inverting direct derivations, since double-pushout diagrams can be read from right to left. We have that $\delta_1 = G \overset{p_1,m_1}{\Longrightarrow} H$ and $\delta_2 = H \overset{p_2,m_2}{\Longrightarrow} J$ are sequentially independent iff δ_1^{-1} and δ_2 are parallel independent. There are interesting consequences if two rewritings are independent, which is assured for conflict-free rewriting through the theorem known as *local Church-Rosser*. The name *Church-Rosser* is a reference to the famous theorem that proves the confluence of beta-reduction in the untyped lambda-calculus, proved by Alonzo Church and J. Barkley Rosser. By *local* it is understood it refers to a context where the scope of rewritings is restricted, such as graph transformation. Formally, it states that confluence is obtained from parallel independence:

Theorem 26 (Local Church-Rosser)**.** *The following two statements are equivalent descriptions of local confluence in the DPO approach:*

1. *Let* $\delta_1 : G \overset{p_1,m_1}{\Longrightarrow} H_1$ *and* $\delta_2 : G \overset{p_2,m_2}{\Longrightarrow} H_2$ *be parallel independent. Hence, there are two derivations* $\delta_2^* : H_1 \overset{p_2,m_2^*}{\Longrightarrow} J$ *and* $\delta_1^* : H_2 \overset{p_1,m_1^*}{\Longrightarrow} J$ *such that* $(\delta_1; \delta_2^*)$ *and* $(\delta_2; \delta_1^*)$ *are sequentially independent.*

2. *Let* $\delta_1 : G \overset{p_1,m_1}{\Longrightarrow} H_1$ *and* $\delta_2^* : H_1 \overset{p_2,m_2^*}{\Longrightarrow} G'$ *be sequentially independent derivations. Then, there are sequentially independent derivations* $\delta_2 : G \overset{p_2,m_2}{\Longrightarrow} H_2$ *and* $\delta_1^* : H_2 \overset{p_1,m_1^*}{\Longrightarrow} G'$ *such that the pair* (δ_1, δ_2) *is parallel independent.*



The notion of parallel independence leads to the precise definition of parallel rewritings. Given T-typed graph rules p_1 and p_2, the *parallel rule* $p_1 + p_2$ may be obtained component-wise disjoint union of p_1 and p_2. Parallel rules may be obtained from

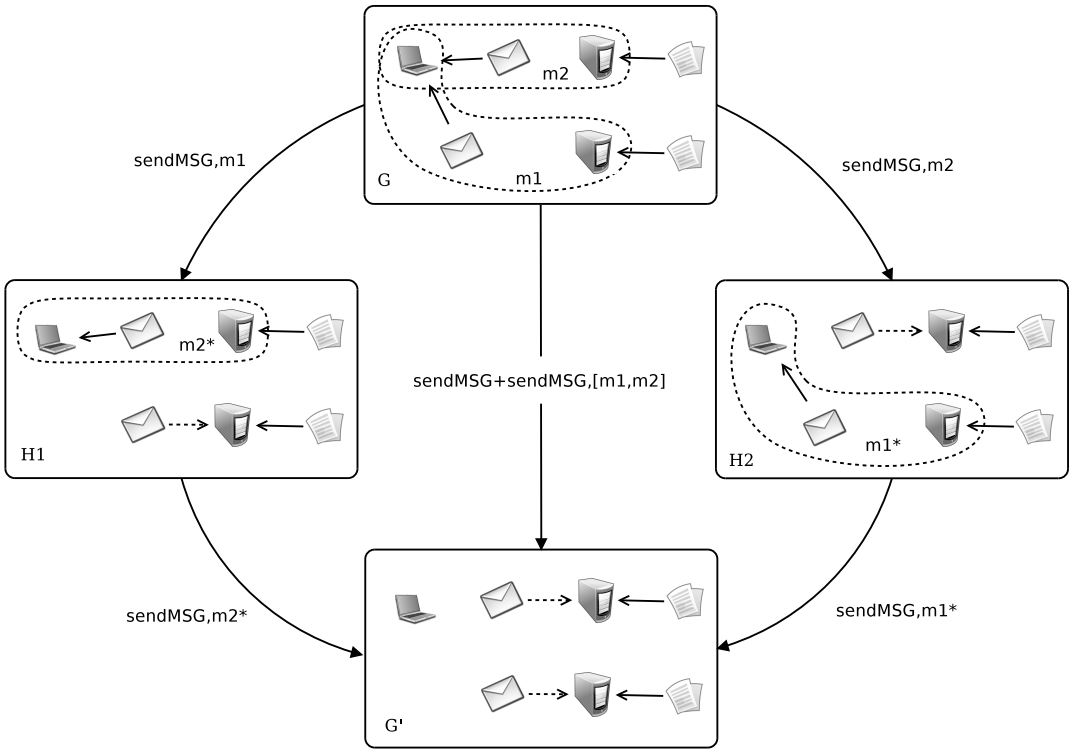


Figure 2.11: Parallel independence and parallel rule execution in graph rewriting.

the disjoint union of simple rules and/or other parallel rules. Given parallel independent derivations $\delta_1 : G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $\delta_2 : G \overset{p_2,m_2}{\Longrightarrow} H_2$, the *parallel direct derivation* $\delta_1 + \delta_2 : G \overset{p_1+p_2,[m_1,m_2]}{\Longrightarrow} G'$ is defined as the application of the parallel rule p_1+p_2 with match $[m_1, m_2] : L_1 + L_2 \to G$, obtained from the superposition of m_1 and m_2 (technically, this morphism arises as a unique arrow from the coprodut object L_1+L_2). Notice that a parallel rule $p_1 + p_2$ is not an atomic entity: it represents the parallel application of two rules and have both p_1 and p_2 as components. Generally, parallel rules are not part of the ruleset P, unless P is infinite and closed under disjoint union of rules. Likewise, parallel direct derivations are not a subset of the system direct derivations. The parallelism theorem allows to relate parallel derivations with sequentially independent direct derivations, as follows.

Theorem 27 (Parallelism)**.** *Let p_1 and p_2 be two (possibly parallel) graph rules. Then, there is a parallel derivation $\delta_1 + \delta_2 : G \overset{p_1+p_2,[m_1,m_2]}{\Longrightarrow} G'$ iff there is a sequential independent pair of rewritings $\delta_1 : G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $\delta_2^* : H_1 \overset{p_2,m_2^*}{\Longrightarrow} G'$.*

Proofs for both local Church Rosser and the parallelism theorem may be found at (ROZENBERG, 1997), Chapter 3, Section 3.4.

Example 28. *Figure 2.11 describes a situation where parallel independence between rewritings* sendMSG,m1 *and* sendMSG,m2 *entails of sequentially independent two-step derivations and a parallel rule rewriting. By local Church Rosser, there are rewritings* sendMSG,m1* *and* sendMSG,m2* *converging to G'. By the parallelism theorem, there is the parallel rewriting* sendMSG+sendMSG,[m1,m2].

We have presented the definitions for dependency in more detail because they will be required in the understanding of the critical pair analysis technique. Now we focus on the

observational aspect of graph grammar specifications, i.e. how to formally characterize their execution.

2.4 Semantic models for graph grammars

In this section we briefly comment on semantic models for graph grammars, distinguished by the way they represent nondeterminism and parallelism. Semantic model are usually categories whose objects represent graphs and morphisms, the respective rewritings transforming one graph into another. Precise definitions may be found in (ROZENBERG, 1997).

Sequential, concrete semantics: This is arguably the simplest semantic model for graph grammars. Given a graph grammar $\mathcal{G}$ we denote by $Der(\mathcal{G})$ the category whose objects are T-typed graphs and morphisms $f : G \to H$ are arbitrarily sized derivations $G \overset{p_1,m_1}{\Longrightarrow} G' \overset{p_2,m_2}{\Longrightarrow} \ldots \overset{p_n,m_n}{\Longrightarrow} H$. The composition operation is derivation concatenation, and the identity of G is the "no-operation" DPO diagram where all morphisms are the identity of G. To consider the behaviour starting from a particular grahp G_0, we simply take the slice category $G_0 \downarrow Der(\mathcal{G})$. Besides its inherent simplicity, there are some issues with such model:

1. *too much redundancy:* for each typed graph, there are infinite isomorphic graphs, differing only in the elements of the instance graphs. Moreover, for each direct derivation $\delta : G \to H$, there are infinitely many isomorphic rewritings, differing only by the concrete elements in each of the graphs. Hence, even for small graphs G and H we find infinitely many morphisms between them.

2. *concurrency is implicit:* the model focus mainly on sequential execution. Concurrency situations need to be extracted for sequentially independent components of a bigger derivation. Two distinct serializations of the same collection of rewritings events are considered distinct because of the order direct derivations occur.

Abstract semantics: This models handles the issue of redundancy in representation: objects of the category are equivalence classes of isomorphic graphs, and morphisms are classes of isomorphic derivations. An important consequence of using this model is the possibility to obtain finite categories as semantic models for terminating graph grammars. One example is the free category generated from the transition system presented in Figure 1.2.

Truly concurrent semantics: In this model, derivations that only differ in permutations of sequentially independent rewritings are said to be *shift-equivalent*. For instance, if we have a two-step derivation $G \overset{p_1,m_1}{\Longrightarrow} H \overset{p_2,m_2}{\Longrightarrow} J$ where the direct derivations are sequentially independent, from local Church Rosser we also have a derivation $G \overset{p_2,m_2^*}{\Longrightarrow} H' \overset{p_1,m_1^*}{\Longrightarrow} J$ for some H'. In this semantic model, both are considered *shift-equivalent* derivations, and hence they are identified as the same morphism.

Abstract, truly concurrent semantics: Combines abstraction and shift-equivalence. This model is considered equivalent to another notion of concurrent semantics of graph gram-

mars named *graph processes* (CORRADINI; MONTANARI; ROSSI, 1996), although both models are defined in considerably different ways.

Unfolding semantics: Roughly speaking, the unfolding semantics is a structure which represents all alternative rewritings of a given graph grammars, marking explicitly the dependences and conflicts between individual graph rewriting steps. It is inspired on event structures and was originally defined in (RIBEIRO, 1996) for the single-pushout approach. Recently, it has been proposed for adhesive HLR sytems in (BALDAN et al., 2009).

Besides such models, it is also usual to extract a labeled transition system (LTS) based on the execution of a graph grammar, and use it as a semantic model for analysis tools based on model checking. For instance, that was the method in (FERREIRA, 2005), where the SPIN model checker was used to study object-oriented graph grammar specifications. Notice, however, that such LTS extraction process needs to address the same kind of issues considered by the referred semantic models, such as representation nondeterminism and possibly infinite behaviour. More details regarding verification of graph grammars may be found in (RENSINK; SCHMIDT; VARRÓ, 2004) and (BALDAN; KÖNIG; RENSINK, 2005).

2.5 Rules with negative application conditions

The presented description of graph grammars using typed graphs and DPO rules is sufficiently powerful to model any system, supported by the fact that graph grammars are (in most of the variations) Turing-computable models, i.e., we may represent any computable function. However, being powerful does not necessarily mean being convenient, and the appeal of being a visual language would fade away if the models became excessively cluttered. Ideally, expressiveness should not be obtained in exchange for clarity. There are several variations of the basic typed graph grammars, making the modelling process more convenient and the models clearer. In this section we will describe the adoption of negative application conditions for rules.

In the DPO approach, rules are applicable whenever there are valid matches for their left-hand sides, therefore the existence of a match can be seen as a necessary condition for the rule application. If we want to extend the requirements for a rule application without changing its effect, we may simply require it to read more context by adding new preserved elements representing the additional requirements. On the other hand, we do not have a direct mechanism to *avoid* rule application according to the presence of an undesired context. As an example of the described situation, we could try to represent in our example grammar the fact that each server has a limit: each server can process only two messages at a given time. This means that we can only send a new message to a server if it has at most one message connected. To implement such requirement, we could require the introduction of new elements, for instance, self-edges in the server as allocation counters to be read and updated by the rules. However, although straightforward, this solution creates extra elements only required for control. A reasonably better solution would be to avoid rule application by stating directly that sendMSG cannot be executed if there are more than one message in the server. A negative application condition (NAC) for rule $p : L \leftarrow K \rightarrow R$ is a morphism $n : L \rightarrow N$, i.e. a morphism going from the LHS of the rule to another graph N. A *graph rule with NACs* is a pair (p, N) where p is a rule and N is a set of negative application conditions for p.

Example 29 (Rule with NAC)**.** *Figure 2.12 depicts the* sendMSG2 *rule, which has a NAC determining that it cannot be executed if the receiving server already is already processing two messages.*

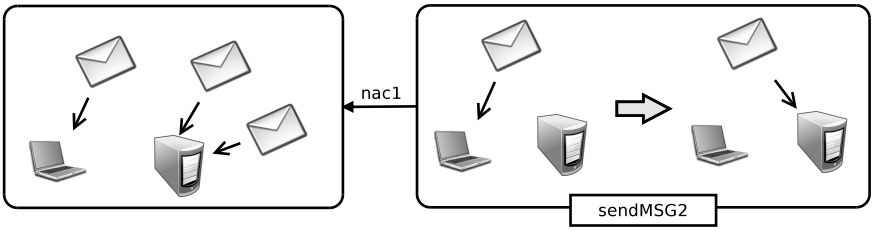
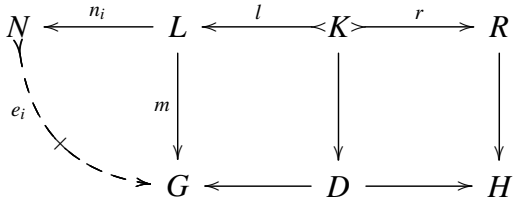


Figure 2.12: Rule with negative application condition.

The allowed direct derivations for (p, N) are all rewritings $G \overset{p,m}{\Longrightarrow} H$ such that *there is no* monomorphism $e_i : N \to G$ such that $e_i \circ n_i = m$, for all $n_i \in N$, i.e. none of the application conditions has a monic factorization of the match m. The situation is diagrammatically shown below.



The restriction that all factorizations e_i have to be monomorphisms is important: if it were not present, the NAC of the example would forbid the execution of the rule if the server had one message in it, because the factorization morphism would enable to map the two messages in N to the same message in graph G. In general, negative application conditions allow for convenient definition of rules that can only be triggered until a given condition is achieved. For instance, in the place-transition example of Chapter 1, they were present in the model to assure that a new counter place is created only if it already does not exist. This situation is an example of the usage of NACs to ensure termination for rewritings based on non-deleting rules.

One important characteristic of rules with NACs is that they allow a new kind of parallel and sequential dependencies. While a conventional rewriting $\rho = G \overset{r,m}{\Longrightarrow} H$ could forbid the execution of $\rho' = G \overset{r',m'}{\Longrightarrow} H'$ only by removing elements from the image of m', now we can forbid ρ' by *creating* in H' some element that may allow the existence of a factorization of *some* NAC of r'. In later sections, we will present how the definitions for conflict and dependency must be adapted for the presence of matches. Good references for this topic are (LAMBERS; EHRIG; OREJAS, 2006; LAMBERS et al., 2008; LAMBERS; EHRIG; OREJAS, 2008).

2.6 Typed attributed graph grammars

In the typed graph grammar context, encodings are required whenever we need to specify natural numbers, arithmetic even booleans conditions. Although it is possible to encode such entities by means of nodes and edges, this is not very convenient, and thus a mechanism that allows data to be used directly in graphs and graph rewriting is

very desirable. This was precisely the motivation for the introduction of typed attributed graphs (TAG), which allow the association of values to their nodes and edges. For this, there are special *attributing edges*, whose source refers to graph element (nodes or edges) and the target, to data values. Those data values are defined as elements of an algebra which signature is encoded into the type graph of the specification. In the following, we will omit the formal definitions, focusing mainly on the intuition.

Example 30. *Figure 2.13 shows an example of typed attributed graph to represent messages. In the type graph T, the node MSG has tree attributing edges: from and to of type String, and dataID of type Integer. Each node instance of MSG may have associated attributing edges providing a particular value for each attribute type.*

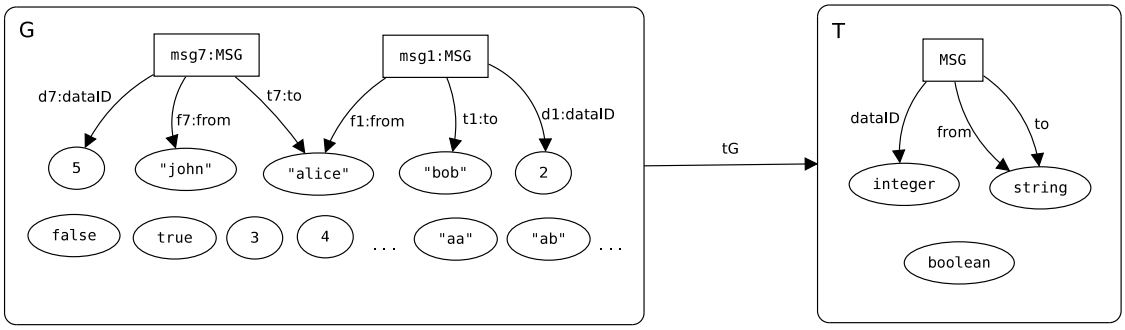


Figure 2.13: Typed attributed graph.

To illustrate the usefulness of attributes we remark that, although suitable to illustrate the basics of graph rewriting, our example graph grammar is too abstract for describing realistically clients and servers. In the current setting, messages are sent to servers, data is collected and returned to clients in a totally non-deterministic way: no preferred server, data or client to return to. In more detailed models, data within messages such as sender and receiver address should be considered by the rewriting rules. Typed attributed rules use variables to represent instantiated values in the LHS, and can update attributes by means of algebraic expressions in the RHS. Moreover, relations between the attributes in the LHS guide the rewriting as well.

Example 31. *Figure 2.14 shows an example of typed attributed graph grammar for the same scenario of clients and servers. The conventional notation for TAG is used: values for attributes shown inside nodes, as in object diagrams of UML. Notice that all transitions require some kind of matching between the information in the message and the identification of the node, referred by means of variable x.*

2.7 Tools and analysis techniques

There are several tools for working with graph grammars, usually implementing creation, simulation and analysis of models. Here we mention some of the most popular graph grammar tools, and comment on some analysis techniques implemented by them.

AGG (TAENTZER, 2000) allows one to model and simulate typed attributed graph grammar specifications. The tool supports defining rules with negative and positive application

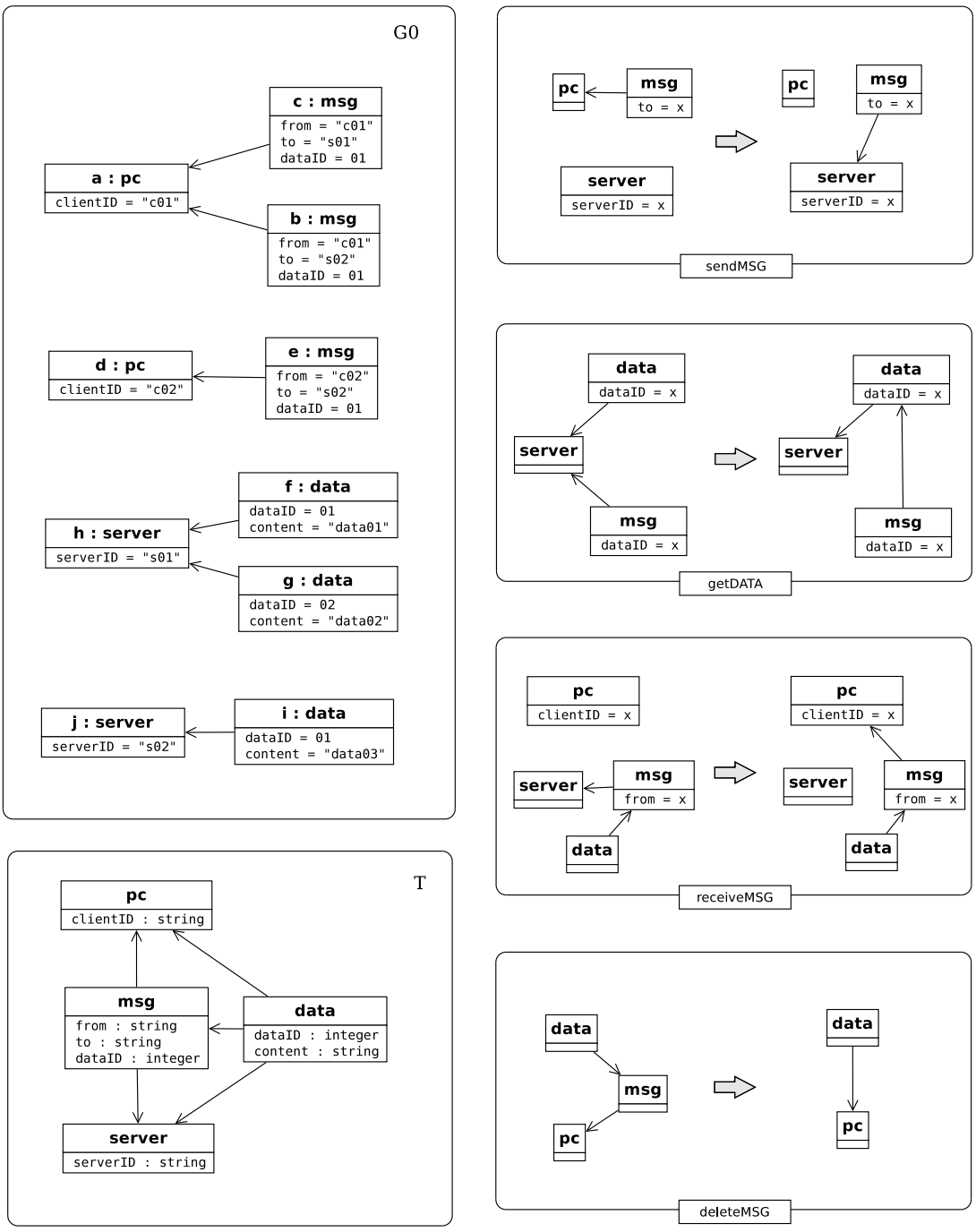


Figure 2.14: Typed attributed graph grammar.

conditions, layered rule execution, and integration with user-defined Java methods. Concerning analysis, it implements critical pair analysis for specifications. However, it does not allow for model checking of the space-state of the model.

Groove (RENSINK, 2004) allows to define and simulate typed graph grammars, with support for attributes and other extensions for the formalism. The interesting aspect of this tool is that it allows the generation and model-checking – by means of Computational Tree Logic (CTL) – of the space-state of the execution of the graph grammar, which is quite useful from the point of view of understanding the rewriting process. It does not implement, however, critical pair analysis.

Augur (KÖNIG; KOZIOURA, 2005) is a verification tool that implements the approximated unfolding analysis technique for the exploration of graph grammar space-state.

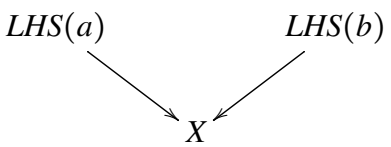
There are also important tools such as FUJABA (KLEIN et al., 1999), PROGRESS (SCHÜRR, 1990) and GREAT (BALASUBRAMANIAN et al., 2006), which employ the graph rewriting mechanism in the context of model and meta-model transformation in software engineering scenarios.

2.8 Critical pair analysis

A critical pair is an overlap between components of two rules which makes their applications conflicting. It is usually referred as a "essential" conflict since it reproduces the interaction between two rules from the point of view of their matches (or co-matches), independently of the remaining of the graph. The precise definition will depend on the usage or not of NACs. A critical pair may refer to parallel dependence if we consider the LHS of both rules, or sequential dependence, if we take the RHS of the first rule with the LHS of the second one.

Critical pair analysis (CPA) is a static analysis technique that is used to detect potential conflicts and dependencies between two graph rewriting rules within the same specification. When we say potential, we mean that the conflicting situation may actually not occur due to particular conditions on the initial graph, but they may occur given a suitable initial condition. On the other hand, all kinds of conflict that actually occur in a system are always foreseen by the method. Operationally, the calculation of critical pairs (for rules without NACs) is done according to the algorithm shown in Figure 2.15.

Initially, we construct all possible pairs of rules. For each pair (a, b), we calculate all possible overlaps, i.e., all *jointly surjective* pairs of morphisms $f : LHS(a) \to X$ and $g : LHS(b) \to X$, as shown below,



representing the possible interactions between the images of matches for both rules. The function *calculateAllOverlaps* can be implemented by calculating all possible partitions of the disjoint union $LHS(a) + LHS(b)$, subjected to constraints on types, edge connectivity, and application conditions for a and b. After this, each overlap is tested for conflicts, and the non-conflicting ones are removed from the list. The algorithm returns the collection

Input : A graph grammar $\mathcal{G} = \langle T, G_0, P, \pi \rangle$

Output: A list of triples (r_1, r_2, ol) where r_1 and r_2 are rule names in P and o is a conflicting overlap of LHS(r_1) and RHS(r_2);

begin

```
// Calculate all pairs of rule names
```
$P \leftarrow \{(r_1, r_2) \mid r_1 \in P,\ \ r_2 \in P\}$;

```
// Initialize accumulator with the empty list
```
result ← [];

```
// Calculate all overlaps
```
for $(a, b) \in P$ **do**

```
    // Generate all possible superpositions between the LHS of
       both rules
```
    overlaps ← calculateAllOverlaps ($LHS(a)$, $LHS(b)$);

```
    // Remove from the list all superpositions that are not
       conflicting
```
    criticalPairs ← filter (isConflicting, overlaps);

```
    // Prefix each superposition with the respective rule names
```
    labeledCriticalPairs ← map ($(o \mapsto (a, b, o))$, criticalPairs);;

```
    // Accumulate critical pairs for rules a and b
```
    result ← concatenate(result, labeledCriticalPairList);

return *result*

Figure 2.15: Algorithm for calculating critical pairs (conflicts).

of critical pairs of all pairs of rules in the specification, labeled according to their respective rules. Notice that the same procedure can be used to enlist all potential sequential dependencies between rules. In this case, the only required modification is need in the calculation of overlaps, whereas we use $RHS(a)$ and $LHS(b)$. Implementations of CPA in tools usually present a table counting the number of critical pairs between rules, and allow to inspect interactively the structure of each one of them.

One of the advantages of CPA is that it provides structural information about what causes the conflicts between rewritings. Because of the non-deterministic, data-driven execution model of graph grammars, it is sometimes hard to foresee some interactions between rules which may be overlooked during the modelling phase of a system.

Example 32. *The resulting tables for critical pair analysis (in AGG) for the simple client-server example of Figure 2.7 (without attributes) are shown in Figure 2.16. Taking a particular example of critical pair, the table of conflicts shows 2 potential conflicts between* receiveMSG *and* getDATA*. They arise from the situation where the first rule deletes the edge from message to server, while the other one may be trying to "load" the message with another data, and thus preserving the same edge. Because AGG tests asymmetric conflicts (counting a conflict in* (a,b) *only when a disables b and not when b disables a), the conflict table is not perfectly symmetric.*

Conflicts	sendMSG	getDATA	receiveMSG	deleteMSG
sendMSG	2	0	0	2
getData	0	3	0	0
receiveMSG	0	2	6	1
deleteMSG	3	0	1	3

Dependencies	sendMSG	getDATA	receiveMSG	deleteMSG
sendMSG	0	1	2	0
getData	0	0	3	0
receiveMSG	2	0	0	2
deleteMSG	3	0	1	3

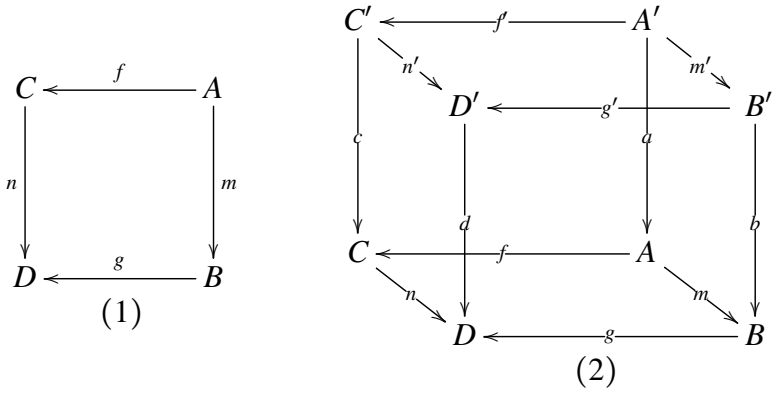
Figure 2.16: Results from critical pair analysis.

2.9 Adhesive HLR categories and systems

The framework of Adhesive High-Level Replacement (HLR) Systems (EHRIG et al., 2004) is a generalization of the DPO rewriting approach. It allows to reuse several results from typed graph rewriting to other contexts, such as typed attributed graphs, place-transition nets and algebraic specifications. Roughly speaking, it states minimum requirements for a given category which are sufficient for it to conform with the theory of DPO rewriting. A category is *adhesive HLR* if it has pushouts and pullbacks which are compatible with each other along a special class of monomorphisms $\mathscr{M}$, being those morphisms in $\mathscr{M}$ the ones used to build up rules in the specification. More precisely, the compatibility condition refers to pushouts along $\mathscr{M}$ being Van Kampen squares, as shown in the next two definitions.

Definition 33 (Van Kampen square)**.** *A pushout (1) is a Van Kampen (VK) square if, for*

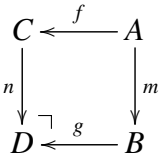
any commutative cube (2) with (1) in the bottom and where the back faces are pullbacks, the following statement holds: the top face is a pushout iff the front faces are pullbacks:



$$isVK(f,m,n,g) = isPO(f,m,n,g) \ \wedge \ isPB(a,m',b,m) \ \wedge \ isPB(a,f',c,f) \implies$$

$$(\ isPO(f',m',n',g') \iff isPB(n,d,c,n') \wedge isPB(g,d,b,g')\ )$$

Definition 34. *A pushout diagram*



is said to be along a given class of morphisms $\mathcal{M}$ iff either $f \in \mathcal{M}$ or $g \in \mathcal{M}$.

Definition 35 (adhesive HLR category)**.** *A pair $(\mathscr{C}, \mathcal{M})$ where $\mathscr{C}$ is a category and $\mathcal{M}$ is a class of monomorphisms in $\mathscr{C}$ is said to be a* adhesive HLR category *iff*

- *$\mathcal{M}$ is a class of monomorphisms closed under isomorphisms, composition $(f : A \to B \in \mathcal{M}, g : B \to C \in \mathcal{M} \Rightarrow g \circ f \in \mathcal{M})$ and decomposition $(g \circ f \in \mathcal{M}, g \in \mathcal{M} \Rightarrow f \in \mathcal{M})$*
- *$\mathscr{C}$ has pushouts and pullbacks along $\mathcal{M}$-morphisms and $\mathcal{M}$-morphisms are closed under pushouts and pullbacks*
- *pushouts in C along $\mathcal{M}$-morphisms are VK squares*

Although rather cryptic, the notion that all pushouts along $\mathcal{M}$-morphisms are Van Kampen squares is sufficient to assure that several important pre-requisites for DPO-related properties hold in the category in question. As examples, we mention the following properties:

- pushouts along $\mathcal{M}$-morphisms are pullbacks;
- pushout complements for diagrams in the format $A \leftarrow B \nleftarrow C$ are unique (up to isomorphism).

For more details (and the respective proofs) of those facts, we refer the reader to (EHRIG et al., 2005). Whenever the class $\mathcal{M}$ correspond to all monomorphisms in $\mathscr{C}$, we say that $\mathscr{C}$ is an *adhesive category*. Historically, adhesive categories were defined first in (LACK; SOBOCIŃSKI, 2003), and adhesive HLR categories (EHRIG et al., 2004) are

a generalization of the framework that includes important practical formalisms such as typed attributed graph grammars. The category **Set** is the most straightforward example of adhesive category, and we know that adhesive categories are closed under product, slice, co-slice and functor construction. Hence, we can deduce that **Graph**, *T*-**Graph** are all adhesive by construction. Those properties of construction of adhesive categories will be particularly important for the description of modifications in graph rules, as will become clear in the next chapters. The equivalent concept of a graph grammar for a given adhesive HLR category is an adhesive HLR system.

Definition 36 (Adhesive HLR system)**.** *An adhesive HLR system* $AS = (\mathscr{C}, \mathscr{M}, S, P)$ *consists of an adhesive HLR category* $(\mathscr{C}, \mathscr{M})$*, a start object* S *and a set of productions* P*, where each production* $p = L \xleftarrow{l} K \xrightarrow{r} R \in P$ *is a span in* $\mathscr{C}$ *such that* $l, r \in \mathscr{M}$*.*

We aim to characterize the second-order rewriting layer as an adhesive HLR system itself. In this way we obtain a precise notion of critical pair and independence for second-order rewritings, among other properties. Furthermore, it is particularly helpful to work with general frameworks if we wish to generalize the results obtained from typed graphs towards more complex scenarios such, for instance, typed attributed graphs.

2.10 Summary

In this chapter we have sketched some of the main concepts of the graph transformation area that will be useful in this work. The research area is huge, and large parts of it were purposely omitted by reasons of space and focus. We started with the basic definitions of graph grammar and their derivations, then we commented on conflicts and dependencies, basic properties such as local Church Rosser, semantic models, negative application conditions for rules, attributed graph grammars, tools, analysis techniques, and, finally, the generalization of the rewriting framework referred as *adhesive HLR systems*.

For the reader wishing a more detailed vision of the field we recommend the classic books of the trilogy *Handbook of graph grammars and computing by graph transformation*: Volume 1: Foundations (ROZENBERG, 1997), Volume 2: applications, languages and tools (EHRIG et al., 1999) and Volume 3: concurrency, parallelism, and distribution (EHRIG et al., 1999). Another great book is *Fundamentals of algebraic graph transformation* (EHRIG et al., 2005), which mentions in detail the framework of adhesive HLR systems.

3 HIGHER-ORDER IN LAMBDA-CALCULUS

This chapter elaborates on the concept of higher-order constructions. For this, we revise the lambda-calculus formalism, well-known for being convenient for defining higher-order functions. The definitions are mostly taken from standard references such as (BARENDREGT, 1992). Our focus here is not to dive into the vast theory of lambda calculi, but instead build up an intuition of how it represents higher-order functions. Guided by this intuition, we provide a comparison between lambda-calculus and graph rewriting, and sketch a characterization of higher-order terms for the context of graph transformation.

3.1 Untyped lambda calculus

The untyped lambda calculus, as introduced by Church, was initially proposed as a language to formalize the concepts of function definition and function application. Based on the original untyped description of the calculi, several variations were developed across the years, specially several typed variations. The basic untyped lambda calculus is Turing-computable, i.e. it allows to represent all computable functions between natural numbers, and thus has the same expressiveness as other universal formalisms. The remarkable fact is that it is minimalistic in its constructions, as shown by the its syntax of terms.

Lambda calculus refer to lambda terms and their transformations. Lambda terms, or lambda expressions, are constructed as equivalence classes of so-called pre-terms. Pre-terms are syntactical expressions built from the following grammar

$$M, N ::= x \mid M\ N \mid \lambda x.M$$

Uppercase letters such as M, N, P and Q refer to arbitrary pre-terms. The set of all pre-terms is denoted Λ. Pre-terms may be elements from an arbitrary infinite but countable set X of names, usually referred by lowercase letter such as x and y. Pre-terms in the format $P\ Q$ are named *applications*, and intuitively correspond to the execution of term P as a function receiving term Q as an argument. An application $P\ Q$ will be written $@(P, Q)$ when we want to focus on its representation as an abstract syntax tree. Pre-terms in the format $\lambda x.P$ and named *lambda abstractions* and denote an anonymous functions where the name x is an input parameter and the calculated expression is given by P. Mostly often, x occurs inside P, describing how the input parameter is processed. The usual syntactical conventions to avoid an excess of parenthesis say that (i) applications are associative to the left: $x\ y\ z = (x\ y)\ z \neq x\ (y\ z)$, (ii) the term inside an abstraction extends to the right as much as possible: $\lambda x.xyz = \lambda x.(x\ y\ z) \neq (\lambda x.x)\ y\ z$ (iii) subsequent abstractions may be abbreviated as a single λ followed by a sequence of variables: $\lambda x.(\lambda y.x) = \lambda x\ y.x$. We

present some examples of pre-terms as follows.

$$\begin{array}{lll} x & x\ y & (\lambda x.x\ x) \\ x\ (\lambda y.x) & (\lambda x.y)(\lambda y.x\ y) & (\lambda x.\lambda y.y\ x) \end{array}$$

Each occurrence of a variable x inside a term P may be either bound or free. An occurrence of x is *bound* if it is inside a sub-term $\lambda x.M$. It is *free* otherwise. For instance, in the term $(\lambda x.x\ y)\ y$ is free but x is bound. Variable occurrences always bind to the innermost λ with the same name.

Bound variables have the same role as formal parameters in programming languages. Therefore, two terms structurally equal only differing in the choice of formal parameter names can be considered essentially the same. This equality is captured by an equivalence relation $=_\alpha$ on pre-terms, named α-*equivalence* and defined as the least congruence including the equality

$$\lambda x.M =_\alpha \lambda y.M[x := y] \qquad (y \notin FV(M)) \tag{α}$$

where $FV(M)$ is the set of all variables occurring free in M and $M[x := y]$ means the substitution of all free occurrences of x in M by y. To exemplify, the identity term corresponds to the equivalence class $\{\lambda a.a, \lambda b.b, \lambda c.c, \ldots\}$. The collection of all lambda terms is given by the quotient set $\Lambda/=_\alpha$. By convention, we will refer to terms by any of its representatives, chosen smartly to avoid capture of free variables during substitution.

The dynamics of a lambda term is given by function application, formalized by the beta-reduction relation. A term of the format $(\lambda x.P)\ Q$ is called a *redex* (reducible expression), and its respective *contractum* is the term $P[x := Q]$, i.e. the term P where all the free occurrences of x were substituted by Q. Notice that a single term may have several redex subterms. A term which does not contains any redex is called a *normal form*. A one-step reduction, written $M \to_\beta N$, represents the calculation of N by substituting a given redex in M by its contractum. Formally, it is defined by the following axiom schema

$$(\lambda x.P)\ Q \to_\beta P[x := Q] \tag{β}$$

together with adequate contextual extensions: if $P \to_\beta Q$ then $P\ M \to_\beta Q\ M$, $M\ P \to_\beta M\ Q$ and $\lambda x.P \to_\beta \lambda x.Q$. The reflexive and transitive closure of the one-step reduction $\to_\beta$ is the beta-reduction relation denoted $\twoheadrightarrow_\beta$. Beta-reduction aims to represent evaluation without changing the intrinsic "value" of the term. The beta-equivalence relation between terms identifies all terms that converge by beta-reduction: if $P \twoheadrightarrow_\beta M$ and $Q \twoheadrightarrow_\beta M$ then $P =_\beta Q$. If eventually a term reduces to a normal form, them that normal form assumer the role of "value" of all the terms that arrive at it.

As a theory, untyped lambda calculus has several interesting properties. One of them is that beta-reduction is confluent, usually referred as Church-Rosser: if $P \to_\beta M$ and $P \to_\beta N$ with $M \neq N$, there exists for sure a term Q such that $M \twoheadrightarrow_\beta Q$ and $N \twoheadrightarrow_\beta Q$. A consequence of confluence is the uniqueness of normal forms: it is impossible for a term reduce to distinct normal forms. Furthermore, as already mentioned, lambda-calculus is a universal formalism, and therefore it is possible to encode all kinds of data structures using lambda terms.

Lambda terms may be interpreted as functions and as arguments at the same time. Furthermore, both function creation and function application may be specified by combinators, i.e. terms without free variables. For instance, the combinator

$$K = \lambda x.\lambda y.x$$

receives a term a and produces the constant function $\lambda y.a$. We also have terms that, given two arguments, define the application of one as argument of the other.

$$App1 = \lambda x\ y.x\ y \qquad\qquad App2 = \lambda x\ y.y\ x$$

Another interesting fact is that there are combinators that produce fixed-points for any term (seen as a function). In other words, we have a lambda term Y such that, for every term M we have

$$M\ (Y\ M) =_\beta Y\ M$$

There are actually several of such combinators, where the simplest is referred as the Y combinator:

$$Y = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

The existence of fixed-points means that, given an abstraction $P = \lambda x.M$, we can obtain a term $Y\ P$ such that P acts as the identity. Another interpretation of is that of a potentially infinite (in fact finite, but as big as required) sequence of applications of P, as shown below

$$Y\ P = P(P(P...)))$$

The untyped lambda calculus may be also referred as a theory of unityped lambda calculus, since all the terms are taken as living in the same domain. The models for lambda calculi are the ones such the set D of terms and the set $D \to D$ of functions modifying terms are isomorphic: $D \cong D \to D$. This implies that the cardinality of the model of function application has to be restricted to a subset of the possible set-theoretical functions between two sets.

3.2 Simply typed lambda calculus

Lambda calculus may be extended with the notion of types, denoting classes of terms according to their inputs and outputs. Across the typed variations, the simply typed lambda calculus is (as the name suggests) the simplest. There are two main approaches, namely the Curry style (type-free terms) and the Church style (explicit types in lambda abstractions). In both styles, types are built from the following grammar

$$\sigma ::= \alpha \mid \sigma \to \sigma$$

in which α (and other greek letters) represents a *type name* drawn from a infinite but countable set *Types*. Terms in the Church style have type annotations in the arguments, and thus follow this grammar:

$$M, N ::= x \mid \lambda x^\sigma.M \mid MN$$

A *type environment* Γ is an association of variables names (in terms) to type names, denoted by the following grammar

$$\Gamma = \bullet \mid \Gamma; x : \sigma$$

where $\bullet$ represent the empty environment and the $\Gamma; x : \sigma$ operator denotes the extension or updating of Γ with $x : \sigma$. Type environments define the types for free variables in terms. A type judgement $\Gamma \vdash M : \sigma$ says that a term M has type α under the type environment Γ. Type judgments are obtained by the following inference rules

$$\frac{x : \sigma \ \in \ \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma; x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad N : \sigma}{\Gamma \vdash M\,N : \tau}$$

Definitions of α-equivalence, β-reduction and β-equivalence are essentially the same as in the untyped case. Simply typed lambda calculus has different properties in comparison with the untyped variation. Types stratify terms into layers: consider the following terms

$$Id_1 = \ \bullet \vdash \lambda x^{\alpha}.x : (\alpha \to \alpha)$$
$$Id_2 = \ \bullet \vdash \lambda z^{\alpha \to \alpha}.z : (\alpha \to \alpha) \to (\alpha \to \alpha)$$

Under the same empty environment, $Id_2\ Id_1$ is valid and has type $(\alpha \to \alpha)$. On the other hand, even the effect of both terms being the same, the term $Id_1\ Id_2$ may not be typed due to incompatibility in the types. Since types are always finite, some terms such as the $\lambda x.x\,x$ may not be typed, since x would have to be of type α and $\alpha \to \alpha$ at the same time: in this particular typed language there are no fixed points for the type constructor $\to$.

Simply typed lambda calculus enjoys strong normalization: all typed terms have a terminating and confluent beta-reduction. Concerning expressivenes, the fact that all beta-reductions terminate says that the halting problem for well-typed lambda terms is decidable. This shows that the simply typed lambda calculus is not a Turing computable language.

The importance of mentioning types is because we want to explore the hierarchy given by base types and the $\to$ type constructor to characterize higher-order terms. This will require the definition of a height for a given type. Since we have uniqueness of types for each term in the simply typed lambda calculi à la Church, a well-typed term has a unique height.

Definition 37 (Height of a type)**.** *The height $h(\sigma)$ of type σ is defined as*

$$h(\alpha) = 0$$
$$h(\sigma_1 \to \sigma_2) = \max(\ h(\sigma_1) + 1,\ h(\sigma_2) + 1\ )$$

Notice that this definition of height is not the same as the usual in the literature, where the return type height is not increased: our definition aims to denote as higher-order terms both the ones which receive and return functions.

Definition 38 (Higher-order term)**.** *A closed term M is a* higher-order term *iff $\vdash M : \sigma$ (it is well-typed) and $h(\sigma) \geq 2$.*

According to this particular characterization, the following terms are higher order:

- $K = \lambda x^{\alpha}.\lambda y^{\beta}.x : (\alpha \to (\beta \to \alpha))$
- $Id_2 = \lambda x^{\alpha \to \alpha}.x : (\alpha \to \alpha) \to (\alpha \to \alpha)$
- $L = \lambda x^{\alpha}.\lambda^{\beta} y.y : (\alpha \to (\beta \to \beta))$
- $App_3 = \lambda x^{\alpha \to (\beta \to \gamma)}.\lambda y^{\alpha \to \beta}.\lambda z^{\alpha}.(xz)(yz) : (\alpha \to (\beta \to \gamma)) \to (\alpha \to \beta) \to (\alpha \to \gamma)$

In fact, it is quite difficult to avoid using higher-order terms in typed lambda calculus. For instance, if we need to represent a numeric function $f : \sigma_1 \times \sigma_2 \ldots \times \sigma_n \to \sigma$ with more than one argument as a lambda term, it is required to convert it into a single-argument higher-order function $f' : \sigma_1 \to (\sigma_2 \to \ldots (\sigma_n \to \sigma) \ldots)$ through the well-known *currying* operation.

3.3 Comparison of beta-reduction and graph transformation

In this section, we study the similarities and distinctions between beta-reduction in lambda calculus and DPO graph rewriting. Then, we sketch a notion of a higher-order term in the graph rewriting setting based on the similarities of the formalisms. The analogies are not based on the literature, but rather on a critical analysis of the operational semantics of both models.

Both lambda calculus and the graph grammars are rule-based rewriting mechanisms. The objects being rewritten by beta-reduction are terms, specific trees built from a particular syntax and a countable set of variable names, while in graph rewriting we modify typed graphs. Both the untyped version of lambda calculus and graph grammars are Turing-complete models, being naturally capable of simulating each other. This simulation is not direct, however, since the nature of the rewriting differs. In the following discussion, we compare specific points of both kinds of rewriting. For brevity, we will employ the abbreviations LC, ULC and TLC, to denote, respectively, lambda calculus (in general), untyped lambda calculus and simply typed lambda calculus. The DPO approach for graph transformation will be abbreviated as GT. The comparison runs through some specific criteria, as follows.

Active elements: in LC, the elements representing functions to be applied are lambda abstractions, terms in the format $\lambda x.P$. In GT, the equivalent notion corresponds to graph transformation rules, denoted by injective spans $L \leftarrow K \rightarrow R$ of typed graphs.

Passive elements: in the ULC, any lambda term may be passed as a parameter for an abstraction, including the abstraction itself. In TLC, this is not the case, since the typing mechanism ensures that well-typed applications receive terms of the same type as declared in the lambda abstraction parameter. In the GT scenario, the passive elements are graphs typed over the same graph as the rule: for this purpose, the typing mechanism for graphs works in a similar way as in the TLC, ensuring a certain level of compatibility between active and passive elements.

Opacity of passive elements: within the scope M of a lambda abstraction $(\lambda x.M)$, the formal parameter x behaves like an atomic black box: its internal structure is unavailable, and the only possible way of manipulating it is including it in a bigger expression. On the other hand, graph transformation rules do not act on the graph as a whole, but rather on a particular region matching its left-hand side, and thus the internal structure of the graph is visible by definition.

Replication of passive elements: within the scope M of abstraction $\lambda x.M$, the term x may be "replicated" and appear an arbitrary number of times. In the DPO approach for graph rewriting with *injective spans*, replication is not possible. Although we can copy nodes and edges by creation, the new elements would always be accounted as distinct from the old ones, rather than "clones".

Rewriting process: A given lambda term M may contain n redexes, each implying a particular one-step beta reduction $M \rightarrow_\beta N_n$. The evaluation follows by choosing one redex $i \leq n$, applying it, and repeating the process for the resulting N_i term. Besides the choice of a particular redex, each one of them are reducible in exactly one way. For a term without redexes such as $(x\ z)$, the evaluation is *stuck*. On the context of GT, a rewriting always

depends on finding a particular match for rule $q : L \leftarrow K \rightarrow R$ on graph G. We now introduce the notation $@(q, G)$ and $@(P, G)$ to denote rule q and ruleset P actuating over graph G, respectively. Considering $@(P, G)$, there are two levels of nondeterminism to account for:

1. the choice of rule
2. the choice of a particular match for the chosen rule

The distinction we draw between a rule and a ruleset inside $@$ refers to the extent of its effect: a ruleset may affect a graph in many ways and in many places, while a rule affects the graph in a unique way in many places. We can argue that the equivalent notion of redexes in the GT are all the possible matches for $@(P, G)$. Notice that such matches are implicit and must be found at each reduction step, unlike redexes in a lambda expressions which are fixed by the structure of the term. When there is no possible rewriting for any of the rules in P for a given graph H, the evaluation is considered to be *stuck*.

Confluence: In LC, we have the Church-Rosser property, assuring the existence of a term P such that $N_n \twoheadrightarrow_\beta P$, for all n. In GT, we know that graph rewriting is not confluent in the general case. For the particular case of parallel independent rewritings, one-step confluence is assured by means of the local Church-Rosser property.

Dynamic transformations: Consider the beta-reduction $(\lambda x.M)P \rightarrow_\beta M[x := P]$. One interpretation is that the lambda abstraction $(\lambda x.M)$ is *consumed* together with its argument to provide the result. On the ULC, terms may have an infinite reduction by simply recreating previously consumed redexes: the famous example is the term $(\lambda x.x\,x)(\lambda x.x\,x)$, which always evaluate to itself. Lambda applications may be *created* dynamically as well: consider the term $(\lambda x.\lambda y.x)(\lambda z.z\,z)$. It is certainly not an abstraction in the format $\lambda x.M$, but it reduces to an abstraction in one step, and thus can be used as a function. As a counterpoint, graph rewriting rules work more or less like catalysts: they are not consumed by the process of graph rewriting, and can be applied as many times as required, being conserved by the whole rewriting process.

If we consider the original motivations for both formalisms, it is natural that they differ in so many points: lambda calculus intends to represent functions and effective computation, while graph transformation focus on modelling distributed and non-deterministic systems. However, it is also possible to identify commonalities by drawing analogies between elements of both languages. We start by identifying redexes in lambda calculus with $@(P, G)$, i.e, a ruleset P affecting a graph G. This association reflects the intuition of considering a typed graphs G the equivalent of a typed primitive value $v : \alpha$ and a rule (or ruleset) with a lambda abstraction $\lambda x^\alpha.M$, where $M[x := v] : \alpha$. Moreover, a *match m* may be associated with the application symbol $@$ in lambda calculus. This view is portraited in Figure 3.1.

It needs to be mentioned there are other possible associations. One example is to relate the β-reduction scheme, for which all redexes are matches, with a graph rule or ruleset in the graph transformation setting. This way, all possible redexes are all the distinct matches for the reduction scheme. The argument against this view, however, it that it does not reflect the common characteristic of graph rewriting rulesets and lambda abstractions being both *programmed transformations*, i.e. constructions the programmer or modeller will manipulate in order to represent a desired behavior.

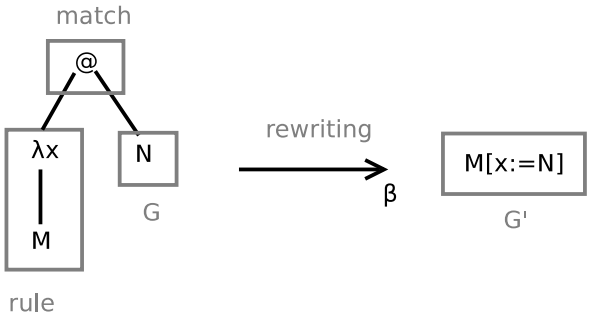


Figure 3.1: Analogy between beta-reduction and graph transformation.

Let us move forward in our analogy by considering the typing of higher-order terms in the simply typed lambda calculus. We can claim that

$$(\alpha \to \alpha) \to (\alpha \to \alpha)$$

i.e., the type of second-order functions that transform first-order functions of type $(\alpha \to \alpha)$, may be associated to the concept of *second-order rules modifying graph transformation rules*, as shown in Figure 3.2 by our current analogy. The type

$$((\alpha \to \alpha) \to (\alpha \to \alpha)) \to ((\alpha \to \alpha) \to (\alpha \to \alpha))$$

is associated with *third-order rules transforming second-order rules*, and so on. That is the first intuition, and follows from the abstract notion of higher-order modifications. Other higher-order types, however, may suggest more complex effects in the setting of graph transformation. For instance,

$$\alpha \to (\alpha \to \alpha)$$

may be associated with the notion of *rules for converting graphs into graph transformation rules*; the type

$$(\alpha \to \alpha) \to \alpha$$

with the opposite operation, *rules encoding rules into graphs*. Since graph transformation rules only modify the argument rather than fully consume it and return the output, the realization of such inter-level terms would require a common graph-based representation for both graphs and graph transformation rules, as we have in the untyped lambda calculus. In other words, a unique representation for both $(\alpha \to \alpha)$ and α. This, however, will not be the focus of this work. Another interesting suggestion may be obtained from types such as

$$(\alpha \to \alpha) \to (\beta \to \beta)$$

In this case, since base types such as α and β are associated with typed graphs, a possible reading is the one of *rules retyping other rules*, probably through modifications in the type graph.

For our purposes of representing model transformations, we intend to focus on a proper definition for second-order rules that modify a collection of first-order rules. The intuition is to have higher-order rules defining model transformations while conventional rules represent the semantics of the system itself. There are several important details to consider about this idea:

1. how to generalize the notion of higher-order rules in the DPO approach for graph transformation.

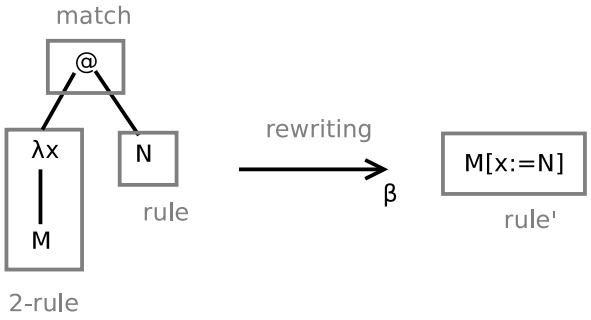


Figure 3.2: Analogy between beta-reduction and rule transformation.

2. given that "rules modifying rules" are provided, how to handle modifications in the structure of the grammar, such as addition and creation of new rules in the rule collection? Can modification, creation and deletion be modelled by mean of higher-order rules, or higher-order collections of rules? Moreover, how to deal with other components of a specification, such as the initial graph and type graph?

3. how to characterize the interaction between the two levels of rewriting. Is there an equivalent notion of critical pair between higher-order rules and lower order rules?

In the next chapters we intend to discuss those issues in detail, introducing an adequate solution to each one of them.

3.4 Summary

This chapter started with a brief review of untyped and simply-typed variations of lambda calculus, which was done to introduce the notion of *higher-order terms* in simply-typed lambda calculus. The original part of this chapter comes from the comparison between the rewriting principles behind lambda calculus and graph transformation. Based on this analogy, we sketched how some types of lambda terms suggest some kinds of higher-order rewriting rules for the graph transformation setting.

4 SECOND-ORDER GRAPH REWRITING

This chapter faces the problem of defining a suitable notion of modification for graph transformation rules, i.e. how to obtain a notion of second-order rule which affect conventional (first-order) rules in a graph grammar specification. We follow the track of generalizing the algebraic DPO approach: first, we need to fix a category in which rules are objects and second-order rules are monic spans; then, we attempt to use DPO diagrams in such categories to describe second-order rewriting. For such, we have considered the category of arbitrary spans, named T-**Span**, and also its subcategory of monic spans objects, named T-**Rules**, and have identified some issues regarding DPO diagrams in both cases. We arrive at the conclusion that second-order rule rewriting can be better defined through DPO rewriting in T-**Span** with negative application conditions, which we take as a new notion of second-order rewriting. The original contributions of this chapter are the following:

- discussion regarding DPO diagrams in T-**Span** and T-**Rules**, and their issues;
- precise definition of second-order rules and second-order rewriting;
- determination of the importance of NACs in several aspects of second-order rewriting;
- characterization of T-**Span** as an adhesive HLR category with NACs.

We start by defining the category of arbitrary spans of typed graphs.

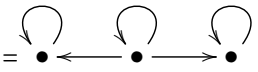
4.1 DPO diagrams in the category of graph spans

In the DPO approach, graph transformation rules are diagrams in the format $L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R$ where both l and r are monomorphisms. In the definition of rules, monic spans are considered instead of arbitrary ones because this simplifies considerably the rewriting theory. For instance, uniqueness of pushout complements, essential characteristic for DPO rewriting, holds in adhesive categories only along diagrams with monomorphisms. Another reason is that we also avoid to consider more complex (although interesting) situations such as cloning of graph elements – which occurs if l is not mono – and confusion – which follows from non-monic r. Since monic spans are special cases of arbitrary spans, we begin defining morphisms between arbitrary spans, and after, consider its subcategory containing only rules as objects.

Definition 39 (Span morphism)**.** *Let $p = (l_p, r_p)$ and $q = (l_q, r_q)$ be spans in a category $\mathscr{C}$. A* span morphism *$f : p \to q$ is a triple (f_L, f_K, f_R) of morphisms between the objects of both spans such that the following diagram commutes in $\mathscr{C}$.*

$$\begin{array}{ccc} R_p & \xrightarrow{f_R} & R_q \\ {\scriptstyle r_p}\uparrow & & \uparrow{\scriptstyle r_q} \\ K_p & \xrightarrow{f_K} & K_q \\ {\scriptstyle l_p}\downarrow & & \downarrow{\scriptstyle l_q} \\ L_p & \xrightarrow{f_L} & L_q \end{array}$$

Observe that every span over $\mathscr{C}$ can be seen as a particular functor from the small category **Span**, depicted below,

Span = 

to the category $\mathscr{C}$. Following this view, the presented definition of span morphisms coincides with the definition of natural transformations between functors of **Span** over $\mathscr{C}$. This allows us to define the category of spans over $\mathscr{C}$ as the category of functors $[\mathbf{Span} \to \mathscr{C}]$. Notice that we are talking about *concrete spans* in opposition to *abstract ones*, i.e. two distinct spans with isomorphic central objects are still considered to be distinct. Returning to the context of graphs, it is clear that the category of spans of typed graphs is an instantiation of this construction.

Definition 40 (Category of spans of typed graphs)**.** *The category* T-**Span** *of spans over* T-**Graph** *is the category of functors* $[\mathbf{Span} \to T\text{-}\mathbf{Graph}]$.

This way of defining T-**Span** is technically useful because it allows the inheritance of properties from T-**Graph**. Monomorphisms and epimorphisms are simply triples of the monos and epis in T-**Graph** and limits and colimits are also constructed component-wise. In particular, we have a suitable category for double-pushout rewriting, because

Proposition 41. T-**Span** *is an adhesive category.*

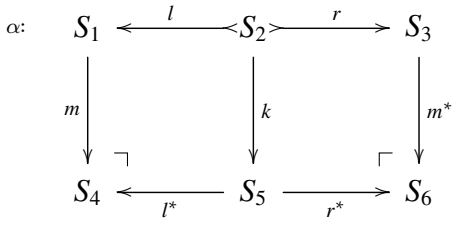
Proof. By construction: it is known that T-**Graph** is adhesive, **Span** is a small category and, as shown in (LACK; SOBOCIŃSKI, 2003), if $\mathscr{D}$ is adhesive and $\mathscr{C}$ is small, then the functor category $[\mathscr{C} \to \mathscr{D}]$ is also an adhesive category. □

This is a simple but important fact, which allows us to define rewriting of spans in the most natural way: by taking monic spans (of spans) as rules and double-pushout diagrams as rewriting steps. In the following, to distinguish between spans *of graphs* (objects of T-**Span**) and spans *in* T-**Span** (or spans of spans of graphs), we refer to the latter as *2-spans*. As with graph transformation rules, we consider monic spans as the subset of rules, which we call *2-rules*.

Definition 42 (2-rule)**.** *A 2-span* $S_L \xleftarrow{l} S_K \xrightarrow{r} S_R$ *such that* S_L, S_K, S_R *are graph rules (monic graph spans) and* l, r *are monic span morphisms will be called a* 2-rule.

Notice that we are also enforcing that all objects in the monic span are actually graph rules themselves. As we do with graphs, we take DPO diagrams in T-**Span** as the basis for the notion of rewriting.

Definition 43 (Span rewriting)**.** *Given a 2-rule* $\alpha = S_1 \overset{l}{\leftarrow} S_2 \overset{r}{\rightarrow} S_3$ *and a span morphism* $m : S_1 \rightarrow S_4$ *(representing a match), we say that the 2-rule* α *rewrites the span* S_0 *into* S_6 *iff there is a double-pushout diagram in* T-**Span** *as shown below.*



As with graphs, we denote span rewriting by $S_4 \overset{\alpha,m}{\Longrightarrow} S_6$*. We may also refer it as a* one-step span derivation*.*

An interesting characteristic of span rewriting is that we can visualize a double pushout in T-**Span** as a diagram in T-**Graph**, as shown in Figure 4.1. This allows us to inspect the rewriting locally over the three graphs that compose the target span.

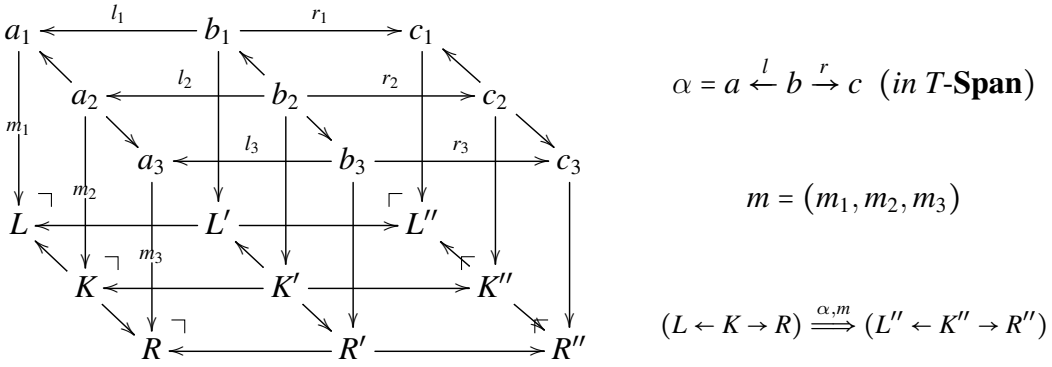


Figure 4.1: Representation of DPO span rewriting in T-**Graph**.

Example 44. *In Figure 4.2 we have an example of double-pushout rewriting in* T-**Span**. *For simplicity, we consider the type graph T to be the unitary graph with one node and no edges. Each span is represented by three shadowed regions, representing (from left to right, downwards to upwards) the LHS, interface and RHS. The mapping of individual elements in each span is given by small gray arrows, while span morphisms are denoted by big, black arrows. In this particular case, the span being rewritten, if interpreted as a graph rule, has the effect of deleting one element (x) and preserving another one (a,b,c). The 2-rule (roof of the diagram) affects the original span as follows: i) it converts a preserved element (1,2,3) into a deleted-and-recreated one by erasing its representation in the interface; ii) a new element (4) is introduced into the RHS, increasing the number of created elements. The match associates 1,2 and 3 to a, b and c, respectively. The resulting span, interpreted as a graph rule, deletes two nodes (x and a) and creates other two (4 and c).*

As it is natural in the DPO approach, the existence of a pushout complement (POC) depends on particular conditions. If the POC exists, however, we know it is essentially

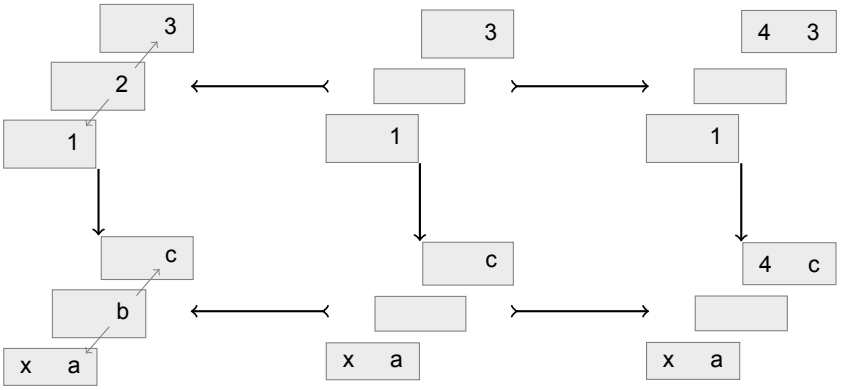
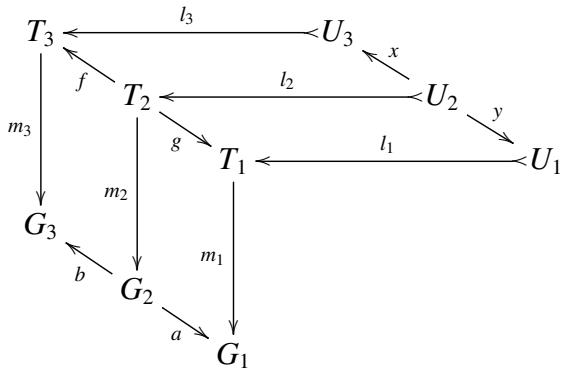


Figure 4.2: Double pushout rewriting of spans.

unique (up to isomorphism), because T-**Span** is an adhesive category and we defined 2-rules from monic span morphisms.

Proposition 45 (Application conditions for DPO span rewriting)**.** *Given the diagram $G \overset{m}{\leftarrow} T \overset{l}{\nleftarrow} U$ in T-**Span**, there exists a pushout complement $G \overset{l^*}{\nleftarrow} V \overset{k}{\leftarrow} U$ iff the following conditions are satisfied:*

1. *dangling and identification conditions hold for each component $G_i \overset{m_i}{\leftarrow} T_i \overset{l_i}{\nleftarrow} U_i$ in T-**Graph**, for $i \in \{1,2,3\}$ as shown below.*



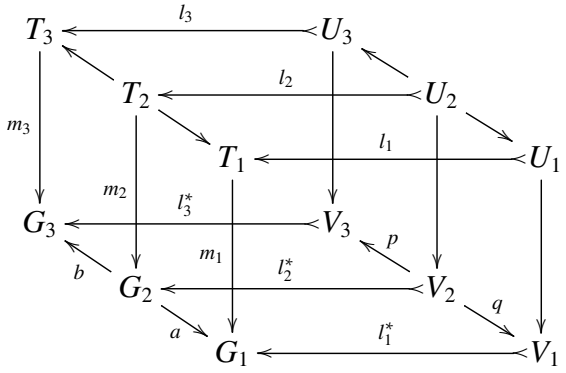
2. *(dangling span condition) an element x in G_1 (or G_3) "deleted" by l_1 (or l_3) must have all elements in $a^{-1}(x)$ (or in $b^{-1}(x)$) also "deleted" by l_2. Formally,*

   (a) $\forall x \in m_1(T_1), \quad Del_1(x) \rightarrow (\forall y \in a^{-1}(x), Del_2(y))$

   (b) $\forall x \in m_3(T_3), \quad Del_3(x) \rightarrow (\forall y \in b^{-1}(x), Del_2(y))$

   where $Del_i(j) = (\exists k \in T_i,\ m_i(k) = j) \quad \wedge \quad (\nexists m \in U_i,\ l_i(m) = k)$

Proof. We need to show that the conditions of Definition 45 are sufficient for the existence of pushout complement in T-**Span**.



1. the pushout complements containing V_1, V_2 and V_3 are obtained from condition 1 in Proposition 45.

2. (a) l_1^* is mono, because l_1 is mono and pushouts preserve monomorphisms in T-**Graph**. Hence, if $x \in l_1^*(V_1)$ then there is a unique y such that $l_1^*(y) = x$, and we denote $l^{*-1} : l_1^*(V_1) \to V_1$ by $l^{*-1}(x) = y$;

   (b) condition 2 in Proposition 45 implies that $a(G_2) \subseteq l_1^*(V_1)$. It follows that $a \circ l_2^*(V_2) \subseteq l_1^*(V_1)$;

   (c) from facts (a) and (b), we can define $\forall x \in V_2,\ \ q(x) = l^{*-1} \circ a \circ l_2^*(x)$. We have a similar definition for p.

□

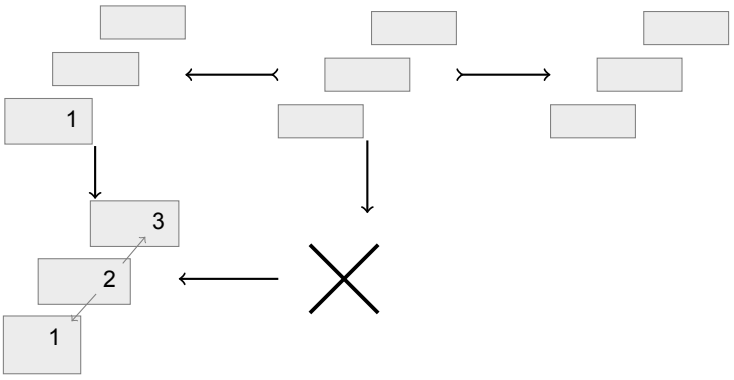


Figure 4.3: Invalid match in DPO span rewriting.

Intuitively, the dangling span condition says that there is not pushout complement if an element is deleted from the LHS or the RHS of the span without deleting all its pre-images in K. The following example presents this situation.

Example 46 (Application conditions)**.** *Figure 4.3 shows a situation where the rewriting is not possible because the match does not satisfy the dangling span condition. There would be no element in the LHS of the POC object to associate the element 2 (out of the match, and thus maintained by the rewriting).*

Given that we have a notion of rule, match, application conditions, and rewriting, the category T-**Span** seems very promising as a framework for second-order DPO rewriting. It remains only to test if DPO diagrams in T-**Span** preserve the property of the rewritten

span being a rule. The next two propositions show that this holds for the leftmost part of the DPO construction, i.e., the calculation of pushout complements.

Proposition 47. *If $f : A \to B$ in T-**Span** is monic and B is a rule, then A is a rule.*

Proof. Consider the following diagram in T-**Graph**, representing $f : A \to B$. The composition $f_K; l_B$ is monic and also is equal to $l_A; f_L$. Therefore l_A must also be a monomorphism by decomposition of monic arrows. Similarly, we have that r_A is also mono, which entails A is a rule.

$$\begin{array}{ccc} R_A & \xrightarrow{f_R} & R_B \\ {\scriptstyle r_A}\uparrow & & \uparrow{\scriptstyle r_B} \\ K_A & \xrightarrow{f_K} & K_B \\ {\scriptstyle l_A}\downarrow & & \downarrow{\scriptstyle l_B} \\ L_A & \xrightarrow{f_L} & L_B \end{array}$$

□

Proposition 48. *Consider a diagram $S_0 \overset{a}{\leftarrow} S_1 \overset{b}{\leftarrow} S_2$ in T-**Span** such that S_0, S_1, S_2 are rules and b is monic. Then, the object S_3 of the unique pushout complement $S_0 \overset{c}{\leftarrow} S_3 \overset{d}{\leftarrow} S_2$ is a rule.*

Proof. Because T-**Span** is adhesive, pushouts preserve monomorphisms. Therefore, if b is mono then c is also mono, and, by Proposition 47, we have that S_3 is a rule. □

Thus, we can say that the leftmost part of a DPO diagram does preserve rules. Notice that this is true independently of the match being injective or not. Now we need to consider the rightmost part of the DPO diagram: unfortunately, as shown by the following example, we do not have rule preservation.

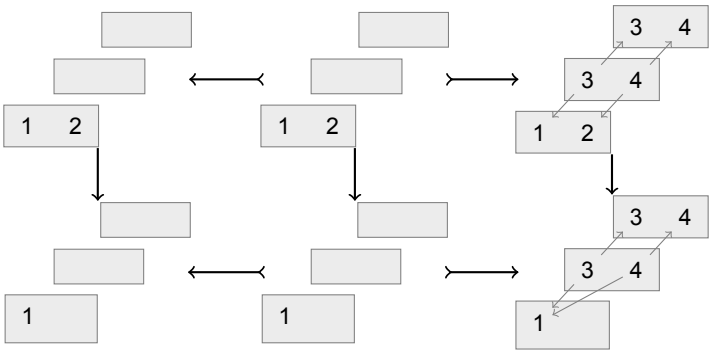


Figure 4.4: Double pushout diagram in T-**Span**

Example 49. *Figure 4.4 presents a double pushout diagram in the category T-**Span**, The 2-rule on the roof of the diagram has the effect of enforcing that some elements in the LHS of the span become "preserved" (by creating respective nodes in the interface and RHS). Notice that this particular match is not injective, and also that the resulting span cannot be considered a graph rule since it is not monic.*

This example elucidates an important characteristic of pushouts in T-**Span**: they are not closed under rules. Formally speaking,

Proposition 50. *Consider a 2-span $s = S_1 \leftarrow S_2 \rightarrow S_3$, where S_1, S_2 and S_3 are graph rules. If we calculate the pushout $PO(s) = S_1 \rightarrow S_4 \leftarrow S_3$, then it may not be the case that S_4 is a rule.*

Proof. For such, we just need a counterexample. One was already presented in Figure 4.4 as the rightmost pushout of the DPO diagram. Another counterexample is introduced in Figure 4.5: this case is particularly interesting because the diagram consists of a monic 2-span where all three objects are rules, and yet the object of the pushout is a non-rule. Therefore, injectivity of all morphisms does not suffice to ensure preservation of rules in pushouts. □

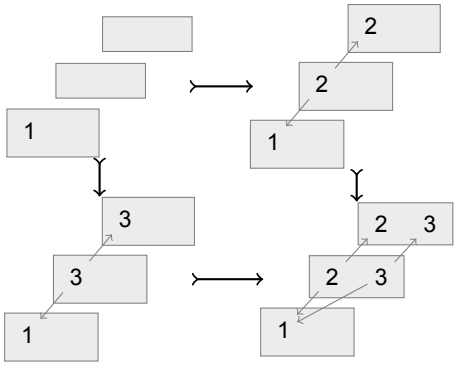


Figure 4.5: Pushout of monic 2-span of rules in *T*-**Span**

4.2 DPO diagrams in the category of rules

The fact that DPO rewriting of rules in *T*-**Span** may generate non-rules is a clear problem: if we change a rule p into a non-rule p', we may have unpleasant surprises when applying p' to modify graphs. For instance, if p' is not left-linear, which means that $l : K \rightarrow L$ is not mono, we will have a non-deterministic result. In other words, it would be possible to obtain as result of the rewriting two distinct, non-isomorphic graphs from the same match of p'. The need to maintaining rules in second-order rewriting motivates the investigation of a more restricted scenario than *T*-**Span**, i.e., a subcategory where we only consider rules as objects.

Definition 51 (Category of typed graph rules)**.** *The category T-***Rules** *is the full subcategory of T-***Span** *such that all objects are monic typed graph spans.*

This definition for *T*-**Rules** is quite straightforward, and it coincides with the definition of category **MSpan** in (CORRADINI et al., 1996). Notice that morphisms between objects do not need to be mono. For our purposes, it is relevant to study the properties of *T*-**Rules**, such as the existence of limits and colimits, and – very important – to check if it is an adhesive or adhesive HLR category.

A reasonable starting point for studying *T*-**Rules** is to focus on its relationship with *T*-**Span**. For instance, one possible question is whether the obvious inclusion functor

$$incRule : T\text{-}\mathbf{Rules} \rightarrow T\text{-}\mathbf{Span}$$

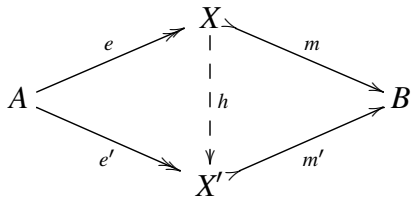
has a left or right adjoint, which would give us information about how to construct limits and colimits in T-**Rules**. We have found that there is a left-adjoint to *incRule*, which we refer as

$$toRule : T\text{-}\mathbf{Span} \to T\text{-}\mathbf{Rules}$$

Intuitively, *toRule* is a free functor, which generates canonically a rule from an arbitrary span. In the following, we will provide its definition in detail and also a proof of the adjunction. As a technical requirement, we need to introduce choice functions for pushouts and epi-mono factorization in T-**Graph**. Choice functions are interesting because they allow the definition of functions based on non-unique constructions such as pushouts. Notice this is adequate for our purposes because we are aiming to define a functor adjoint to *incRule*, and adjoints are known to be unique only up to a natural isomorphism (i.e. there are various different isomorphic functors differing only on the particular choice functions used).

Definition 52 (Choice of pushout)**.** *A choice of pushout for a category $\mathscr{C}$ is a function PO which maps every span $d = B \leftarrow A \rightarrow C$ to a fixed cospan $PO(d) = B \rightarrow D \leftarrow C$.*

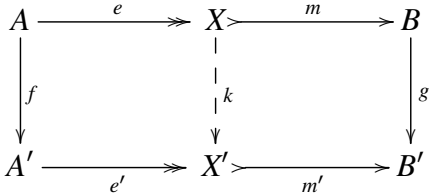
Definition 53 (Epi-mono factorization)**.** *A category $\mathscr{C}$ has epi-mono factorization iff every morphism $f : A \rightarrow B$ can be decomposed into a pair $e; m$ where e is epic and m is mono such that, for all other decompositions $e'; m'$, (with e' epic and m' mono) there is a unique morphism $h : X \rightarrow X'$ making the following diagram commute. Moreover, h is an isomorphism.*



Definition 54 (Choice of factorization)**.** *A choice of factorization for a category $\mathscr{C}$ (with epi-mono factorization) is a function EM that maps a given morphism f to a fixed pair of morphisms (e, m) such that e is epi, m is mono and $e; m = f$.*

The categories **Set**, **Graph** and T-**Graph** are known to have epi-mono factorization (EHRIG et al., 2005). For the following proof of adjoint situation, we will also require a derived result from the existence of epi-mono factorization, presented as Proposition 4.4.5 in (BORCEUX, 1994).

Proposition 55. *Consider the following diagram in a category with epi-mono factorization.*



There exists a unique morphism $k : X \rightarrow X'$ such that both inner squares commute.

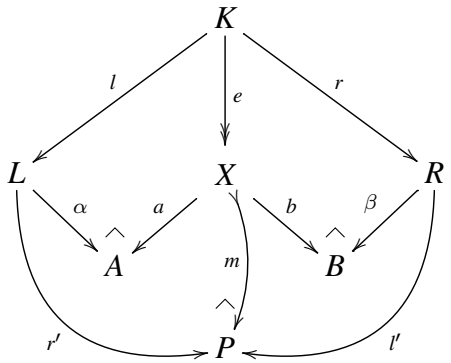
Proof. Consider the epi-mono factorizations $EM(f) = A \overset{a}{\twoheadrightarrow} A'' \overset{b}{\rightarrowtail} A'$, $EM(g) = B \overset{c}{\twoheadrightarrow} B'' \overset{d}{\rightarrowtail} B'$, $EM(m; c) = X \overset{x}{\twoheadrightarrow} Y \overset{y}{\rightarrowtail} B''$, and $EM(b; e') = A'' \overset{z}{\twoheadrightarrow} Z \overset{w}{\rightarrowtail} X'$. Since $A \overset{a;z}{\twoheadrightarrow} Z \overset{w;m'}{\rightarrowtail} B'$ and $A \overset{e;x}{\twoheadrightarrow} Y \overset{y;d}{\rightarrowtail} B'$ are two epi-mono factorizations for $e; m; g$, there is a unique morphism $u : Y \rightarrow Z$. Finally, set $k = x; u; w$. □

Choice functions for pushouts and epi-mono factorizations are used directly in the definition of *toRule*, as shown.

Definition 56 (*toRule* function)**.** *Given a graph span $p = L \xleftarrow{l} K \xrightarrow{r} R$, we first perform the following calculations in T-***Graph***.*

1. *the pushout $PO(p) = L \xrightarrow{r'} P \xleftarrow{l'} R$*
2. *the arrow $k = l; r' = r; l'$*
3. *the factorization $EM(k) = (e, m)$*
4. *the pushout $PO(L \xleftarrow{l} K \xrightarrow{e} X) = L \xrightarrow{\alpha} A \xleftarrow{a} X$*
5. *the pushout $PO(X \xleftarrow{e} K \xrightarrow{r} R) = X \xrightarrow{b} B \xleftarrow{\beta} R$*

as shown in the next diagram, and we finally define $toRule(p) = A \xleftarrow{a} X \xrightarrow{b} B$.



Example 57. *The effect of the function toRule can be seen in Figure 4.6. Intuitively, it combines all elements in the interface graph which are mapped to the same element in the LHS or in the RHS and then propagates such merging to the LHS and RHS as well.*

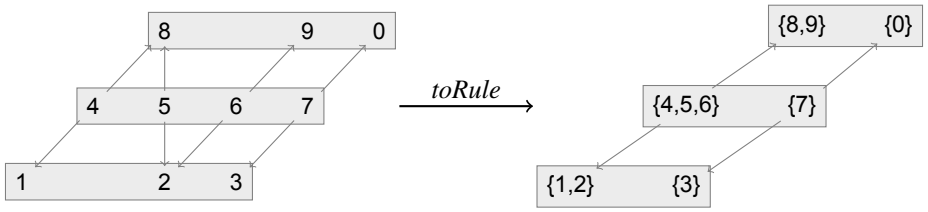


Figure 4.6: Example of application of *toRule*.

Proposition 58. *The toRule function gives raise to a functor T-***Span** $\rightarrow$ *T-***Rules***.*

Proof. We need to show that

- $A \xleftarrow{a} X \xrightarrow{b} B$ is a monic span;
- the *toRule* mapping of objects extend canonically for morphisms.

The first requirement is obtained by analyzing the the diagram of Definition 56, as follows:

1. there is a unique morphism $h_A : A \to P$ since (r', m) is a pre-pushout of (α, a).

2. since m is mono and $m = a; h_A$, then we have that a is mono. The same argument shows that b is mono too.

For the second requirement, we need to consider an arbitrary span morphisms $g = \langle g_L, g_K, g_R \rangle : p_1 \to p_2$. Consider we draw the diagram of Definition 56 for p_1 and p_2, and add the morphism components.

1. there is a unique morphism $h_P : P_1 \to P_2$ since $(r'_2 \circ g_L,\ l'_2 \circ g_R)$ is a pre-pushout of (r'_1, l'_1);

2. since $g_K; e_2; m_2 = e_1; m_1; h_P$ and both $e_1; m_1$ and $e_2; m_2$ are epi-mono factorizations, then according to Proposition 55 there is a unique morphism $h_X : X \to X'$.

3. there is a unique morphism $h_A : A_1 \to A_2$ since $(\alpha_2 \circ g_L, a_2 \circ h_X)$ is a pre-pushout of (α_1, a_1).

4. there is a unique morphism $h_B : B_1 \to B_2$ since $(b_2 \circ h_X, \beta_2 \circ g_R)$ is a pre-pushout of (b_1, β_1).

From the obtained unique arrows, we set $toRule(g) = \langle h_L, h_X, h_R \rangle$. □

Now, we show that *incRule* and *toRule* are actually adjoints. More precisely, *toRule* is the left-adjoint, *incRule* is the right adjoint and the unit μ is obtained from the construction of *toRules*, as follows.

Definition 59 (unit μ)**.** *For each span s in T-***Span***, the morphism $\mu_s : s \to incRule \circ toRule$ is the span morphism $\langle \alpha, e, \beta \rangle$ as shown in the diagram of Definition 56. The collection of μ_s for all objects $s \in T$-***Span*** is the natural transformation $\mu : T$-***Span*** $\to incRule \circ toRule$.*

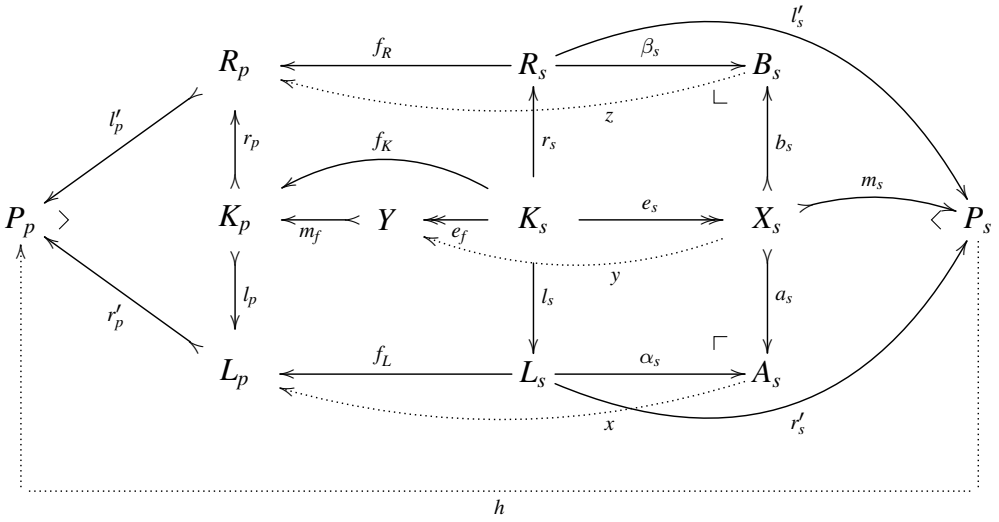
Proposition 60. *For all $s \in T$-***Span***, μ_s is an epimorphism.*

Proof. We just need to show that all the components of μ_s are epi in T-**Graph**. This follows from inspection on the diagram of Definition 56. The arrow e is obviously epi. Since pushouts always preserve epi arrows, we have that both α and β are also epi. □

Theorem 61. *The functor toRule is the left-adjoint of incRule, with μ as unit.*

$$T\text{-}\mathbf{Span} \underset{incRule}{\overset{toRule}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} T\text{-}\mathbf{Rules}$$

Proof. We need to show that for all $f : s \to incRule(p)$, we have a unique morphism $h : toRule(s) \to p$ such that $\mu_s; incRule(h) = f$. For this, let us draw first the components of f and μ_s in T-**Graph**.



Then, we apply the following steps:

1. calculate the pushout $PO(p) = L_p \xrightarrow{r'_p} P_p \xleftarrow{l'_p} R_p$. Since T-**Graph** is adhesive, we have that pushouts preserve monomorphisms. We can then conclude that l'_p and r'_p are mono.

2. calculate the pushout $PO(s) = L_s \xrightarrow{r'_s} P_s \xleftarrow{l'_s} R_s$. There exist a unique arrow $h : P_s \to P_p$ due to pre-pushout $(l'p \circ f_R, r'_p \circ f_L)$.

3. obtain a factorization $EM(f_K) = (e_f, m_f)$.

4. unique morphism $y : X_s \to Y$ due to Proposition 55 and two epi-mono factorizations $e_s; m_s$, $e_f; (m_f; r_p; l'_p)$ and commutativity $e_s; m_s; h = i_{K_s}; e_f; (m_f; r_p; l'_p)$.

5. unique morphism $z : B_s \to R_p$ due to pre-pushout $(r_p \circ m_f \circ y, f_R)$.

6. unique morphism $x : A_s \to L_p$ due to pre-pushout $(l_p \circ m_f \circ y, f_L)$.

Thus, we have a unique span morphism $u = \langle x, m_f \circ y, z \rangle$ such that $\mu_s; incRule(u) = f$. □

Theorem 61 is relevant because it provides us enough information regarding the construction of limits and colimits in T-**Rules**. For instance, the fact that the faithful inclusion functor *incRule* has a left adjoint characterizes T-**Rules** as a *reflective subcategory* of T-**Span**, which entails the following:

- the co-unit ε of the adjunction $(incRule, toRule, \mu)$ is a natural isomorphism;
- from the adjunction result, *toRule* preserves colimits and *incRule* preserves limits;
- T-**Span** is closed for limits, i.e., limits in T-**Rules** are the same as the ones in T-**Span** for the same diagram;
- colimits in T-**Rules** are related to colimits in T-**Span**, i.e. we can calculate the colimit of a diagram in T-**Rules** by calculating the colimit in T-**Span** and then applying the natural transformation μ on the colimit object.

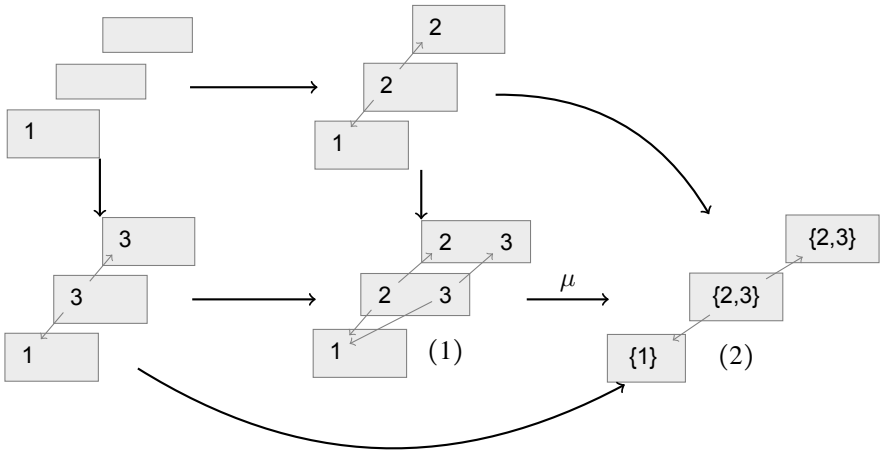


Figure 4.7: Pushout in T-**Span** (1) and in T-**Rules** (2).

Example 62. *Figure 4.7 shows how to obtain the pushout of a diagram in* T-**Rules** *by calculating the pushout in* T-**Span** *first and applying the unit of the adjunction.*

Now, given that we know how to calculate limits and colimits in T-**Rules**, we can argue about the suitability of the category for DPO rewriting: in other words, if it is adhesive. Unfortunately, the answer to this question is negative, as it is clarified by next two propositions.

Proposition 63. *Given a diagram* $S \overset{m}{\leftarrowtail} T \overset{l}{\leftarrowtail} U$ *in* T-**Rules** *such that* l *is mono, we do* not have *unique pushout complement.*

Proof. By conterexample: Figure 4.8 shows two alternative, non-isomorphic pushout complements in T-**Rules** for the same diagram. Notice that one of them reproduces the effect of "deleting" some elements, while the other "preserves" all the original elements. This is a consequence of the merging effect caused by the unit morphims μ (in T-**Span**). □

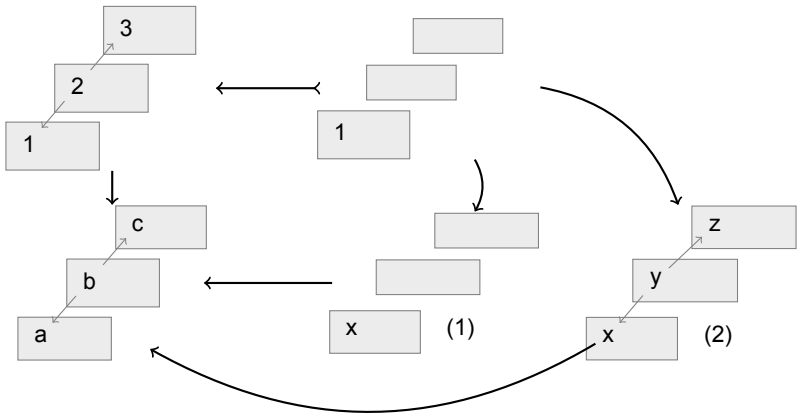


Figure 4.8: Two distinct POCs, (1) and (2), for the same diagram in T-**Rules**.

Proposition 64. T-**Rules** *is not adhesive.*

Proof. By contraposition, since uniqueness of pushout complement for diagrams in the format $A \leftarrow B \nleftarrow C$ follows from adhesiveness. □

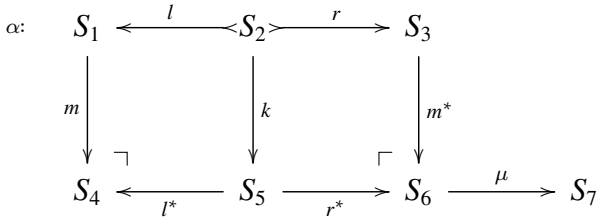
The consequence of Proposition 63 is that not all DPO diagrams in T-**Rules** have the expected intuitive effect of deletion and creation of graph elements. Although creation of elements is "correct" because it produces only well-formed graph rules, deletion depends on the choice of pushout complement, which is not unique. This contrasts with T-**Span**, where deletion is "well-defined" but creation may generate ill-formed rules. We observe that neither category offers a context where rule rewriting can be performed using DPO rewriting in its simplest form. Therefore, we are forced to consider more advanced mechanisms in order to provide a notion of second-order graph rewriting.

Considering the two defined categories, we can defend that T-**Span** is a better starting point than T-**Rules** since it works adequately for arbitrary spans. What remains is to provide a mechanism to ensure that DPO rewriting of rules generates only rules as result. The following two sections explore two different approaches to this same issue: the first approach considers the *correction* of the resulting span by means of the unit μ. The second approach goes in the direction of *avoiding* rewritings which generate non-rules, using for such purpose the mechanism of negative application conditions.

4.3 Rule rewriting correcting rule invalidation

This section considers the approach of *correcting* the resulting span in order to enforce it is a rule. This approach explores the fact that there is a canonical way of converting arbitrary spans into rules: the unit μ. Since DPO rewriting in T-**Span** fails to preserve rules only because of its rightmost part, we could attempt to use the pushout construction in T-**Rules** instead, which we know can be obtained in T-**Span** by applying μ to the object of the pushout. This way, we would have both well-formed deletion and creation.

Definition 65 (Rule rewriting with correction)**.** *Given a monic 2-rule $\alpha = S_1 \overset{l}{\leftarrow} S_2 \overset{r}{\rightarrow} S_3$, a rule S_4 and a span morphism $m : S_1 \rightarrow S_4$, we say that the 2-rule α rewrites the rule S_4 into S_7 with correction iff $S_4 \overset{\alpha,m}{\Longrightarrow} S_6$ (span rewriting) and $cod(\mu_{S_6}) = S_7$, where μ is the unit of the adjunction where incRule is a right-adjoint.*



We denote second-order rule rewriting with correction by $S_4 \overset{\alpha,m}{\Longrightarrow}_{\mu} S_7$.

Example 66 (Rule rewriting with correction)**.** *Figure 4.9 presents an example of second-order rewriting for a non-injective match.*

Intuitively speaking, this notion of rewriting corresponds to calculating pushout complements in T-**Span** (for deletion) and a pushout in T-**Rules** (for creation of elements). Notice also that it does not correspond to the traditional DPO rewriting, since it contains and additional morphism μ that introduces some asymmetry, since rule applications are not always invertible. The main advantage of this approach would be to obtain a precise

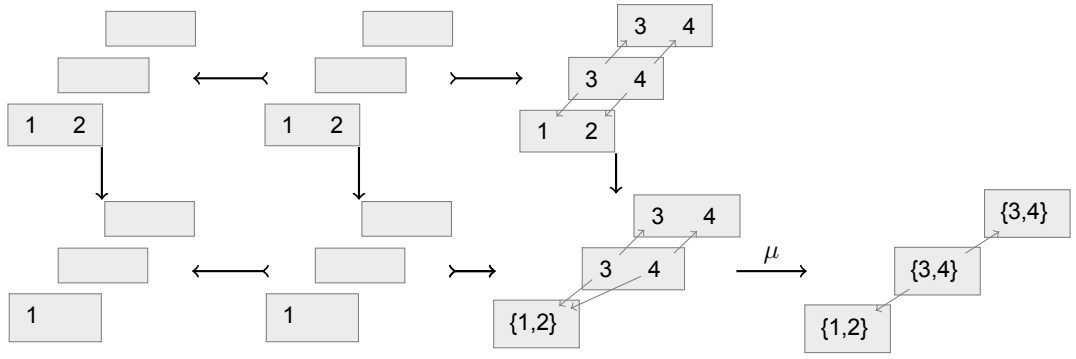


Figure 4.9: Rule rewriting with correction.

and definite notion of rewriting, based on free creation of rules given canonically by the adjoint situation between T-**Span** and T-**Rules**. However, there are some idiosyncrasies introduced by the unit μ, being the most remarkable the possibility of merging created and preserved elements. This has indeed some important consequences, which we illustrate through examples.

Example 67. *Figure 4.10 shows a rule rewriting with correction over a very simple graph rule which preserves one single node. The 2-rule has the effect of converting a deleted node and a created node in the graph rule into a preserved node by creating a pre-image in the interface. However, the 2-rule matches an already preserved element in the graph rule, creating a redundant pre-image in its interface. By the effect of the unit μ, the original and the new pre-image are merged into a single node. Thus, the overall rewriting, although possible, does not change the graph rule at all, even if the 2-rule definition shows one node being created in the interface. This shows that the notion of correcting may result in rewritings with null effect from a 2-rule which creates something.*

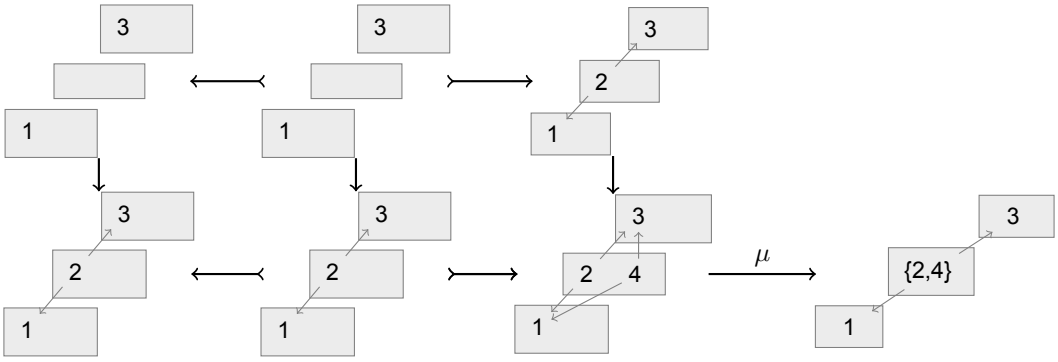


Figure 4.10: Rewriting with null effect due to merging in rule rewriting with correction.

Example 68. *Figure 4.11 shows the consequences of the merging effect over the notion of parallel independence. We have two 2-rules, α, which changes a deleted-created pair of nodes into a preserved node, and β, which has the exact opposite effect, deleting the element from the interface. Below, we see the two possible applications of the 2-rules over a rules which simply preserves one element. Notice first that the match for both applications only touches preserved elements in the LHS and the RHS. Moreover, the rewriting $p \overset{\alpha,n}{\Longrightarrow} p'$ creates a new element in the interface, which is merged with the current element*

in p, making p and p′ isomorphic. The rewriting $p \overset{\beta,m}{\Longrightarrow} p''$ *removes the node in the interface of p. However, when applying* β *over p′, we end deleting both the element created by the previous rewriting and the element in the original rule, since they have been merged. When applying* α *over p″, we end up creating a new node in the interface, since the first* β *rewriting only deleted the original interface node. Both rewritings* $p \overset{\alpha,n}{\Longrightarrow} p'$ *and* $p \overset{\beta,m}{\Longrightarrow} p''$ *would be independent if we were performing conventional span rewriting, but they are actually dependent considering the notion of correction, due to the merging effect of the unit* μ*. This means that the test for independence for second-order rewriting with correction must take into account not only information from the matches, but also elements in the interface that connected with matched elements in the LHS and RHS of the rule.*

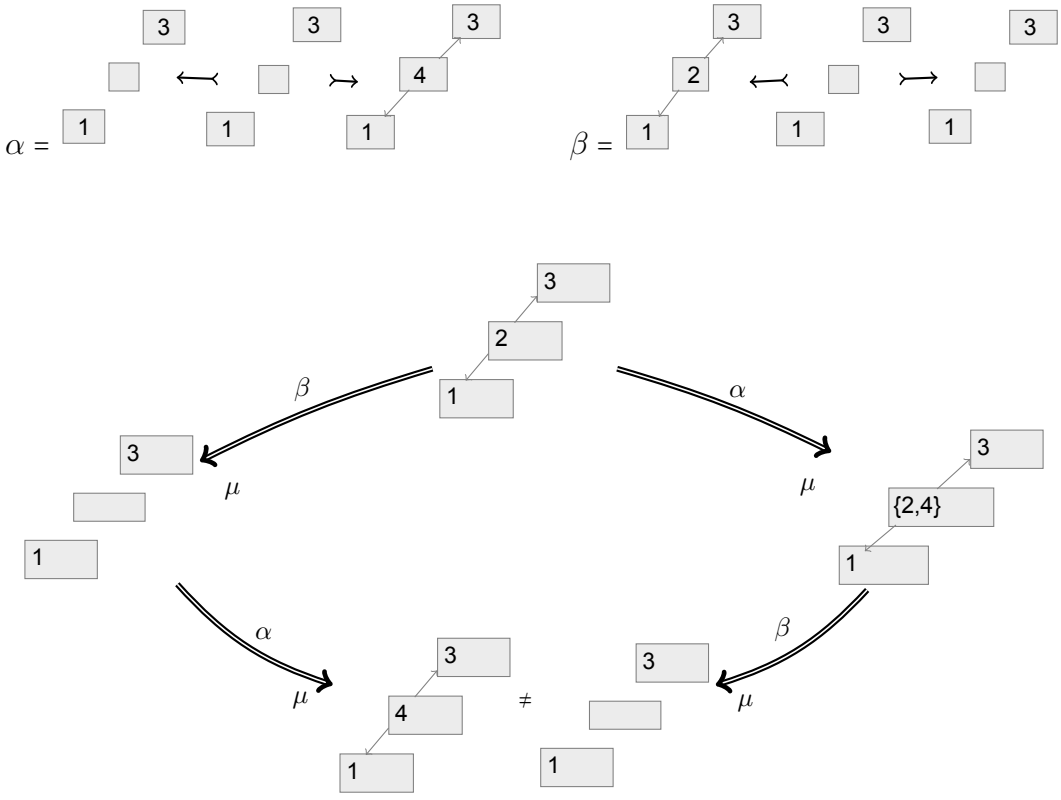


Figure 4.11: Merging of elements affecting local confluence of rewritings with correction.

Both examples show in detail some issues raised by this notion of correction. One is the fact that we are required to deal with *merging* of nodes and edges, which enables to obtain 2-rules with a non-null effect (i.e. that creates new elements) entailing rewritings that do not modify at all the structure of the target graph rule. Another consequence refers to the fact that we need to look beyond the matches in order to test for parallel independence, since the context may entail merging, and this may create additional dependencies. Those shortcomings, summed up to the fact of dealing with a non-standard notion of DPO rewriting, pose serious difficulties to the adoption of this alternative. Its advantage would be to require minimum modification in rules to provide a safe notion of rewriting. However, the fact that we are required to probe for the context beyond the matches to characterize parallel dependencies minimizes such advantages, and points strongly to the next option: avoidance of problematic rewritings.

4.4 Rule rewriting avoiding rule invalidation

Given the issues with the previous approach of correcting the result of rewritings, in this section we analyse a more standard approach to second-order rewriting, which consists of identifying and marking as invalid the matches which may lead to non-preservation of rules. In principle, this effect may be obtained by means of a mechanism already commonly studied and used in the graph rewriting area: negative application conditions.

To realise this idea, one requires that we have a complete description of all possible ways in which rule rewritings may create non-rule spans. Then, we may employ NACs to avoid such situations, allowing only rule-preserving rewritings to occur. Given that the theory of graph rewriting with NACs has been recently generalized for adhesive HLR systems in (LAMBERS et al., 2008), we can make use of it and obtain a precise notion of conflicts and dependencies. This requires, however, that we verify if T-**Span** satisfies the requirements of the framework.

4.4.1 Rule invalidation in DPO span rewriting

This subsection focus on finding the precise conditions that generate a non-rule in the rightmost part of a DPO diagram. That is exactly what the next theorem refers to, focusing on diagrams that may appear in the rightmost part of a double-pushout construction in T-**Span**. First, let us fix an useful auxiliary definition.

Definition 69 (Orphan element)**.** *Let $f : K \to X$ be a typed graph morphism. We say a graph element x (node or edge) is orphan along f, namely $\mathcal{O}_f(x)$, iff $x \in X \wedge x \notin f(K)$ i.e., there is not a pre-image element $y \in K$ such that $f(y) = x$. We denote that x is not orphan along f by $\overline{\mathcal{O}_f(x)}$ or $\neg\mathcal{O}_f(x)$.*

The notion of an orphan element is used to refer to non-preserved elements from a graph rule, i.e. both deleted and created ones. This notation is useful in the following theorem.

Theorem 70 (Conditions for rule invalidation)**.** *Let $d = p_1 \overset{f}{\leftarrow} p_2 \overset{g}{\nrightarrow} p_3$ be a diagram in T-**Span** where p_1, p_2, p_3 are graph rules and g is a monomorphism. The object p_4 of the pushout $PO(d) = p_1 \to p_4 \leftarrow p_3$ is a not a graph rule iff*

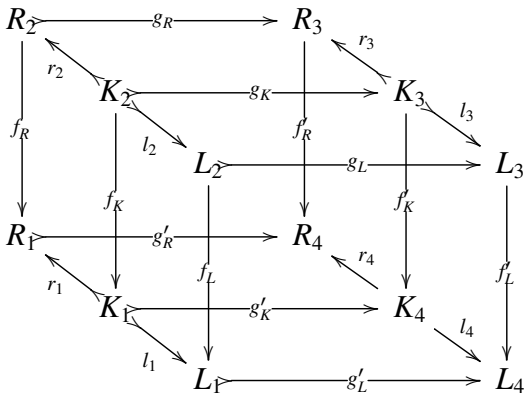
1. *there is $x \in L_2$ such that $\mathcal{O}_{l_2}(x)$, $\overline{\mathcal{O}_{l_3}(g_L(x))}$ and*
   (a) *$\overline{\mathcal{O}_{l_1}(f_L(x))}$, or*
   (b) *there is $y \in L_2$ such that $x \neq y$, $f_L(y) = f_L(x)$ and $\overline{\mathcal{O}_{l_3}(g_L(y))}$*

   or

2. *there is $x \in R_2$ such that $\mathcal{O}_{r_2}(x)$, $\overline{\mathcal{O}_{r_3}(g_R(x))}$ and*
   (a) *$\overline{\mathcal{O}_{r_1}(f_R(x))}$, or*
   (b) *there is $y \in R_2$ such that $x \neq y$, $f_R(y) = f_R(x)$ and $\overline{\mathcal{O}_{r_3}(g_R(y))}$*

Notice that the definition makes reference to morphisms of the diagram representation in

T-**Graph** *of PO*(d)*, as shown below where* $p_i = L_i \overset{l_i}{\leftarrow} K_i \overset{r_i}{\rightarrow} R_i$.



Proof. $(\Rightarrow)$ is given by the following argument:

1. the fact that p_4 is not a graph rule means that there are two distinct elements $m, n \in K$ such that $l_4(m) = l_4(n) = a$ or $r_4(m) = r_4(n) = b$. Those two cases give raise for conditions 1 and 2 in the theorem, respectively. We present now the reasoning for case 1, which consider $l_i : K_i \to L_i,\ i \in \{1, 2, 3, 4\}$ pointing that case 2 is symmetric for $r_i : K_i \to R_i,\ i \in \{1, 2, 3, 4\}$.

2. knowing that K_4 is the pushout object of (f_K, g_K), it can be divided into three disjoint regions:

$$\begin{aligned} I &= \{x \mid x \in img(g'_K) \wedge x \notin img(f'_K)\} \\ II &= \{x \mid x \in img(g'_K) \wedge x \in img(f'_K)\} \\ III &= \{x \mid x \notin img(g'_K) \wedge x \in img(f'_K)\} \end{aligned}$$

   According to the placement of m and n into regions *I*, *II* and *III* we obtain distinct cases.

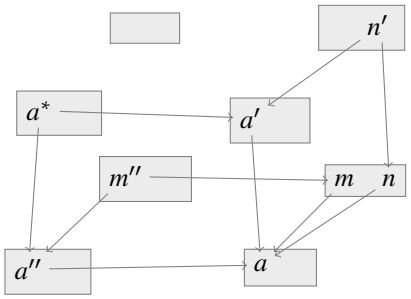
3. one of $\{m, n\}$ must be necessarily in area *III* of K_4. The proof for this is given by assuming (for the sake of contradiction) both $m, n \in g'_K(K_1)$. Then, we have that $g'_K; l_4$ is not mono because $l_4(m) = l_4(n)$. However, the span morphism g' is mono because pushouts preserve monic arrows in *T*-**Span**, and therefore $l_1; g'_L$ is mono by composition. Since $g'_K; l_4 = l_1; g'_L$ by commutativity of the bottom face, we have a contradiction. This means one of the elements, let us say n, must not be in $img(g'_K)$, remaining only to be in region III. From this, we assure the existence of $n' \in K_3$ such that $f'_K(n') = n$, and also $a' \in L_3$ such that $l_3(n') = a'$ and $f'_L(a') = a$.

4. given that n is known to be located in III, there are essentially three distinct cases to consider, corresponding to the possible placement of m in *I*, *II* or *III*. We will show that case $m \in I$ corresponds to clause $1.a$ in Theorem 70, and cases $m \in II$ and $m \in III$, to clause $1.b$.

5. case $m \in I$

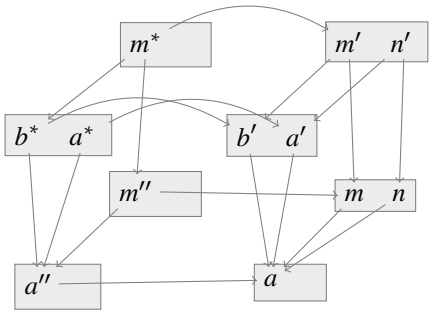
   - There is $m'' \in K_1$ and $a'' \in L_1$ such that $g'_K(m'') = m$ and $l_1(m'') = a''$. Since a'' and a' converge to a and L_4 is a pushout object, then there is a common pre-image $a^* \in L_2$ such that $g_L(a^*) = a'$ and $f_L(a^*) = a''$.

- There is no element $o \in K_2$ such that $l_2 = a^*$. Suppose there is o for contradiction. This would imply $g_L \circ l_2(o) = a'$, and since l_3 is mono and $l_3(n') = a'$, then consequently $g_K(o) = n'$. By the same kind of reasoning, we would have $f_K(o) = m''$. However, this way $f_K; g'_3(o) \neq g_K; f'_K(o)$, which contradicts commutativity $f_K; g'_3 = g_K; f'_K$.
- By instantiating $x = a^*$, we have condition 1.*a*.



6. case $m \in II$

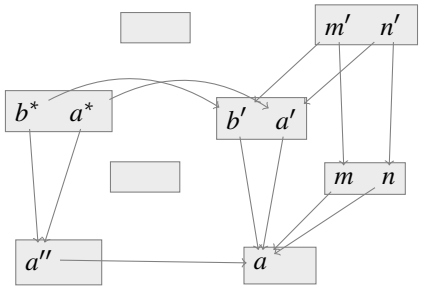
- In this case, we have two pre-images for m, $m' \in K_3$ such that $f'_K(m') = m$ and $m'' \in K_1$ such that $g'_K(m'') = m$. Since K_4 is a pushout object, then there is a common pre-image $m^* \in K_2$ such that $f_K(m^*) = m''$ and $g_K(m^*) = m'$. Because l_3 is mono, there is $b' \in L_3$ such that $b' \neq a'$ and $l_3(m') = b'$. By commutativity of the right face, $f'_L(b') = f'_L(a') = a$. Because L_4 is a pushout object, $f'_L(b') = f'_L(a')$ and g_L is mono, it only remains that both $b', a' \in img(g_K)$, there are $b^*, a^* \in L_1$ such that $b^* \neq a^*$, $f_L(b^*) = f_L(a^*) = a''$ and $g'_K(a'') = a$. By commutativity of the top face, $l_2(m^*) = b^*$.
- There is no element $o \in K_2$ such that $l_2 = a^*$. Assuming (for contradiction) there was o, then by commutativity of the top face and the fact that l_3 is mono, $g_K(o) = n'$. By commutativity of the left face and the fact that l_1 is mono, $f_K(o) = m''$. However, this would imply non-commutativity of the back face on the element o.
- By instantiating $x = a^*$ and $y = b^*$, we have condition 1.*b*.



7. case $m \in III$

- We assure the existence of m', b', b^*, a^* and a'' as in the previous case. However, in this case there are no pre-images for both m and n along g'_K.

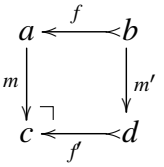
- There is no pre-image for a^* along l_2. Assuming (for contradiction) there was $o \in K_2$ such that $l_2(o) = a*$. By commutativity of the top face and the fact that l_3 is mono, we would have $g_K(o) = n'$. However, since there is no pre-image for n along g'_K, there is no element to map o in K_1 maintaining commutativity of the back face, and we reach a contradiction. The same argument shows that there is no pre-image for b^* along l_2.
- By instantiating $x = a^*$ and $y = b^*$, we have condition $1.b$.



$(\Leftarrow)$ is given by simply instantiating x and y, calculating element-wise the respective pushouts and verifying the injectivity of l_4 or r_4. The possible situations correspond to the diagrams of each possible case in the $(\Rightarrow)$ part of the proof. □

There are essentially two possibilities for making p_4 a non-monic span, corresponding to the subcases (a) and (b) of Theorem 70. Sub-condition (a) occurs when we create a new pre-image in K for an element x in the LHS (or RHS) while there is already a pre-image for it in K that is out of the match. This is the case, for instance, of the pushout depicted in Figure 4.5. Sub-condition (b) refers to the case where, by means of non-injectivity in f, we identify x (for which we are creating a pre-image in K) with another element y for which we create or preserve its pre-image in K. This is what happens in the rightmost part of the DPO diagram depicted in Figure 4.4. The importance of Theorem 70 is that it provides the confidence of being able to track all cases in which we may have rule invalidation through DPO span rewriting. For instance, consider 2-rule $\alpha = a \nleftarrow b \nrightarrow c$, graph rule p and match $m : a \to p$. As known from Proposition 48, we have that the calculation of pushout complement $p \leftarrow p' \leftarrow b$ of a base diagram $p \leftarrow a \nleftarrow b$, in some sense *preserve graph rules* because p' is known to be a rule. Now, Theorem 70 presents all the possible ways that a diagram $p' \leftarrow b \nrightarrow c$ will have a pushout $p' \to p'' \leftarrow c$ such that p'' is not a graph rule. It only remains to characterize the occurrence of the problematic cases on the leftmost part of a DPO diagram, i.e., regarding the match $a \to p$ and the 2-rule morphism $b \to a$.

Theorem 71. *Let $c \overset{m}{\leftarrow} a \overset{f}{\nleftarrow} b$ be a diagram in T-***Span** *such that a,b and c are graph rules, and there is a pushout complement $c \overset{f'}{\nleftarrow} d \overset{m'}{\leftarrow} b$ as shown below.*



Then the following holds:

1. *If there is* $x \in LHS(b)$ *such that* $\mathcal{O}_{l_b}(x)$*, then* $\neg\mathcal{O}_{l_d}(m'(x)) \Leftrightarrow \neg\mathcal{O}_{l_c}(m \circ f(x)) \wedge \mathcal{O}_{l_a}(f(x))$*;*

2. *If there are elements* $x, y \in LHS(b)$ *such that* $x \neq y$ *and* $m'(x) = m'(y)$*, then there are elements* $f(x) \neq f(y)$ *such that* $m \circ f(x) = m_L \circ f(y)$*.*

Proof.

1. $(\Rightarrow)$

   - Let $y \in K_d$ be the element such that $l_d(y) = m'(x)$. Then, there is an element $f'(y) \in K_c$ such that, by commutativity of the bottom square in T-**Graph**, $l_c(f'(y)) = f' \circ m'(x)$. Thus, $\neg\mathcal{O}_{l_c}(m \circ f(x))$.
   - Suppose for contradiction that $\neg\mathcal{O}_{l_a}(f(x))$. Let us call $z \in K_a$ the pre-image of $f(x)$ along l_a, and y as in the previous item. By commutativity of the base and left square, then $l_c \circ m(z) = l_c \circ f'(y)$. By the fact that l_c is injective, $m(z) = f'(y)$. Neither $y \in K_d$ and $z \in K_a$ have a common pre-image in K_b, however $m(z) = f'(y)$, which contradicts the fact that we have a pushout square along K_a, K_b, K_c, K_d. Hence, $\mathcal{O}_{l_a}(f(x))$.

   $(\Leftarrow)$ assume for contradiction that $\mathcal{O}_{l_d}(m'(x))$. Let us call $w \in K_c$ the pre-image such that $l_c(w) = m \circ f(x)$. Because of commutativity of bottom and left faces and the fact that $\mathcal{O}_{l_a}(f(x))$ and $\mathcal{O}_{l_d}(m'(x))$, there are no pre-images for w along m and f'. However, this contradicts the fact we have a pushout square along K_a, K_b, K_c, K_d. Hence, $\neg\mathcal{O}_{l_d}(m'(x))$.

2. $f_L(x) \neq f_L(y)$ is given by injectivity of f, and $m_L \circ f_L(x) = m_L \circ f_L(y)$ by commutativity of $m \circ f = f' \circ m'$ and injectivity of f'.

□

Theorem 71 tells us how to test in the leftmost part of a DPO diagram conditions for rule invalidation. Condition 1 tells us that if we obtain $m'_L(x)$ with a pre-image along l_d, then we know for sure there is no pre-image along l_a for $f_L(x)$. By taking f to the be leftmost part of a 2-rule, then we know the possible elements that would trigger condition 1.a in Theorem 70 are the ones for which we do not delete the pre-image in the 2-rule. Condition 2 shows us that we can track problematic identifications of elements, referring to condition 1.b in Theorem 70, directly in the match m.

4.4.2 Span rewriting with negative application conditions

The previous section gives us tools to track rule invalidation to the structure of the 2-rule and the match.This section extends the concept of negative application conditions from the category T-**Graph** to T-**Span**. There are essentially no significant changes in the characterization, and we finish by proving that T-**Span** fits the framework of adhesive HLR systems with NACs.

Definition 72 (2-rules with negative application conditions)**.** *A negative application condition (NAC) for a 2-rule* $\alpha = a \nleftarrow b \nrightarrow c$ *is a span morphism* $\eta : a \to n$*. A 2-rule with NACs is a pair* $\alpha_N = (N, \alpha)$ *where* α *is a 2-rule, and N is a set of NACs for* α*.*

Definition 73 (Span rewriting with negative application condition)**.** *Consider $\alpha_N = (N, \alpha)$ a 2-rule with NACs such that $\alpha = a \nleftarrow b \nrightarrow c$, and p a graph rule. We say a match $m : a \to p$ respects NAC $\eta : a \to n$, denoted $m \vDash \eta$, iff there is not a monomorphism $e : n \nrightarrow p$ such that $\eta; e = m$. If a match m respect all NACs in set N, we write $m \vDash N$. A span rewriting with NACs $p \overset{\alpha_N, m}{\Longrightarrow} p'$ is a span rewriting $p \overset{\alpha, m}{\Longrightarrow} p'$ such that $m \vDash N$.*

Now, we inquire about the usual DPO properties in the setting of second-order graph rewriting. For such, we will verify if *T*-**Span** satifies all the requirements to be considered a adhesive HLR system with NACs. If this holds, as a consequence we obtain a series of results including a precise notion of conflicts and dependencies. First, we recall the requirements for negative application conditions in adhesive HLR categories, as presented in (LAMBERS et al., 2008).

Definition 74 (adhesive HLR category with NACs)**.** *An* adhesive HLR category with NAC *$(\mathscr{C}, \mathscr{M}, \mathscr{M}', \mathscr{E}', \mathscr{Q})$ is an adhesive HLR category $(\mathscr{C}, \mathscr{M})$ together with two classes of monomorphisms $\mathscr{M}'$, $\mathscr{Q}$, and a class of pair of morphisms $\mathscr{E}'$, with the following properties:*

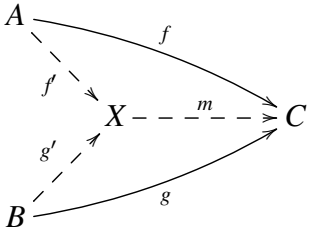
1. *unique $\mathscr{E}'$-$\mathscr{M}'$ factorization;*
2. *epi-$\mathscr{M}$ factorization;*
3. *$\mathscr{M}$-$\mathscr{M}'$ PO-PB decomposition property;*
4. *$\mathscr{M}$-$\mathscr{Q}$ PO-PB decomposition property;*
5. *initial PO over $\mathscr{M}'$-morphisms;*
6. *$\mathscr{M}'$ is closed under PO's and PB's along $\mathscr{M}$-morphisms;*
7. *$\mathscr{Q}$ is closed under PO's and PB's along $\mathscr{M}$-morphisms;*
8. *induced PB-PO property for $\mathscr{M}$ and $\mathscr{Q}$;*
9. *if $f : A \to B \in \mathscr{Q}$ and $g : B \to C \in \mathscr{M}'$ then $g \circ f \in \mathscr{Q}$;*
10. *$g \circ f \in \mathscr{Q}$ and $g \in \mathscr{M}'$ then $f \in \mathscr{Q}$;*
11. *$\mathscr{Q}$ is closed under composition and decomposition.*

For more details on the requirements, please see (LAMBERS et al., 2008)

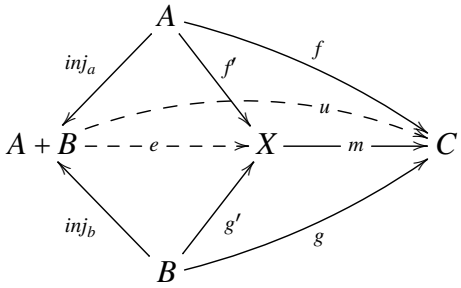
Those numerous requirements have a role in individual proofs for DPO-related properties of rewritings, and they were presented in such way to to keep the maximum generality. Intuitively, the class $\mathscr{Q}$ refers to the factorization monomorphisms that must not factor a given *NAC* in order for a rule to be applied. The classes $\mathscr{E}'$ and $\mathscr{M}'$ refer to the $\mathscr{E}'$-$\mathscr{M}'$ factorization, as follows.

Definition 75. *Let $\mathscr{C}$ be a category with a class $\mathscr{E}'$ of pairs of morphisms with the same target, and a class $\mathscr{M}'$ of morphisms. We say $\mathscr{C}$ has $\mathscr{E}'$-$\mathscr{M}'$ factorization iff for each pair*

of morphisms $A \xrightarrow{f} B \xleftarrow{g} C$ *there is an object* X *and morphisms* $(f' : A \to X, g' : B \to X) \in \mathscr{E}'$ *and* $(m : X \to C) \in \mathscr{M}'$ *such that* $f'; m = f$ *and* $g'; m = g$*, as shown below.*



It is known that if a category $\mathscr{C}$ has coproducts and epi-mono factorization, we have $\mathscr{E}'$-$\mathscr{M}'$ factorization where $\mathscr{E}'$ is the class of all jointly surjective pair of morphisms, and $\mathscr{M}'$ the class of all monomorphisms. Given the pair $(f : A \to C, g : B \to C)$, we calculate the coproduct $(A+B, inj_A, inj_B)$, obtain a unique arrow $u : A+B \to C$ and also the factorization $EM(u) = e; m$. Finally, we set $f' = inj_A; e$ and $g' = inj_B; e$, as shown below. The morphisms f' and g' are jointly surjective because e is an epimorphism.



Proposition 76. *The tuple* $(T\text{-}\mathbf{Span}, M, M, E, M)$ *is an adhesive HLR category with NACs, where* M *is the class of all monomorphisms in* T-**Span** *and* E *is the class of all jointly surjective pair of morphisms in* T-**Span**.

Proof. T-**Graph** meets all requirements if we take $\mathscr{M} = \mathscr{M}' = \mathscr{Q}$ as the class of all monomorphisms, and $\mathscr{E}'$ as the class of all jointly surjective graph morphisms (EHRIG et al., 2005). Because T-**Span** is built from T-**Graph** component-wise, we obtain that the conditions from Definition 74 hold by construction. Conditions 1-2 derive from T-**Span** having epi-mono factorization and coproducts and, consequently, E-M factorization. Conditions 3-8 results from $M = \mathscr{M} = \mathscr{M}' = \mathscr{Q}$ and the fact that $(T\text{-}\mathbf{Span}, M)$ is an adhesive HLR category. Conditions 9-11 comes from composition and decomposition of monomorphisms in general categories, since $M = \mathscr{M}' = \mathscr{Q}$. □

4.4.3 Rule preservation by means of span rewriting with NACs

Provided that T-**Span** fits into the adhesive HLR category with NACs framework, we ask the following: given a 2-rule α, is there a set of NACs for it such that it forbids all rewritings that invalidate rules, and, at the same time, allows all the other rewritings. If this set exists, we call it a *minimal rule-preservation set of NACs*. Formally,

Definition 77 (minimal safety NACs)**.** *Let* α *be a 2-rule. We say that a set* N *of NACs for* α *is a* minimal rule-preservation set *iff, for any rule* p *and match* m*,*

- *if* $p \overset{\alpha,m}{\Longrightarrow} p'$ *and* p' *is not a rule, then* $m \not\models N$*, and*
- *if* $p \overset{\alpha,m}{\Longrightarrow} p'$ *and* p' *is a rule, then* $m \models N$

The set N will also be referred as the collection of minimal safety NACs for α.

The existence of a minimal rule-preservation set for a 2-rules would allow a direct characterization of second-order graph rewriting using the mechanism of span rewriting with NACs. Given any 2-rule, we state that there is an algorithm which makes possible to calculate a minimal rule-preservation set for it, provided the graph components of the rule are finite. The intuition is basically to pinpoint all potentially problematic elements in the LHS and RHS, as pointed out by Theorem 70.

Definition 78 (Calculated minimal rule-preservation set)**.** *The algorithm shown in Figure 4.12 calculates a minimal rule-preserving set of NACs for a finite 2-rule given as input. We denote the minimal rule-preserving set calculated from α by $\mathscr{S}(\alpha)$.*

Example 79 (Minimal safety NACs)**.** *In Figure 4.13, we have a 2-rule together with its set of calculated minimal safety NACs from the algorithm of Definition 78. The NACs η_1 and η_2 avoid the creation of redundant pre-images if elements 3 and 4 are matched against elements already containing pre-images. The NACs η_3, η_4, η_5 and η_6 make sure that there will be no merging of elements in the match resulting in rule invalidation.*

Theorem 80. *$\mathscr{S}(\alpha)$ is a minimal rule-preserving set of NACs for a finite input 2-rule $\alpha = a \leftarrow b \rightarrow c$.*

Proof. Consider a span rewriting $p \overset{\alpha,m}{\Longrightarrow} p''$, where p is a rule, represented by the following DPO diagram in T-**Span**.

$$\begin{array}{ccccc} a & \xleftarrow{f} & b & \xrightarrow{g} & c \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle k} & & \downarrow{\scriptstyle m^*} \\ p & \xleftarrow[f^*]{} & p' & \xrightarrow[g^*]{} & p'' \end{array}$$

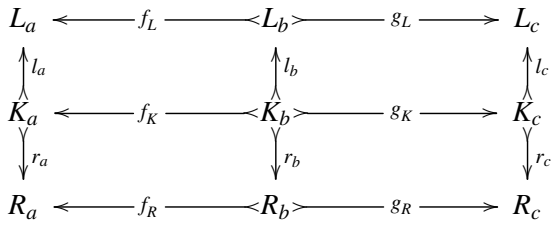
The fact that p'' is not a rule is equivalent to say that one of the following statements is true, according to Theorem 70. In the following, all references to orphan elements consider the morphism from the interface graph of each rule to the LHS or RHS.

1. $\exists x \in LHS(b)$ satisfying ($\mathcal{O}(x)$, $\neg\mathcal{O}(k(x))$ and $\neg\mathcal{O}(g(x))$)
2. $\exists x' \in RHS(b)$ satisfying ($\mathcal{O}(x')$, $\neg\mathcal{O}(k(x'))$ and $\neg\mathcal{O}(g(x'))$)
3. $\exists c, d \in LHS(b)$ satisfying ($c \neq d$, $k(c) = k(d)$, $\mathcal{O}(c)$, $\neg\mathcal{O}(g(c))$ and $\neg\mathcal{O}(g(d))$)
4. $\exists c', d' \in RHS(b)$ satisfying ($c' \neq d'$, $k(c') = k(d')$, $\mathcal{O}(c')$, $\neg\mathcal{O}(g(c'))$ and $\neg\mathcal{O}(g(d'))$)

Theorem 71 applies to the leftmost part of the DPO diagram since p, p', a, b are rules, and f is monic. The objective is to substitute the parts of the expression which depend on k. From subcondition (a), we obtain the statements 1 and 2, and from subcondition (b), the statements 3 and 4.

1. $\exists f(x) \in LHS(a)$ satisfying ($\mathcal{O}(x)$, $\mathcal{O}(f(x))$, $\neg\mathcal{O}(m \circ f(x))$ and $\neg\mathcal{O}(g(x))$)
2. $\exists f(x') \in RHS(a)$ satisfying ($\mathcal{O}(x')$, $\mathcal{O}(f(x'))$, $\neg\mathcal{O}(m \circ f(x'))$ and $\neg\mathcal{O}(g(x'))$)
3. $\exists f(c), f(d) \in LHS(a)$ satisfying ($c \neq d$, $m \circ f(c) = m \circ f(d)$, $\mathcal{O}(c)$, $\neg\mathcal{O}(g(c))$ and $\neg\mathcal{O}(g(d))$)
4. $\exists f(c'), f(d') \in RHS(a)$ satisfying ($c' \neq d'$, $m \circ f(c') = m \circ f(d')$, $\mathcal{O}(c')$, $\neg\mathcal{O}(g(c'))$ and $\neg\mathcal{O}(g(d'))$)

Input : A finite 2-rule $\alpha = a \xleftarrow{f} b \xrightarrow{g} c$, represented by the following diagram in T-**Graph**.



Output: A set $\mathscr{S}(\alpha)$ of NACs for α.

begin

`// Initialize N with the empty set`

$N := \{\}$;

`// Insert NACs to avoid condition (a) in Thm 70`

$ProbL := \{f_L(x) \mid x \in L_b \ \wedge \ \mathscr{O}_{l_a}(f_L(x)) \ \wedge \ \mathscr{O}_{l_b}(x) \ \wedge \ \neg\mathscr{O}_{l_c}(g_L(x))\}$;

$ProbR := \{f_R(x) \mid x \in R_b \ \wedge \ \mathscr{O}_{r_a}(f_R(x)) \ \wedge \ \mathscr{O}_{r_b}(x) \ \wedge \ \neg\mathscr{O}_{r_c}(g_R(x))\}$;

for $x \in ProbL$ **do**

  $a' := L_a \xleftarrow{l_a[x' \mapsto x]} K_a \uplus \{x'\} \xrightarrow{r_a[x' \mapsto x'']} R_a \uplus \{x''\}$;

  $N := N \cup \{a \hookrightarrow a'\}$;

for $x \in ProbR$ **do**

  $a' := L_a \uplus \{x''\} \xleftarrow{l_a[x' \mapsto x'']} K_a \uplus \{x'\} \xrightarrow{r_a[x' \mapsto x]} R_a$;

  $N := N \cup \{a \hookrightarrow a'\}$;

`// Insert NACs to avoid condition (b) in Thm 70`

$PairL := \{(f_L(x), f_L(y)) \mid x, y \in L_b \ \wedge \ x \neq y \ \wedge \ \mathscr{O}_{l_b}(x) \ \wedge \ \neg\mathscr{O}_{l_c}(g_L(x)) \ \wedge \ \neg\mathscr{O}_{l_c}(g_L(y))\}$;

$PairR := \{(f_R(x), f_R(y)) \mid x, y \in R_b \ \wedge \ x \neq y \ \wedge \ \mathscr{O}_{r_b}(x) \ \wedge \ \neg\mathscr{O}_{r_c}(g_R(x)) \ \wedge \ \neg\mathscr{O}_{r_c}(g_R(y))\}$;

$Epis := calculateAllPartitions(a)$;

for $e \in Epis$ **do**

  if *($e_L(a) = e_L(b)$ for some $(a, b) \in PairL$) OR*

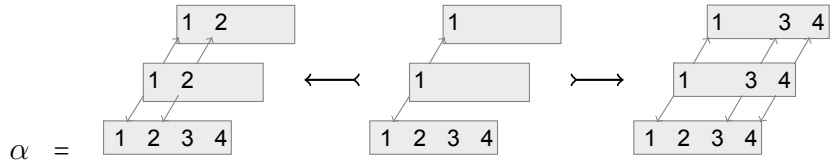
  ($e_R(x) = e_R(y)$ for some $(x, y) \in PairR$) **then**

    $N := N \cup \{e\}$;

`// Return the final set of NACs`

return N;

Figure 4.12: Algorithm for calculating $\mathscr{S}(\alpha)$.



$\mathscr{S}(\alpha) =$

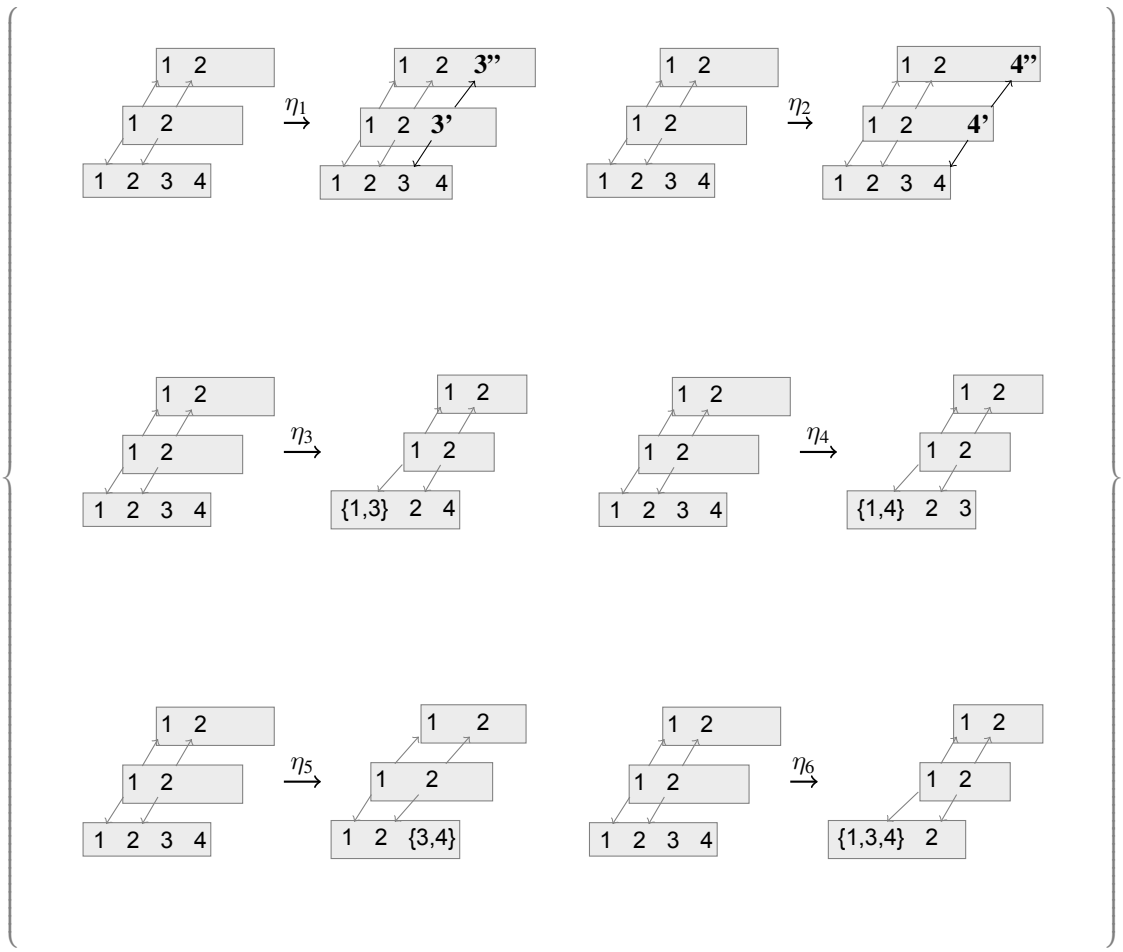


Figure 4.13: Minimal safety NACs calculated from 2-rule α.

Notice that those existential conditions depend on on the match m and the 2-rule structure. Since the 2-rule structure is fixed, what the algorithm does is to create a NAC for each problematic situation in order to invalidate the matches that trigger one of those conditions. Line 2 initializes an empty set of NACs. Lines 3-4 initialize the elements in the LHS or RHS that may trigger subcondition (a) in Theorem 70. For each potentially problematic element x, the for-loops in lines 5-10 add an individual NAC to avoid the situation $\neg\mathcal{O}(m \circ f(x))$. Lines 11-12 calculate the potentially problematic pairs of elements that may trigger subcondition (b) in Theorem 70. Line 13 calculates all the partitions of the LHS, representing all possible ways of identifying elements in the rule a. For each partition e of a, lines 14-17 test if it contains a problematic identification. If it does, the partition is included as a NAC to avoid situation $m \circ f(c') = m \circ f(d')$.

Hence, for the rewriting $p \overset{\alpha,m}{\Longrightarrow} p''$:

- if p'' is not a rule, one of the NACs introduced by $\mathscr{S}(\alpha)$ will be satisfied.
- if p'' is a rule, then none of the NACs introduced by $\mathscr{S}(\alpha)$ will be satisfied.

□

Now we can define second-order rewriting avoiding rule invalidation through span rewriting with NACs. Given a 2-rule with NACs, we augment its set of NACs with the calculated minimum rule-preserving set, and consider as valid second-order rewritings all direct derivations whose matches respect the extended NAC collection.

Definition 81. *Given a 2-rule with NACs $A = (N, \alpha)$ where $\alpha = a \nleftarrow b \nrightarrow c$, a graph rule p and a match $m : a \to p$, we say there is a* rule rewriting avoiding rule invalidation $p \overset{A,m}{\Longrightarrow}_2 p'$ *iff there is a span rewriting with NACs $p \overset{B,m}{\Longrightarrow} p'$ where $B = (N \cup \mathscr{S}(\alpha), \alpha)$*

We defined the notion of rewriting based on 2-rules with NACs because they subsume conventional 2-rules: whenever the set of NACs is empty, we have conventional rule rewriting. Moreover, NACs are usually required for other purposes rather than exclusively assuring rule preservation, as the next section will show.

4.5 Second-order rewriting

The previous sections discussed and presented results concerning several ways of arriving at a notion of second-order rewriting for graph transformation rules. This section studies the possible effects of second-order rewriting on graph rules, and also discusses other uses for NACs besides ensuring rule preservation.

From the previous discussing, it is reasonable to conclude that rule rewriting avoiding rule invalidation satisfy good criteria for representing second-order rewriting. The main reason is that, given a characterization of rule-preservation by means of minimal safety NACs, we may employ the generalized version of DPO rewriting with NACs (LAMBERS et al., 2008) almost directly, which allow us to benefit from a plethora of results proved for such context. Moreover, it allows to discuss the notion of higher-order rewriting in other contexts, which eases a possible generalization of our results. From here onwards, we stop discussing alternatives and fix Definition 81 as the standard notion of **second-order rewriting**.

Notation 82. From here onwards, we will drop the subscripted 2 when denoting a second-order rewriting $p \overset{\alpha_N,m}{\Longrightarrow}_2 p'$. To avoid confusion with the notation for conventional DPO span rewriting, the notation $p \overset{\alpha,m}{\Longrightarrow} p'$ will mean second-order rewriting, unless explicitly

marked as a span rewriting. Notice that the difference is that we do not calculate the minimal rule-preservation set of NACs for span rewritings.

The table shown in Figure 4.14 presents a comprehensive description of all possible kinds of modification that are possible to be made over a graph transformation rule by using second-order rewriting. A particular 2-rule may have as effect any combination of the depicted kinds of basic modifications.

Modification	Rewriting action
increase element consumption	add items to L only
decrease element consumption	remove elements from L only
increase element creation	add elements to R only
decrease element creation	remove elements from R only
convert preservation into deletion+creation	remove items from K only
convert deletion+creation into preservation	add items to K only
convert deletion into preservation	add items to $K + R$
convert creation into preservation	add items to $L + K$
increase context	add items to $L + K + R$
decrease context	remove items from $L + K + R$

Figure 4.14: Possible modifications over a rule $L \leftarrow K \rightarrow R$ by second-order rewriting.

One important issue regarding second-order rewriting is to identify when a given rule will be triggered. For example, we may want to define that we should modify a rule if it *deletes* or *creates* elements of a given type, which are characterized in DPO rules by the existence of elements in L or R without a pre-image in K. Span morphisms from the LHS of the 2-rule may identify the existence of elements in L and R, however they cannot test the *absence* of a pre-image for the matched elements. In other words, the left-hand side of second-order rules cannot be defined to match only on deleted elements, but rather on deleted or preserved elements (and similarly for created ones). The following example presents more clearly this issue.

Example 83. *The 2-rule depicted in Figure 4.15 intends to match all rules that create a message. Its effect is to modify the rule by adding a data node to the message instance in the RHS. However, as it is, the rule matches any rule that either* creates a message *or* preserves a message*, since both have a message node in their RHS.*

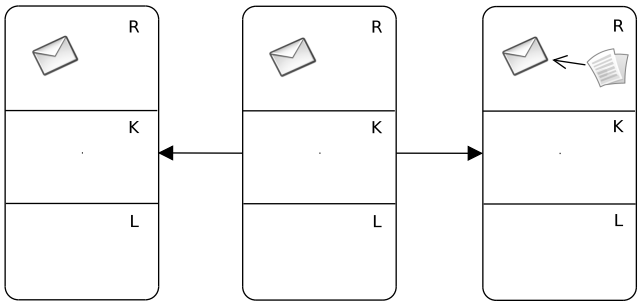


Figure 4.15: 2-rule that matches rules preserving or creating a message.

It is very plausible to wish distinct modifications for rules depending on the fact they create, delete or preserve some elements. The straightforward way to obtain such differentiation is to employ a NAC to avoid matching rules that preserve a message. Figure 4.16

shows the required NAC that provides the correct behavior for the rule in Figure 4.15. Notice that differentiation for 2-rules to match a rule which preserves a given element is not required, since they naturally do not have matches for rules which delete the same element.

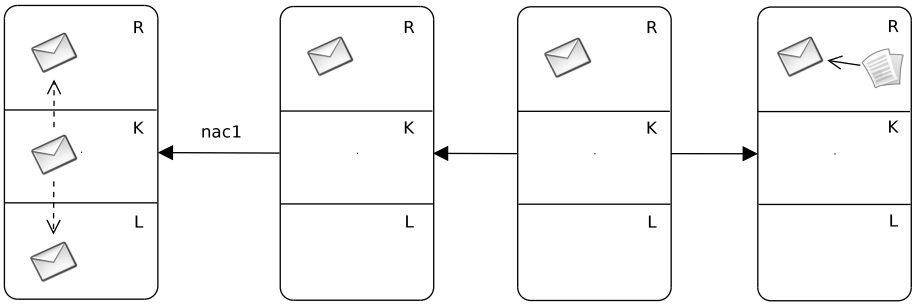


Figure 4.16: Negative application condition for affecting only rules creating message.

Besides ensuring preservation of rule structure and allowing a more refined matching process, NACs are also useful to assure termination condition for rewritings. One of the simplest examples occurs when the modeler desires to have a non-deleting 2-rule acting only once over a given rule. For instance, the 2-rule of Figure 4.15 creates elements without deleting anything, and potentially could be applied to the same rule over and over again. Since the most common purpose is to add only one new data node to a message, the modeler can express this by equipping the rule with the NAC shown in Figure 4.17. Notice that this situation is particularly common in the context of model transformations, where we are usually interested in the result of the transformation process rather than the execution of the transformation, and thus it is important that the rewriting process is terminating.

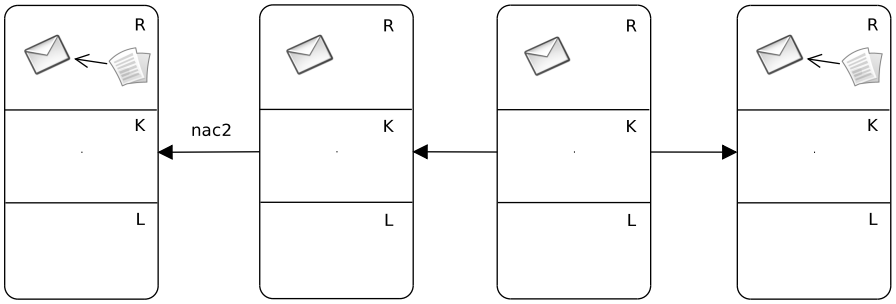


Figure 4.17: Negative application condition to assure unique application of 2-rule.

From the previous discussion, we have identified that negative application conditions are actually quite important for second-order rewriting, since

1. they may be used to ensure rule preservation;
2. they allow to distinguish between preservation and creation/deletion in matches;
3. they may be used to ensure termination condition for the rewriting of rules.

Notice that, even if we had chosen rule rewriting with correction as the default notion of second-order rewriting, NACs would still be required for reasons 2 and 3. This is another convenience of our choice, since for the sake of rule preservation we employ of a mechanism that would be nevertheless required for other purposes.

4.6 Conflicts and dependencies in second-order rewriting

The characterization of second-order rewriting and the fact that T-**Span** fits the framework of adhesive HLR systems with NACs provides us a precise definition for conflicts and dependencies in second-order rewriting. For such, we instantiate the definitions for parallel and sequential independence, as presented in (LAMBERS et al., 2008) for the context of T-**Span**.

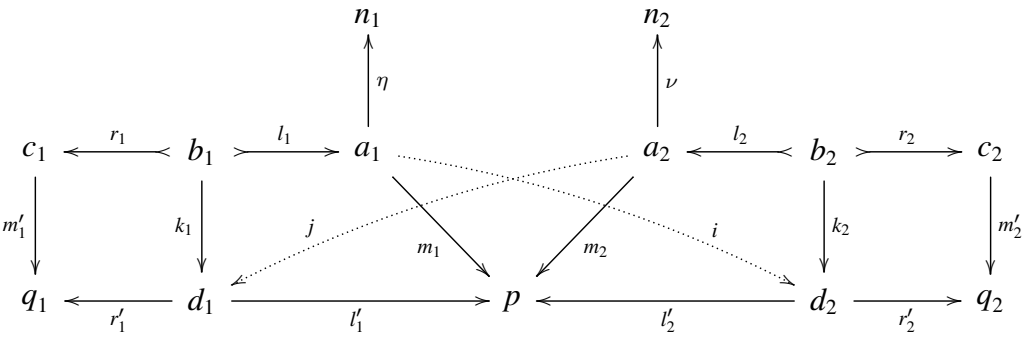
Definition 84 (parallel and sequential independence for span rewriting with NACs)**.** *Let $\alpha_M = (M, \alpha)$ and $\beta_N = (N, \beta)$ be two 2-rules with NACs, where $\alpha = a_1 \overset{l_1}{\nleftarrow} b_1 \overset{r_1}{\nrightarrow} c_1$, $\beta = a_2 \overset{l_2}{\nleftarrow} b_2 \overset{r_2}{\nrightarrow} c_2$, $M = \{\eta_1, \ldots, \eta_k\}$ and $N = \{\nu_1, \ldots, \nu_l\}$. Two span rewritings $p \overset{\alpha_M, m_1}{\Longrightarrow} q_1$ and $p \overset{\beta_N, m_2}{\Longrightarrow} q_2$ with NACs are parallel independent iff*

$$\exists i : a_1 \to d_2 \text{ s.t. } (l'_2 \circ i = m_1 \text{ and } r'_2 \circ i \vDash M)$$

and

$$\exists j : a_2 \to d_1 \text{ s.t. } (l'_1 \circ j = m_2 \text{ and } r'_1 \circ j \vDash N)$$

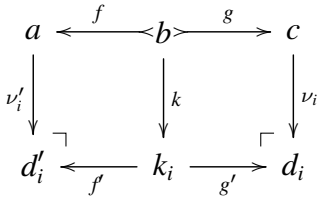
as shown in the following diagram, where $\eta \in M$ and $\nu \in N$:



Negative application conditions are usually defined as morphisms departing from the LHS, which aim is to pose restrictions on the match. However, NACs can be similarly defined over the RHS, posing restrictions on the co-match of a given rewriting. The two ways of defining NACs are equivalent, since we have a process that obtains from a collection of RHS NACs an equivalent collection of LHS NACs restricting the rewriting in the opposite direction. This transformation is required for defining the inverse of a 2-rule with NACs $(M, \alpha)^{-1}$. which, in turn, is required for the definition of sequential dependence.

Definition 85 (LHS NACs from RHS NACs)**.** *Let $\alpha = a \overset{f}{\leftarrow} b \overset{g}{\rightarrow} c$ be a 2-rule, and $N = \{\nu_i\}_{i \in I}$ a collection of RHS NACs over α, i.e., a set of span morphisms $\nu_i : c \to d_i$. We define the collection N^{-1} of LHS NACs over α as the least set for which the following holds:*

if $\nu_i \in N$ and there is a POC for $d_i \overset{\nu_i}{\leftarrow} c \overset{g}{\leftarrow} b$, then $\nu'_i \in N^{-1}$, where ν'_i is the comatch of the span rewriting $d_i \overset{\alpha^{-1}, \nu_i}{\Longrightarrow} d'_i$ as shown in the diagram below.



Definition 86 (Inverted rule with NACs)**.** *Given $\alpha_N = (N, \alpha)$ be a 2-rule with NACs, we define its reverse rule as as $(\alpha_N)^{-1} = (N^{-1}, \alpha^{-1})$*

Definition 87. *Two span rewritings $p \overset{\alpha_M, m_1}{\Longrightarrow} q$ and $q \overset{\beta_N, m_2}{\Longrightarrow} r$ are sequential independent iff $q \overset{\alpha_M^{-1}, m_1'}{\Longrightarrow} p$ and $q \overset{\beta_N, m_2}{\Longrightarrow} r$ are parallel independent.*

Now that we have established a proper notion of conflict and dependency for *span rewriting with NACs*, we can turn to *second-order rewriting* since the latter is based on the former.

Definition 88 (Parallel independent second-order rewritings)**.** *Let (N_1, α) and (N_2, β) be 2-rules with NACs, and $\rho_1 = p \overset{(N_1,\alpha), m_1}{\Longrightarrow} p_1$ and $\rho_2 = p \overset{(N_2,\beta), m_2}{\Longrightarrow} p_2$ be two second-order rewritings. We say that ρ_1 and ρ_2 are parallel independent iff the span rewritings*

$$p \overset{(N_1 \cup \mathcal{S}(\alpha), \alpha), m_1}{\Longrightarrow} p_1$$

and

$$p \overset{(N_2 \cup \mathcal{S}(\beta), \beta), m_2}{\Longrightarrow} p_2$$

are parallel independent. Otherwise, they are said to be conflicting or parallel dependent.

In the case of second-order rewriting, dependencies of kind produce-forbid can be perceived when two rewritings attempt to create a new pre-image to convert a deleted element into a preserved element. This situation can be better visualized in the following example.

Example 89 (Parallel dependency in second-order rewriting)**.** *Figure 4.18 presents a second-order rewriting $p \overset{\alpha, m}{\Longrightarrow} p'$, which converts a deleted element 1 in L into a preserved element by creating a corresponding pre-image 2 in K and an element 3 in R. As a span rewriting, this rewriting does not delete anything, and thus would not be considered conflicting with itself. However, since we have based the definition of second-order rewriting of α as the rewriting with NACs of $(\mathcal{S}(\alpha), \alpha)$, the second-order rewriting $p \overset{\alpha, m}{\Longrightarrow} p'$ is conflicting with itself, since the parallel application of more than one instance of the same rewriting would result in a non-monic span.*

4.7 Summary

In this chapter we took the challenge of creating a notion of rule-based modification of graph rules. The original analysis suggested us to look for DPO constructions in the categories T-**Span** and T-**Rules**. Since neither category was satisfactory for defining the transformation of rules, we had to consider additional mechanisms either to *correct* ill-behaved rewritings, or to *avoid* them. We defined second-order graph rewriting (rule modification) by means of span rewritings (DPO in T-**Span**) that consider an additional collection of *minimal safety NACs*, calculated from the structure of each 2-rule. We also showed that the tuple $(T\text{-}\mathbf{Span}, M, M, E, M)$, where M are all monomorphisms and E all jointly surjective pairs of morphisms, can be characterized as an adhesive HLR category with NACs. This allows us to import several results from DPO graph transformation to DPO rule transformation. Based on this, the calculation of conflicts and dependencies for second-order graph rewriting can be obtained from the respective span rewritings with NACs, as we have shown in the last section of the chapter.

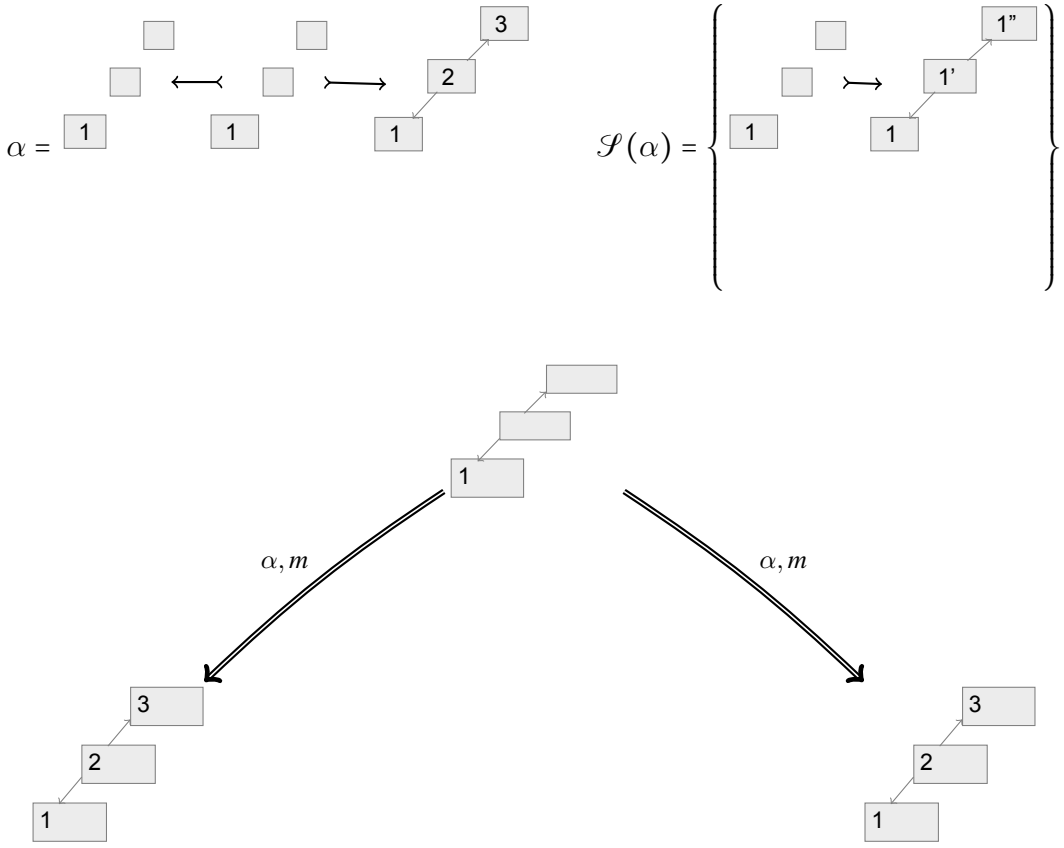


Figure 4.18: Parallel dependent second-order rewritings of 2-rule α.

5 SECOND-ORDER GRAPH GRAMMARS

In Chapter 4 we have arrived at a notion of second-order rewriting that allows to describe transformations in graph rules. Now we discuss how to apply this mechanism to create specifications with both first-order and second-order rewriting rules. In graph transformation systems, it is usual to not have a unique graph rule, but rather a collection of rules affecting the target graph. Therefore, we must extend our notion of second-order rewriting from a single rule to collections of rules. Since we intend to have modifications in particular rules, the notion of state comprises at least two elements: a current graph and also a current collection of graph rules. We start this chapter by addressing how to represent this collection of rules in a convenient way. We identify some issues with conventional sets, and then propose the usage of *coproducts* in T-**Span** as rule collections. Next, based on this notion of coproduct rule collection, we redefine (first-order) graph grammars and derivations, and introduce second-order graph grammars and their derivations. Furthermore, we discuss additional components of a second-order specification, allowing for modifications in the initial graph and type graph. We end up presenting some examples of second-order specifications and their execution. The original contributions of this chapter are:

- discussion regarding structures for defining rulesets, and the introduction of the concept of coproduct collections of rules;
- definition of rewriting for coproduct collections ;
- redefinition of first-order graph grammar using coproduct collection,
- several definitions for second-order graph grammars, differing essentially in how they affect the initial and type graphs;
- the concept of *model-transformation derivation*, characterizing a terminating second-order rewriting process followed by the execution of the new *evolved first-order system*;
- the notion of *evolutionary span* representing the overall modifications of a system that underwent a model transformation.

5.1 Some issues with sets of rules

Whenever we think of collections, one of the first intuitions that arise is the notion of a set, and it would be expected to have a simple (let us say *nameless*) set of rules as part of graph grammar specifications. However, according to Definition 18, the collection of

rules of a graph grammar $\mathcal{G} = (T, G_0, P, \pi)$ is defined by two components: a *set* of rule names P, and a function $\pi : P \to T\text{-}\mathbf{Rules}$ associating rule names to the actual spans. Let us refer to this kind of specification as a *named* set of rules. Even if not clear at first sight, this naming scheme has deeper consequences rather than just providing a convenient way of labelling direct derivations. By allowing the identity of a rule to be associated with a name rather with a concrete span, we allow two rules $p, p' \in P$ to have the exact same behavior and still be considered distinct entities. This is the case whenever we map both names to the exact same span, e.g. $\pi(p) = \pi(p')$, or even to two distinct but isomorphic spans, e.g. $\pi(p) \cong \pi(p')$. In other words, named rule collections allow the modeler to represent multiplicity of behavior, as if we were working with multisets of rules. This approach also provides a simple way of dealing with the issue of *representation non-determinism*, i.e., the fact that a particular transformation may be characterized by several (in fact, infinitely many) concrete graph rules that only differ in the name of nodes and edges in their instance graphs. Through the function π we associate a given name p to a concrete graph rule that acts as a representative of the actual transformation, and we do not need to explicitly refer directly to equivalence classes of rules. Those aspects regarding the adoption of simple sets of spans or named rules are important in their own context, but become even more relevant when we assume the collection of rules may be modified by second-order rewriting. In the following, we will define an update operation (based on second-order rewriting) for both nameless and named versions of rule collections, and then we discuss the advantages and drawbacks of both definitions.

Definition 90 (Nameless ruleset rewriting)**.** *Given a second-order rewriting $r \overset{\alpha,m}{\Longrightarrow} r'$ and a set of graph rules D, we define the* set rewriting operation $\Rrightarrow$ *as follows.*

$$\frac{r \overset{\alpha,m}{\Longrightarrow} r' \qquad r \in D}{D \overset{\alpha,m}{\Rrightarrow} (D - \{r\}) \cup \{r'\}}$$

The nameless ruleset rewriting operation $\Rrightarrow$ extend second-order rewriting in the most obvious way: it removes the rewritten rule from the set and inserts the resulting rule in the collection, whilst maintaining the others rules unchanged. The operation is labelled by the parameters of the direct derivation. Notice that it is not required to specify the element being modified in the label since it can be obtained from the match m. This way of defining a rewrite over a set allows a particular 2-rule to affect one, two, all or none of the rules in D, only depending on the existence of 2-matches for any of the elements of the ruleset. As we introduced earlier, the usage of sets have to some important consequences. For example, consider the nameless ruleset rewriting shown below.

$$\frac{r_3 \overset{\alpha,m}{\Longrightarrow} r_2 \qquad r_3 \in \{r_1, r_2, r_3\}}{\{r_1, r_2, r_3\} \overset{\alpha,m}{\Rrightarrow} \{r_1, r_2\}}$$

Since r_2 is already in $\{r_1, r_2, r_3\} - \{r_3\}$ and sets do not permit repeated elements, the resulting set has one element less than the original one. Due to representation non-determinism, however, the same 2-rule and 2-match (α, m) also gives rise to a rewriting $r_3 \overset{\alpha,m}{\Longrightarrow} r_2'$ such that $r_2 \cong r_2'$ but $r_2 \neq r_2'$. In this case, the resulting set would be $\{r_1, r_2', r_3\}$, and the size of the set would be maintained. We conclude that the size of the resulting set is non-deterministically dependent on the choice of a concrete rule.

Another question is the lack of indexing for rules. Even if we had chosen to make use of *multisets* rather than sets, we still would lack distinction and identity between equal

elements. Since we need to consider that rules which are changing may be applied to graphs in a model with second-order rewriting, it would be important to track individual rules within all its equivalent or equal other ones, in order to properly identify the precise effect of each individual rule in the execution of the model. This kind of issue is solved by the named version of rulesets, i.e. a pair (P, π). We can reproduce the definition for nameless rulesets for the named case, as follows.

Definition 91 (Named ruleset rewriting)**.** *Given a second-order rewriting $p \overset{\alpha,m}{\Longrightarrow} p'$, a set of rule names P and a function $\pi : P \to T$-**Rules**, we define the* named ruleset rewriting operation $\Rrightarrow$ *as follows.*

$$\frac{r \overset{\alpha,m}{\Longrightarrow} r' \qquad p \in P \qquad \pi(p) = r \qquad \pi' = \pi[p \mapsto r']}{(P, \pi) \overset{\alpha,m}{\Rrightarrow} (P, \pi')}$$

In this definition, graph rule names assure that the identity of the rule being modified, which is maintained independently of the way the rule is altered. Essentially, this definition solves the problem of dealing with representation non-determinism and indexing, since rules names act as the identity for the rule rather than the span itself.

Notice, however, that some changes in specification may involve not only modifications in the structure of current rules, but also adding new rules and removing deprecated ones. Given that this is a natural requirement, we can propose addition and removal operations for both nameless and named rulesets, as shown in the next definition.

Definition 92 (Addition and deletion in rule collections)**.** *Below we define addition and removal operations for both nameless and named characterizations of sets of rules.*

Nameless:

$$\frac{r \in D \qquad D' = D - \{r\}}{D \overset{-r}{\Rrightarrow} D'}$$

$$\frac{r \notin D \qquad D' = D \cup \{r\}}{D \overset{+r}{\Rrightarrow} D'}$$

Named:

$$\frac{p \in P \qquad P' = P - \{p\} \qquad dom(\pi') = dom(\pi) - \{p\} \qquad \pi'(x) = \pi(x)}{(P, \pi) \overset{-p}{\Rrightarrow} (P', \pi')}$$

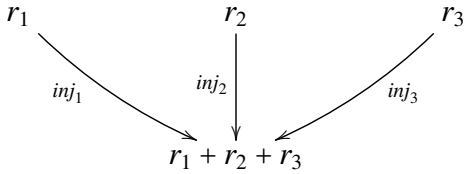
$$\frac{p \notin D \qquad P' = P \cup \{p\} \qquad \pi' = \pi[p \mapsto r]}{(P, \pi) \overset{+(p,r)}{\Rrightarrow} (P', \pi')}$$

Because we work with sets in both nameless and named versions, the addition of a new element depends on the element being already in the set or not. Hence, it is not possible to define a generic addition operation that will always add a new rule to a given ruleset, independently on the rules already in the set. In the named version, the problem is not in the rule specification itself, but rather in how to obtain a fresh name for each addition. Another aspect of this definition is that it suggests structural distinctions between addition, deletion and update (through rewriting). It seems that, for the sake of evolution, you are required to employ one mechanism for addition (set union), deletion (set difference) and modification (set difference, set union and second-order rewriting). Although it is not a problem per-se, we could argue that it would be more elegant if we could represent all possible modifications over a set of rules through a unique kind of operation, for which addition, removal and modification were special cases.

There comes the question of which structure could we use in order to aggregate rules dealing adequately with representation non-determinism, multiplicity, indexing and that would permit a unification of the mechanisms of addition, removal and modification. In the next section, we discuss the usage of coproducts in the category T-**Rules** as collections of rules, and discuss why this choice is adequate regarding the previously mentioned issues.

5.2 Coproducts as rule collections

Coproducts (see Definition 172) can be seen as the categorial generalization of the disjoint union operation in **Set** and other set-based categories such as **Graph**, T-**Graph** and T-**Span**. Given three rules r_1, r_2 and r_3 in T-**Span**, their coproduct is their disjoint union $r_1 + r_2 + r_3$ together with the respective injection morphisms $inj_i \to r_1 + r_2 + r_3$, for $i \in \{1, 2, 3\}$, as the following diagram shows.



Generically, we refer to a coproduct of a collection of rules $\{r_i\}_{i\in I}$ through the respective collection of injections, which we will denote as

$$\coprod\{r_i\}_{i\in I} = \{inj_i : r_i \to \sum_{p\in I} r_p\}_{i\in I}$$

for $I \subseteq \mathbb{N}$. Roughly speaking, the images of injections act as a tagging system, identifying each component of the collection within the amalgamation object $\sum_{p\in I} r_p$. The coproduct construction has the following interesting properties:

- it is a categorial constructions determined uniquely up-to-isomorphism, and consequently, less sensible to representation non-determinism than most set-based definitions;
- it allows repetition of elements and indexing through the injection morphisms;
- addition and removal of elements are independent of naming, assuming that we treat as "names" the injections themselves.

Coproducts, as sets, can also be seen a a collection of all its components, represented by the indexed collection $\{r_1, \ldots, r_n\}$ of all domains of injections. Similarly to set rewriting, we can define our notion of rewriting for coproducts.

Definition 93 (Coproduct rewriting)**.** *Let $R = \{r_i\}_{i\in I}$ be an I-indexed collection of graph rules in T-***Span***. We denote the coproduct rewriting operation by*

$$\frac{inj_k \in \coprod R \qquad dom(inj_k) = r \qquad r \overset{\alpha,m}{\Longrightarrow} r'}{\coprod R \overset{inj_k,\alpha,m}{\Rightarrow} \coprod(\{r_i\}_{i\neq k} \uplus \{r'\})}$$

Notice that, in order to keep track of which exact rule we are modifying, we must record which particular injection "injects" the rewritten element into the coproduct object. One of the advantages of this definition is that the whole updating process of the

ruleset may be totally described diagrammatically. Suppose, for instance, that we have a coproduct $\coprod\{r_1, r_2, r_3\}$ and a second-order rewriting $r_3 \overset{\alpha,m}{\Longrightarrow} r_3''$. We may obtain a coproduct rewriting by calculating the coproduct $\coprod\{r_1, r_2, r_3''\}$ of the modified r_3'' and all unmodified elements of the original coproduct, i.e., $\{r_1, r_2\}$. Figure 5.1 shows this update operation as a diagram in T-**Span**. The problems of representation non-determinism and indexing are solved because we keep the rewritten injection as part of the label. Intuitively, coproduct rewriting follows the same principle as set rewriting, but without the issues we have discussed previously.

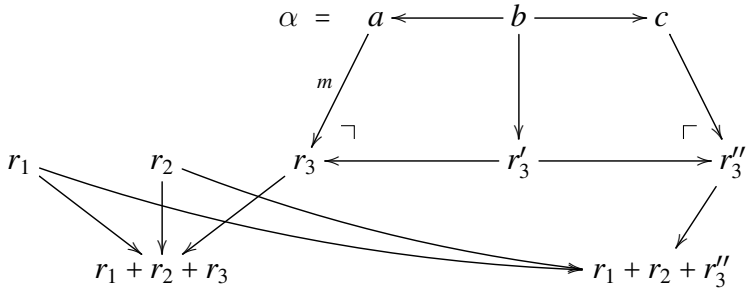
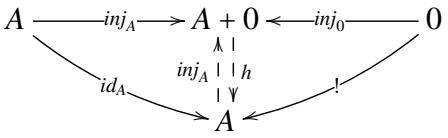


Figure 5.1: Example of coproduct rewriting.

Regarding creation and deletion of elements, we can obviously achieve them by means of taking the diagram formed by all domains of injections (i.e., the rules themselves) removing or adding some of them in the diagram, and recalculating the coproduct of the new diagram. That would be analogous to the addition and removal presented in set. However, that would lead to a definition distinct in nature to the modification mechanism, and it would be nice if we could find some uniformity. For inspiration in this issue, we recall a very basic property of category theory regarding initial objects and coproducts.

Proposition 94. *Let $\mathscr{C}$ be a category with coproducts and initial object* 0*. Then, all object A is isomorphic to $A + 0$, i.e. $A \cong A + 0$.*

Proof. Due to initial object property, there is a pre-coproduct $A \xrightarrow{id_A} A \xleftarrow{!} 0$, and hence a unique arrow $k : A + 0 \to A$. Moreover, also due to initial object property, $inj_A \circ ! = inj_0$, and hence $inj_A : A \to A + 0$ acts as h^{-1}.



□

The practical consequence is that, in a given sense, the initial object 0 acts as an identity for the coproduct calculation. This means that, although $\coprod\{r_1, r_2\} \neq \coprod\{r_1, r_2, 0\}$, since they have a distinct number of injections, they both have isomorphic summation objects, i.e., $r_1 + r_2 \cong r_1 + r_2 + 0$. If we consider a 2-rule $\alpha = 0 \leftarrow 0 \rightarrow r_3$ and the second-order rewriting $0 \overset{\alpha,id_0}{\Longrightarrow} r_3$, it is possible to form the coproduct rewriting $\coprod\{r_1, r_2, 0\} \overset{inj_3,\alpha,id_0}{\Rightarrow} \coprod\{r_1, r_2, r_3\}$ as shown in Figure 5.2. Notice that in this case, the match for the rule α is an

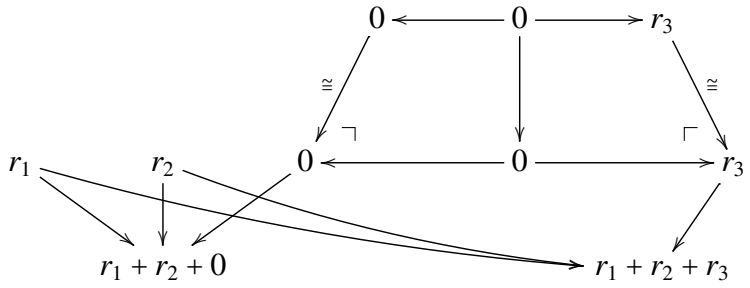


Figure 5.2: Creation as coproduct rewriting.

isomorphism. Since its left-hand side is empty, it could also be used to include the right-hand side rule r_3 as a part of rules r_1 and r_2, and the match would not be iso. The intuition we obtain is that an empty injection, i.e, an injection from the initial object, acts as an *empty slot*. This way, we would be inserting elements in a vacant part of the coproduct object. In contrast, if the same 2-rule acts over an existing rule, let us say r_2, then the second-order rewriting would be modifying the components of an already filled slot. A nice consequence of this view of addition of elements is that we obtain a dual interpretation for deletion. For example, the rule $\alpha^{-1} = r_3 \leftarrow 0 \rightarrow 0$ removes the structure r_3 from some existing rule, or even the whole rule from the collection by transforming it into an empty slot, as in the rewriting $\coprod\{r_1, r_2, r_3\} \overset{inj_3, \alpha^{-1}, id_{r_3}}{\Rightarrow} \coprod\{r_1, r_2, 0\}$, shown in Figure 5.3.

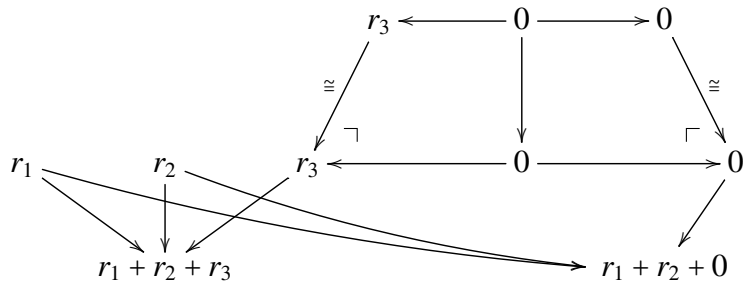


Figure 5.3: Deletion as coproduct rewriting.

In both cases, for rules in the format the distinction between modification and creation or addition depends on the property of the match being an isomorphism or not, and in T-**Span** we have that the comatch is iso if the match is iso. The rules that perform these transformations are actually inverse to each other. Consider the 2-rule $\alpha = p \leftarrow 0 \rightarrow 0$ and second-order rewritings $c = r \overset{\alpha, m}{\Longrightarrow} r'$ and $d = r' \overset{\alpha^{-1}, m'}{\Longrightarrow} r$. We have the following cases when m and m' are not isomorphisms:

- creation: since the LHS is empty, the match m will always be mono. Hence, it may only fail to be iso by not being epi, and this cases characterizes the addition of the pattern p into the rule r;
- deletion: the match can fail to be iso by being:
  - monic, but non-epi: the pattern represented by p is removed from the rule r';

- epi, but non-monic: a rule r', which is smaller than p, is completely deleted. This happens because of non-injectivity in the match m';
- non-monic and non-epi: a sub-pattern of p (due to non-injectivity of the match) is removed from r', but there remains some elements in the resulting rule r since m' was not surjective.

This analysis shows that 2-rules with the shape $0 \leftarrow 0 \rightarrow r$ and $r \leftarrow 0 \rightarrow 0$ are quite versatile. If we ensure that the match is isomorphic, they can be used to delete and create rules in a collection by converting them into the empty rule or from the empty rule, respectively. Notice that this only works as we intend to if somehow we could identify a coproduct $\coprod\{r_1, r_2\}$ with $\coprod\{r_1, r_2, 0\}$, i.e., if injections from 0 would act as a neutral element of coproduct operation. The next definitions address this issue.

Definition 95 (Active injections of coproducts)**.** *Let $\mathscr{C}$ be a category with coproducts and initial object 0. Given a coproduct $c = \coprod R$ in $\mathscr{C}$, we denote by $\mathscr{A}(c)$ the subcollection $A \subseteq \mathbb{N}$ of injections such that their domain are not initial in $\mathscr{C}$. Formally,*

$$\mathscr{A}(c) = \{inj \mid inj \in c \wedge dom(inj) \ncong 0\}$$

Definition 96 (Coproduct collection of rules)**.** *Let $c = \coprod R$ be a coproduct in T-***Span***. We say that c is a* coproduct collection of rules *(alternatively, coproduct rule collection) iff*

1. *c is countably infinite;*
2. *$\mathscr{A}(c)$ is finite;*
3. *for all $inj \in \mathscr{A}(c)$, $dom(inj)$ is finite.*

A coproduct rule collection correspond to a coproduct for a diagram composed of a finite number of non-empty rules, and an infinite number of empty rules, as depicted below.

$$r_1 + 0 + 0 + r_2 + r_3 + 0 + 0 + 0 + 0 + 0 + 0 \ldots$$

The intuition is that we have an infinite number of empty slots, representing possible additions to the collection of rules, and a finite number of occupied slots for rules, represented by the active indices. The infinite collection of injections from 0 solves the problem of finding new names for the addition of rules, since we consider the indexed collection of active injections as the representation of named rules. We can employ this same pattern to describe collections in other categories as the next definition shows.

Definition 97 (Coproduct collection of typed graphs)**.** *Let $c = \coprod R$ be a coproduct in T-***Graph***. We say that c is a* coproduct collection of typed graphs *(alternatively, coproduct typed graph collection) iff c is countably infinite, $\mathscr{A}(c)$ is finite and for all $inj \in \mathscr{A}(c)$, $dom(inj)$ is finite.*

The final step is to include rewriting of rules with negative application conditions in our notion of coproduct rewriting.

Definition 98 (Coproduct rewriting with NACs)**.** *Let $R = \{r_i\}_{i \in I}$ be an I-indexed collection of graph rules in T-***Span** *and (N, α) a 2-rule with NACs. We denote coproduct rewriting with NACs by the following rule*

$$\frac{inj_k \in \coprod R \qquad dom(inj_k) = r \qquad r \xrightarrow{(N,\alpha),m} r'}{\coprod R \overset{inj_k,(N,\alpha),m}{\Rightarrow} \coprod(\{r_i\}_{i \neq k} \uplus \{r'\})}$$

Now, we recall the traditional definition for graph grammars and modify it in order to use coproduct collections.

5.3 Graph grammars with coproduct rule collection

This section redefines the basic definitions for graph grammar models, in order to adapt them towards the usage of coproduct rulesets. This way, we may obtain a better basis before adding a higher-order layer to the specifications. The modifications are basically technical, and do not modify the original intuition behind graph grammar specifications.

Definition 99 (Graph grammar)**.** *A (first-order) graph grammar is a tuple* (T, D_0, D_1) *where T is a type graph,* D_0 *is a coproduct collection of graphs in T-***Graph** *and* D_1 *is a coproduct collection of graph rules in T-***Span***.*

Notice that we are considering the initial graph to be a coproduct collection of graphs. This generalizes the situation of a unique initial graph, and will provide a convenient uniformity between rewriting levels. In practice, however, all our examples will comprise of a unique graph. This choice requires that we adapt the notion of direct derivation in order to tag in the label not only the injection for rule and match, but also the injection of the target.

Definition 100 (Derivation)**.** *Let* (T, D_0, D_1) *be a graph grammar (cf. Definition 99). We define a direct derivation by the following rule*

$$\frac{D_0 \overset{g,dom(r),m}{\Rightarrow} D_0' \qquad g \in \mathscr{A}(D_0) \qquad r \in \mathscr{A}(D_1)}{D_0 \overset{g,r,m}{\Longrightarrow} D_0'}$$

Notice that we are using the injections of the chosen graph and rule as names in the label of the transformation step. This accounts for identifying precisely which rule we are referring to, even if there is repetition in the coproduct rule collection. We set as label the triple (g, r, m) where g is the injection of the target graph, r is the injection of the rule and $m : dom(r) \to dom(g)$ is the match. Finally, we require that the rewriting does not use an empty rule over an empty graph. Because coproducts collections of rules have an infinite number of copies of the empty rule 0, we wish to avoid applying one of them.

However, there is one important consequence of this particular definition for derivation that differs from the traditional one. It refers to the fact we are requiring the graph being rewritten not to be *empty*, by means of condition $g \in \mathscr{A}(D_0)$. This is required because in our modelling an empty graph injection represents an empty slot within a collection. This restriction, however, does not affect most of the scenarios, and we can always use a non-empty encoding avoiding to become empty during execution.

5.4 Second-order graph grammars

In this section, we discuss how to add a second-order transformation layer over a first-order graph grammar. We start by the most simple example, where the second-order layer only changes the collection of rules by means of a collection of two rules. Later, we consider more complex specifications that also modify the initial graph and the type graph of the initial system. Before entering in the details of the models, we are required to establish a category to describe collections of 2-rules, which we call T-**Span**^{2}.

Definition 101. *T-**Span**^{2} is the functor category* $[\mathbf{Span} \to T\text{-}\mathbf{Span}]$.

Proposition 102. *T-**Span**^{2} is adhesive and finitely co-complete.*

Proof. (Sketch) T-**Graph** is a topos. Hence, the functor category $[\mathscr{C} \to T\text{-}\mathbf{Graph}]$ is a topos if $\mathscr{C}$ is small. We know **Span** is small, hence T-**Span** and T-**Span**^{2} are toposes as well. All toposes are adhesive (LACK; SOBOCINSKI, 2006) and finitely co-complete (GOLDBLATT, 2006). □

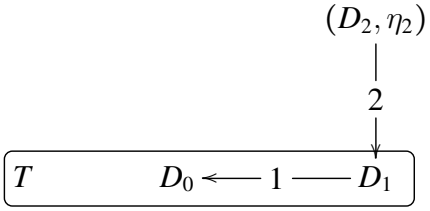
5.4.1 Simple second-order graph grammars

We start by defining what we call a simple second-order graph grammar. We call this definition "simple" because the only novelty in comparison with a first-order graph grammar is the introduction a collection D_2 of 2-rules together with a function η_2 assigning respective NACs to each active rule index. The new collection of 2-rules affects only the underlying collection of rules, and thus do not have a direct impact over the initial graph or type graph.

Definition 103 (Simple second-order graph grammar)**.** *A simple second-order graph grammar (S-SOGG) is a tuple* $(T, D_0, D_1, D_2, \eta_2)$ *where*

- (T, D_0, D_1) *is a first-order graph grammar (cf. Definition 99);*
- D_2 *is a coproduct collection of 2-rules in T-**Span**^{2};*
- η_2 *is a total function mapping active rule injections* $inj \in \mathscr{A}(D_2)$ *to a collection of NACs for rule* $dom(inj)$.

The state of the graph grammar is not just the graph, but also the collection of rules. Regarding the grammar execution, we have two kinds of possible evolutions: the first one is given by the interaction of D_1 and D_0, generating a new current graph collection, and the other comes from the interaction of D_2 and D_1, providing an updated rule collection. The following diagram presents visually those interactions.



Definition 104 (Derivation of a S-SOGG)**.** *A direct derivation of a S-SOGG* $\mathscr{G} = (T, D_0, D_1, D_2, \eta_2)$ *is given by the following rules.*

$$\frac{D_0 \overset{g,dom(r),m}{\Rightarrow} D_0' \qquad g \in \mathscr{A}(D_0) \qquad r \in \mathscr{A}(D_1)}{(T, D_0, D_1) \xRightarrow{(1,g,r,m)} (T, D_0', D_1)} \tag{1}$$

$$\frac{D_1 \overset{r,(\eta_2(\alpha),dom(\alpha)),m}{\Rightarrow} D_1' \qquad r \in \mathscr{A}(D_1) \qquad \alpha \in \mathscr{A}(D_2)}{(T, D_0, D_1) \overset{(2,r,\alpha,m)}{\Longrightarrow} (T, D_0, D_1')} \tag{2}$$

Direct derivations of kind (1) and (2) are referred as, respectively, first-order and second-order, and are tagged accordingly. A derivation is an arbitrary sequence

$$\mathcal{G}_0 \overset{(w_1,x_1,y_1,z_1)}{\Longrightarrow} \mathcal{G}_1 \overset{(w_2,x_2,y_2,z_2)}{\Longrightarrow} \mathcal{G}_2 \overset{(w_3,x_3,y_3,z_3)}{\Longrightarrow} \ldots \overset{(w_n,x_n,y_n,z_n)}{\Longrightarrow} \mathcal{G}_n$$

of direct derivations starting with $\mathcal{G}_0 = (T, D_0, D_1)$.

Example 105. *As an example of a simple SOGG, we use the graph grammar of Figure 2.7, connecting clients and servers, as the initial state. We introduce a 2-rule collection consisting of a single 2-rule* α*, as shown in Figure 5.4. Since we do not have the set of names P, we use the names of rules, 2-rules and graph as the injection indices* inj_i*. The unique 2-rule implements a model transformation that modifies the reading policy of data. In the original system, the reading operation removes the message from the server. The 2-rule* α *modifies this aspect in the following way: it matches all rules that move a data away from servers, augmenting their RHS by creating a fresh copy of the data token in the server. The 2-rule* α *contains a NAC that makes it to be executed only once over a given rule. The only rule in the original specification affected by* α *is* getData*, and the second-order rewriting that modifies it can be seen in Figure 5.5. This second-order rewriting gives rise to the following second-order derivation:*

$$(T,\ D_0,\ \{\textsf{sendMSG, getDATA, receiveMSG, deleteMSG}\})$$
$$\overset{(2,\textsf{getDATA},\alpha,m)}{\Longrightarrow}$$
$$(T,\ D_0,\ \{\textsf{sendMSG, getDATA', receiveMSG, deleteMSG}\})$$

5.4.2 Retyping-aware simple second-order graph grammars

Although the main innovation regarding second-order graph grammar comes from modifications in rules, some changes in other components of the specification, such as type graphs, may still be required. One clear example refers to the proposed model transformation of Chapter 1, depicted in Figure 1.4. While the addition of a place in that representation meant to add a node in the instance graph, in the interpretation of a place-transition system as a graph grammar, places are represented by means of node types. Therefore, in order to add or remove a place, we are required to modify the type graph and adjust the instance graphs accordingly. We can say that rule-based rewriting may not be as effective for untyped graphs than it is for typed graphs, since most of the rule structure is guided by the typing information. Because of this, we intend to consider for typed graph modifications based on spans across types, which have been vastly explored in the context of graph grammar morphisms.

Essentially, those mechanisms define how to adapt typed graph instances due to removal and inclusion of elements in the type graph. The straightforward way of retyping instances is induced by a span $T \leftarrow T' \rightarrow T''$ between two type graphs. Instances of T are converted into instances of T'' through a combination of the reverse pullback functor and direct composition functor between particular comma categories T-**Graph** and T''-**Graph**. These constructions are traditional in the area of graph transformation, and we recall them in the following definitions.

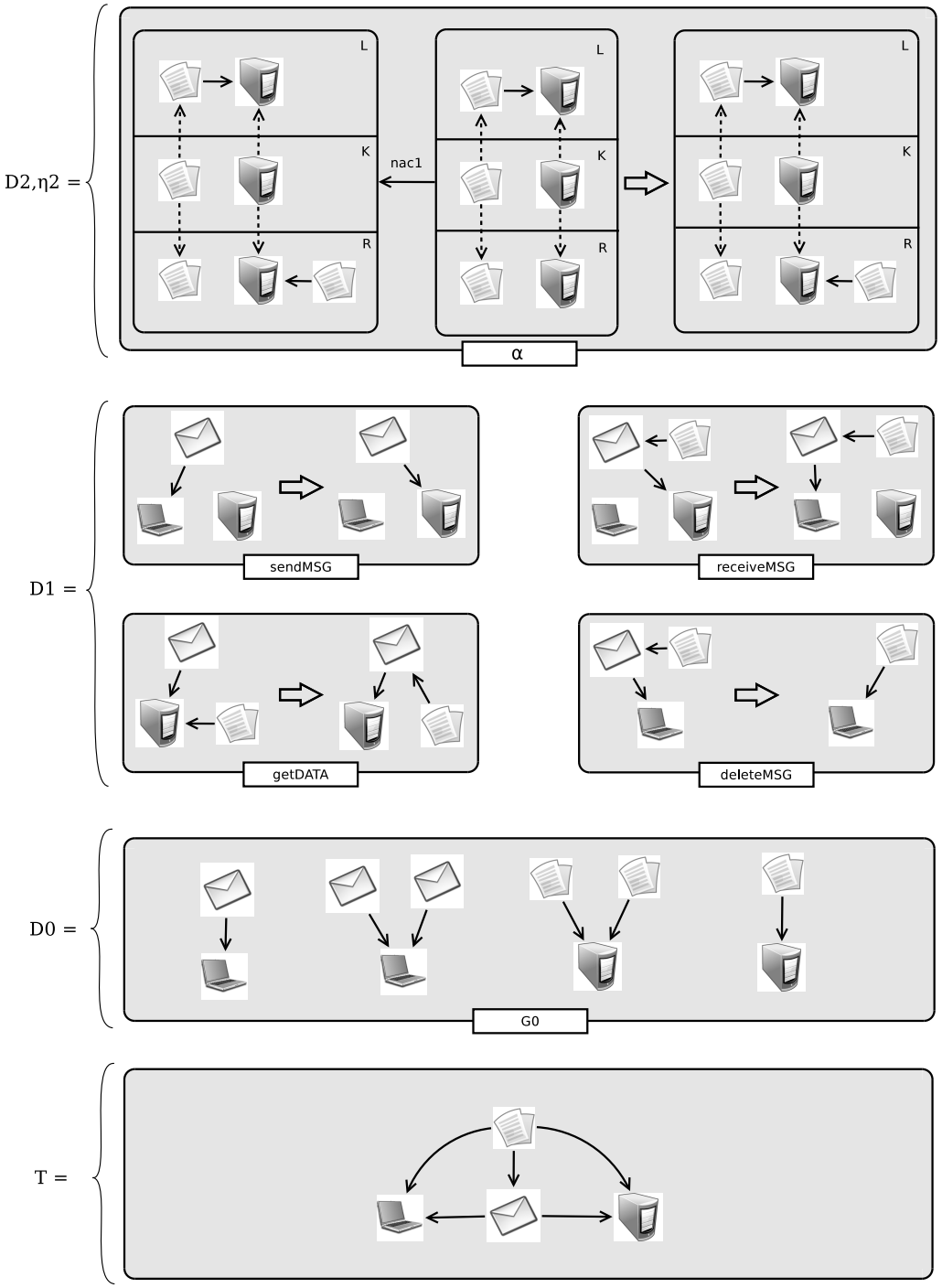


Figure 5.4: Simple second-order graph grammar.

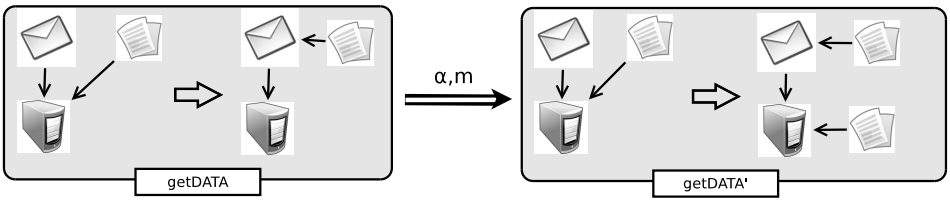
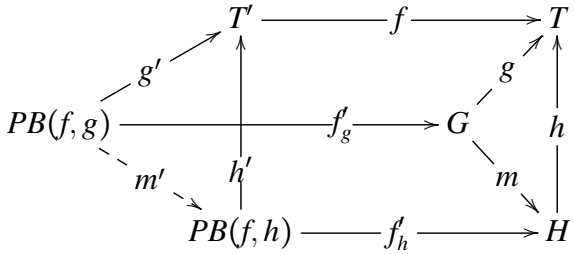
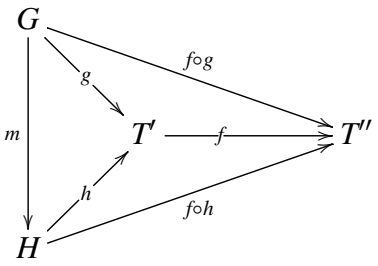


Figure 5.5: Second-order rewriting modifying getDATA.

Definition 106 (Reverse retyping functor)**.** *Let PB be a fixed* choice of pullback *in* **Graph**. *Each arrow* $f : T' \to T$ *allows to define a* reverse pullback functor $f^{\triangleleft} : T\text{-}\mathbf{Graph} \to T'\text{-}\mathbf{Graph}$*, defined by* $f^{\triangleleft}_{Obj}(g : G \to T) = g' : PB(f,g) \to T'$*, and* $f^{\triangleleft}_{Mor}(m : G \to H)$ *is the unique arrow from pre-pullback* $(PB(f,g), g', f'_g; m)$ *to* $PB(f,h)$*, as shown in the diagram below.*



Definition 107 (Direct retyping functor)**.** *Each arrow* $f : T' \to T''$ *induces a functor* $f^{\triangleright} : T'\text{-}\mathbf{Graph} \to T''\text{-}\mathbf{Graph}$ *defined by* $f^{\triangleright}_{Obj}(g) = f \circ g$ *and* $f^{\triangleright}_{Mor}(m) = m$*, as shown in the following diagram.*

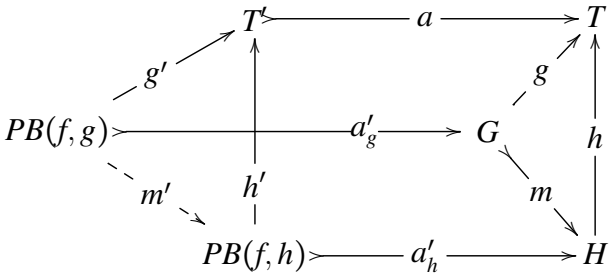


Definition 108 (Span retyping functor)**.** *Each span* $s = T \xleftarrow{a} T' \xrightarrow{b} T''$ *in* T-**Graph** *induces a functor* $s^{\triangleleft\triangleright} : T\text{-}\mathbf{Graph} \to T''\text{-}\mathbf{Graph}$ *defined by* $s^{\triangleleft\triangleright} = b^{\triangleright} \circ a^{\triangleleft}$.

Now, we investigate in under which conditions monomorphisms are preserved across both the reverse and direct retyping functors.

Proposition 109. *If* $a : T' \to T$ *is mono, then* $a^{\triangleleft} : T\text{-}\mathbf{Graph} \to T'\text{-}\mathbf{Graph}$ *preserves monomorphisms.*

Proof. Suppose a monomorphism $m : g \to h$ in T-**Graph**. We then calculate $f^{\triangleleft}(m) = m'$ as shown below



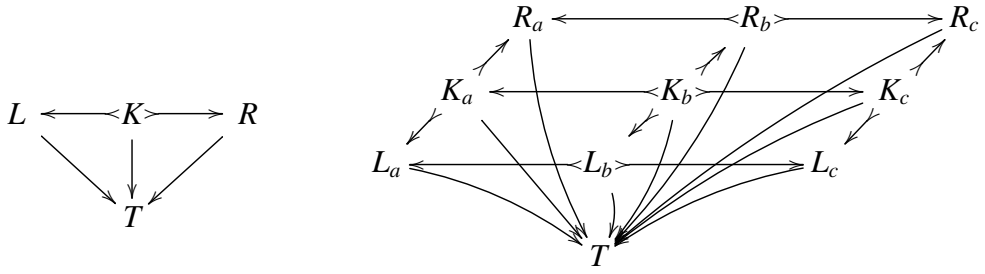
Because pullbacks preserve monomorphisms in all categories and a is mono we have that a'_g and a'_h are mono. Because $a'_g; m = m'; a'_h$, and both $a'_g; m$ and a'_h are mono, so it is m'. □

Proposition 110. *For any $b : T' \to T''$, $b^{\triangleright} : T'$-**Graph** $\to T''$-**Graph** preserves monomorphisms.*

Proof. $b^{\triangleright}(m) = m$, and all mono in T-**Graph** are also mono in **Graph**. □

Proposition 111. *Given a graph span $s = T \xleftarrow{a} T' \xrightarrow{b} T''$ such that a is mono, then $s^{\triangleleft\triangleright}$ preserves rules and 2-rules.*

Proof. Both rules and 2-rules, respectively, may be seen as the following commutative diagrams in **Graph**



rule r 2-rule α

where the top arrows are all monomorphisms. Since a is mono, we have that $a^{\triangleleft}$ preserves monos by Proposition 109. By Proposition 110, $b^{\triangleright}$ preserves monos, and therefore $s^{\triangleleft\triangleright}$ also preserves mono. Hence, $s^{\triangleleft\triangleright}(r)$ and $s^{\triangleleft\triangleright}(\alpha)$ are commutative diagram in **Graph** where the top is composed of all monos and the lower vertice is T'', comprising of valid rules and 2-rules, respectively, in T''-**Graph** and T''-**Span**. □

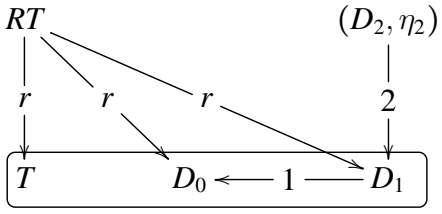
We will represent by the same name $s^{\triangleleft\triangleright}$ the rewriting functors $s^{\triangleleft\triangleright} : (T\text{-}\mathbf{Graph} \to T''\text{-}\mathbf{Graph})$ and $s^{\triangleleft\triangleright} : (T\text{-}\mathbf{Span} \to T''\text{-}\mathbf{Span})$, induced by the same s. By retyping the diagrams representing the current graph and graph rules according to $s^{\triangleleft\triangleright}$, we can convert a whole T-typed graph grammar (T, D_0, D_1) into a well-formed T''-typed graph grammar (T'', D''_0, D''_1).

In our examples, all rewritings layers have to be typed over the same graph, which leads to the notion that T must contain all types required for all elements of the specification. However, it may be common that lower-order layers use just a subset of all possible elements in the type graph, while higher-order layers employ additional types. For instance, in the example of Chapter 1, we have to increase in exact one element the type graph, representing a new place. Moreover, the rewriting rules need to add one instance of this new kind to the RHS of all existing rules. It may also be the case that we may wish to delete some elements from the type graph, as, for instance, removing some places from the example place-transition system. Therefore, we propose to represent retyping as an operation affecting the first-order layer of the specification, which leads to the following definition:

Definition 112 (Retyping-aware simple second-order graph grammar)**.** *A retyping-aware simple second-order graph grammar (RS-SOGG) consists of a tuple $(T, D_0, D_1, RT, D_2, \eta_2)$ where*

- (T, D_0, D_1) *is a first-order graph grammar.*
- $RT = T \overset{a}{\leftarrowtail} T' \overset{b}{\rightarrowtail} T''$ *is a monic span in* **Graph**.
- D_2 *is a coproduct collection of 2-rules in* T''-**Span**^{2}.
- η_2 *is a total function mapping an active rule injection* $r \in \mathcal{A}(D_2)$ *to a set of NACs for its domain* $dom(r)$.

This definition has the intuition of providing the second-order layer its own type graph T'', while recording through the span RT the notion of evolution of the type graph T. The RT-induced retyping operation affects all first-order structures: both initial graphs and rules. The following diagram depicts how some elements of the specification affect others, where r stands for retyping, 1 for first-order rewriting and 2 for second-order rewriting.



Definition 113 (Retyped RS-SOGG)**.** *Given a RS-SOGG* $\mathcal{G} = (T, D_0, D_1, RT, D_2, \eta_2)$*, we define its respective* retyped grammar*, written* $\mathcal{G}_{RT}$*, as the S-SOGG*

$$\mathcal{G}_{RT} = (T', D_0'', D_1'', D_2, \eta_2)$$

where $RT^{\triangleleft\triangleright}(D_0) = D_0''$ *and* $RT^{\triangleleft\triangleright}(D_1) = D_1''$.

The S-SOGG associated with the RS-SOGG is given by retyping all first-order structures according to the RT component. Notice in Definition 113 that, because D_2 was already typed over T'', it is maintained unchanged in its respective *retyped S-SOGG*. The retyped grammar defines the behavior of the RS-SOGG specification.

Definition 114 (Derivation of RS-SOGG)**.** *The derivations of a RS-SOGG* $\mathcal{G} = (T, D_0, D_1, RT, D_2, \eta_2)$ *are the derivations of its retyped graph grammar* $\mathcal{G}_{RT}$.

Example 115 (Place-transition system as S-SOGG)**.** *Figure 5.6 shows a characterization of the place-transition example of Chapter 1 as a retyping-aware S-SOGG. The first-order layer* (T, D_0, D_1) *represents the graph grammar view of the original place-transition system. The retyping span creates an additional place in the specification as a new node in the type graph. The unique 2-rule modifies all the existing rules, creating new nodes of the new type on their RHS. The 2-rule only acts once on each rule due to the existence of a NAC ensuring that new places will only be created if they do not exist already.*

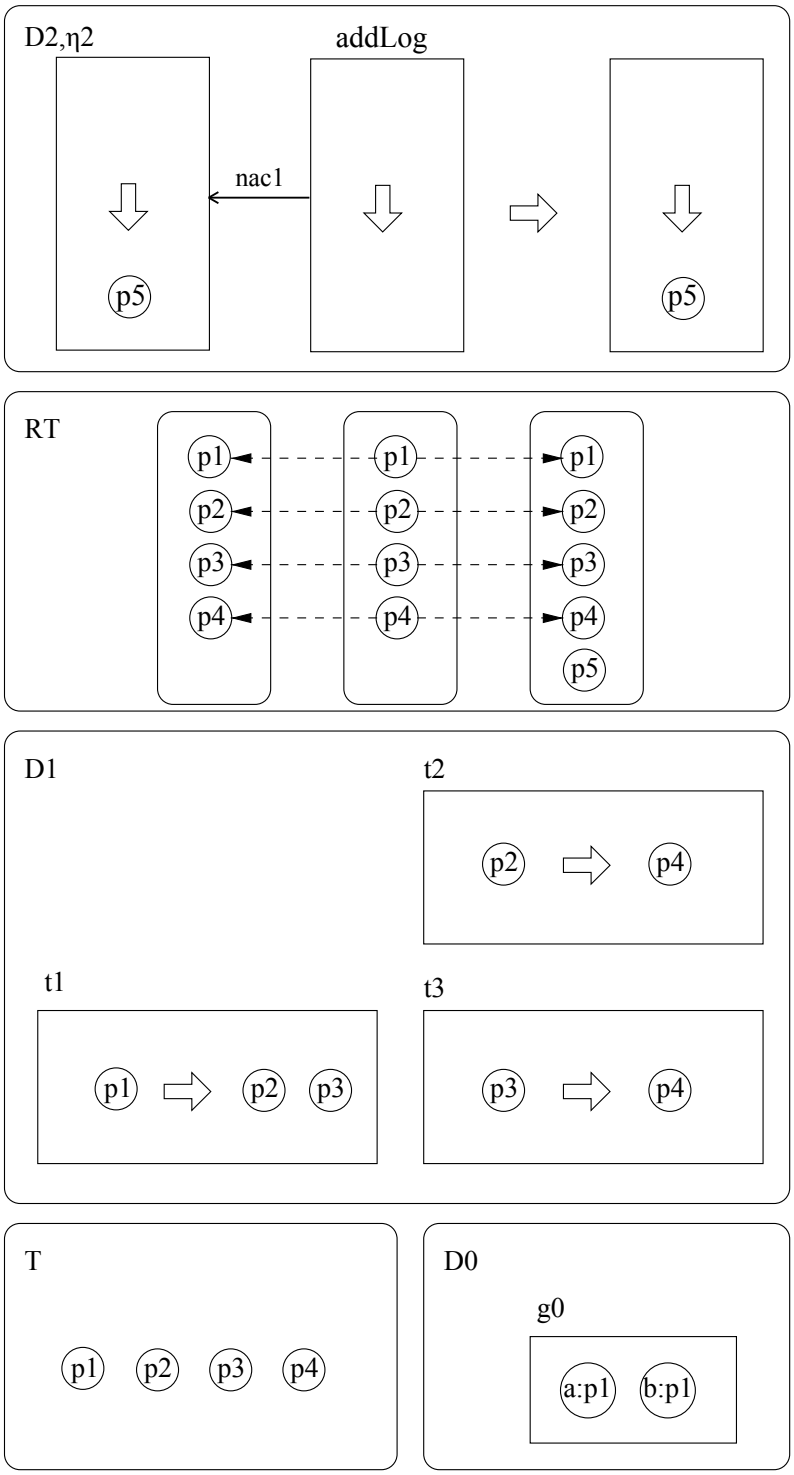


Figure 5.6: Place-transition example as RS-SOGG.

5.4.3 Retyping-aware complete second-order graph grammars

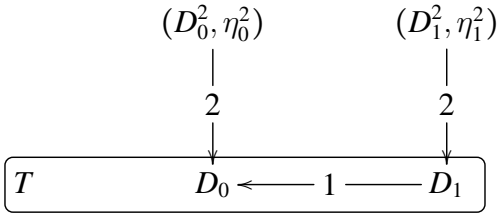
A retyping-aware S-SOGG allows to define modifications in the type graph, which affects all first-order structures, and also in the collection of rules, through second-order rewriting. Notice, however, that there are settings where modifications in rules may also require additional adjustments in the initial graph. One example is to think of a log policy similar to the one from the place-transition system, but in the client and server scenario.

For example, suppose there should be a global log object responsible for a execution trace the system, and that the structure must be read and updated by every rule in the specification. We notice that the creation of additional structures in rules can be done by means of 2-rules. However if we do not create at least one instance of the global log object in the initial graph, all modified rules will have no match. Hence, we propose a definition for what we call a "complete" second-order graph grammar, where besides a collection of 2-rules, we also include a collection of graph rules whose purpose is to update the initial graph of the specification.

Definition 116 (Complete second-order graph grammar)**.** *A complete second-order graph grammar (C-SOGG) is a tuple* $(T, D_0, D_1, D_0^2, D_1^2, \eta_0^2, \eta_1^2)$ *where*

- (T, D_0, D_1) *is a first-order graph grammar (cf. Definition 99)*
- D_0^2 *is a coproduct collection of rules in* T-**Span**
- D_1^2 *is a coproduct collection of 2-rules in* T-$\mathbf{Span}^2$
- η_0^2 *associates a collection of NACs to each active injection of* D_0^2
- η_1^2 *associates a collection of NACs to each active injection of* D_1^2

Notice that there are two rule collections affecting the initial graph: the original D_1 and the new D_0^2. Although both are objects of the same kind, their purposes are distinct: D_1 represents the execution for the model, while D_0^2 represents a model-transformation modification that prepares initial conditions. Therefore, in order to distinguish their role, their tags in derivations are distinct, according to the following scheme.



Definition 117 (Derivation of C-SOGG)**.** *A direct derivation of a C-SOGG* $(T, D_0, D_1, D_0^2, D_1^2, \eta_0^2, \eta_1^2)$ *is given by the following rules.*

$$\frac{D_0 \overset{g,dom(r),m}{\Rightarrow} D_0' \qquad g \in \mathscr{A}(D_0) \qquad r \in \mathscr{A}(D_1)}{(T, D_0, D_1) \xRightarrow{(1,g,r,m)} (T, D_0', D_1)} \tag{1}$$

$$\frac{D_0 \overset{g,(\eta_0^2(r),dom(r)),m}{\Rightarrow} D_0' \qquad g \in \mathscr{A}(D_0) \qquad r \in \mathscr{A}(D_0^2)}{(T, D_0, D_1) \overset{(2,g,r,m)}{\Longrightarrow} (T, D_0', D_1)} \tag{2}$$

$$\frac{D_1 \overset{r,(\eta_1^2(\alpha),dom(\alpha)),m}{\Rightarrow} D_1' \qquad r \in \mathscr{A}(D_1) \qquad \alpha \in \mathscr{A}(D_1^2)}{(T, D_0, D_1) \overset{(2,r,\alpha,m)}{\Longrightarrow} (T, D_0, D_1')} \tag{3}$$

A derivation of a C-SOGG is an arbitrary sequence

$$\mathcal{G}_0 \overset{(w_1,x_1,y_1,z_1)}{\Longrightarrow} \mathcal{G}_1 \overset{(w_2,x_2,y_2,z_2)}{\Longrightarrow} \mathcal{G}_2 \overset{(w_3,x_3,y_3,z_3)}{\Longrightarrow} \dots \overset{(w_n,x_n,y_n,z_n)}{\Longrightarrow} \mathcal{G}_n$$

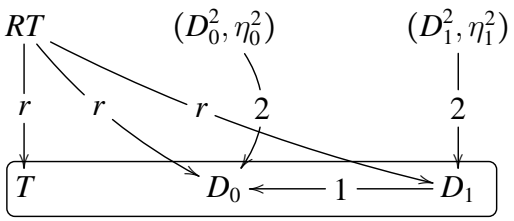
of direct derivations starting with $\mathcal{G}_0 = (T, D_0, D_1)$.

The important thing to notice is that an update of the initial graph, although being a first-order rewriting, has the same tag as an second-order rewriting. This accounts for the definition of the second-order layer (or model transformation layer) comprising both a first-order component (D_0^2, η_0^2) and a second-order component (D_1^2, η_1^2). Provided that we may wish to allow modifications also in the type graph as in a RS-SOGG, we also define a retyping-aware version of a C-SOGG.

Definition 118 (Retyping-aware complete second-order graph grammar)**.** *A retyping-aware complete second-order graph grammar (RC-SOGG) is a tuple* $(T, D_0, D_1, RT, D_0^2, D_1^2, \eta_0^2, \eta_1^2)$ *where*

- (T, D_0, D_1) *is a first-order graph grammar (cf. Definition 99)*
- $RT = T \nleftarrow T' \nrightarrow T''$ *is a monic span in* **Graph**
- D_0^2 *is a coproduct collection of rules in* T''-**Span**
- D_1^2 *is a coproduct collection of 2-rules in* T''-$\mathbf{Span}^2$
- η_0^2 *associates a collection of NACs to each active injection of* D_0^2
- η_1^2 *associates a collection of NACs to each active injection of* D_1^2

Within a RC-SOGG, the affect of each component of the second-order layer over the first-order components can be observed in the following diagram.



As it would be expected, derivations in RC-SOGG are given by their respective retyped C-SOGG.

Definition 119 (Retyped C-SOGG)**.** *Given a RC-SOGG* $\mathcal{G} = (T, D_0, D_1, RT, D_0^2, D_1^2, \eta_0^2, \eta_1^2)$*, we define its respective retyped C-SOGG as*

$$\mathcal{G}_{RT} = (T'', D_0'', D_1'', D_0^2, D_1^2, \eta_0^2, \eta_1^2)$$

where $RT = T \overset{a}{\nleftarrow} T' \overset{b}{\rightarrow} T''$*,* $RT^{\triangleleft\triangleright}(D_0) = D_0''$ *and* $RT^{\triangleleft\triangleright}(D_1) = D_1''$*.*

Example 120 (Clients and servers with logging)**.** *Now we consider a log policy for rule execution in the client-server scenario. According to this policy, we have a particular node in the specification that acts as step-counter of the whole system execution. Each rule application must update the global log object by increasing a linked-list structure at each rule application. This system, modelled as a RC-SOGG, can be seen in Figure 5.7. The retyping component adds a new "log" node type, and a new edge type for connecting log nodes. The unique rule* initLog *in* D_0^2 *creates a single node of type log with a self-edge into the initial graph, assuring that all the modified rules will be able to execute. The modification in rules is given by the 2-rule* addLog*, which modifies the current rules in the following way: the rules must reads the log node, deletes the self-edge and create a new node with its own self-edge, connected to the previous one by means of a new arrow. The modified rules actually are augmenting a linked list. At the end of a n-step derivation, the log linked-list structure will contain* $n + 1$ *nodes.*

5.4.4 Create-delete-modify second-order graph grammar

We have defined the coproduct rewriting operation $\Rightarrow$ that allows us to represent both modification, creation and deletion of active rules. However, creation and deletion require special conditions: 2-rules in the format $0 \leftarrow 0 \rightarrow r$ and $r \leftarrow 0 \rightarrow 0$ (in T-**Span**) and also the guarantee that the match is an isomorphism. In our present definition of coproduct rewriting, we have considered a important restriction: both the involved 2-rule and rule must be *active*, i.e., non-empty. The restriction of active 2-rules represent the fact that empty injections only mean *empty slots*, and thus are not meant to be executed. The restriction on *targets* of second-order rewriting emphasize that we are meaning to represent modifications in current rules rather than creation of new ones. Notice that deletion of rules is still possible under these circumstances, either by single or successive modifications that end up removing all elements from a particular rule. The issue that arises if we remove the restriction on targets is the fact that a 2-rule of shape $0 \leftarrow 0 \rightarrow r$, when used with the purpose of creating a new rule, would always allow an infinite chain of rewritings. It is reasonable to consider that the inclusion or removal of rules of a specification should be controlled operations that must be executed a finite number of times for a given model transformation. Since coproduct collections have an infinite number of empty injections, we arrive at an important issue: there is no way to avoid an infinite sequence of rule creations. Negative application conditions can be used to enforce that 2-rules of kind $0 \leftarrow 0 \rightarrow r$ behave exclusively for creation purposes, but not the opposite, i.e., to ensure only modification of existing rules but not creation of new ones. Hence, although we can represent both creation and deletion as a rewriting between two coproduct collections, both operations seem to require a special scheduling policy that would create and delete rules only once and by means of isomorphic matches. That is the purpose of the model we name *create-delete-modify simple second-order graph grammar*. We start by defining a creation-deletion sequence.

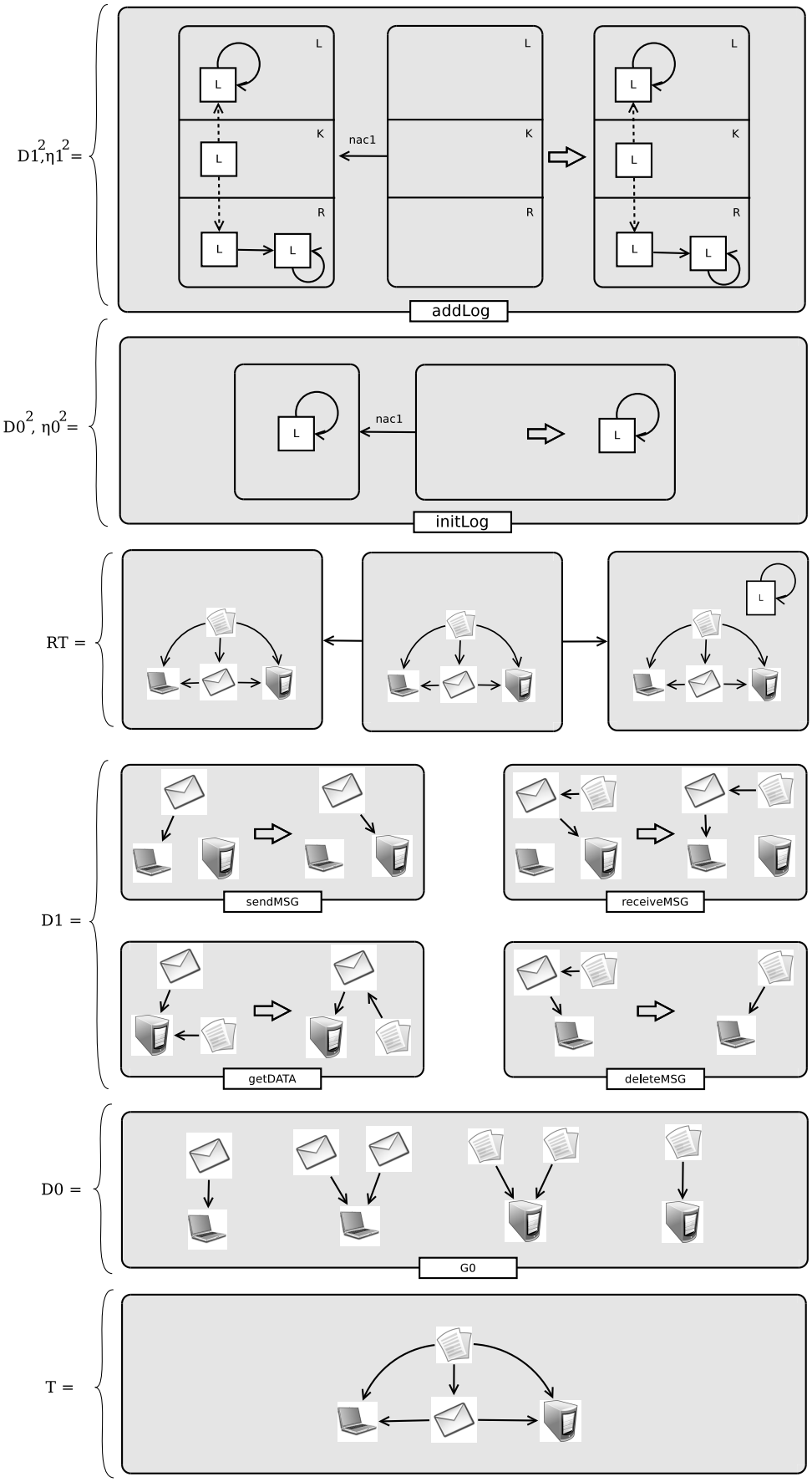


Figure 5.7: Complete second-order graph grammar adding logging to base grammar.

Definition 121 (Create-delete statement)**.** *A (T-typed) create-delete statement is a pair*

$$s = (op, r)$$

such that $op \in \{+, -\}$ *and* r *is a graph rule in* T-**Span**. *A create-delete sequence is a (possibly empty) finite sequence*

$$s_1, s_2, \ldots, s_n$$

of create-delete statements.

From a create-delete statement, we define a coproduct rewriting associated with it by means of second-order rewriting, as follows.

Definition 122 (Create-delete rewriting)**.** *Given a first-order grammar* $\mathcal{G} = (T, D_0, D_1)$ *and a create-delete statement* $s = (op, r)$*, we define a create-delete rewriting by the following rules.*

$$\frac{s = (+, r) \qquad \alpha = 0 \leftarrow 0 \rightarrow r \qquad D_1 \overset{i,\alpha,m}{\Rightarrow} D_1' \qquad m \text{ is iso}}{(T, D_0, D_1) \xRightarrow{(2,i,s,m)} (T, D_0, D_1')} \tag{1}$$

$$\frac{s = (-, r) \qquad \alpha = r \leftarrow 0 \rightarrow 0 \qquad \exists m, (\, D_1 \overset{i,\alpha,m}{\Rightarrow} D_1' \ \wedge \ m \text{ is iso} \,)}{(T, D_0, D_1) \xRightarrow{(2,i,s,m)} (T, D_0, D_1')} \tag{2}$$

$$\frac{s = (-, r) \qquad \alpha = r \leftarrow 0 \rightarrow 0 \qquad \nexists m, (\, D_1 \overset{i,\alpha,m}{\Rightarrow} D_1' \ \wedge \ m \text{ is iso} \,)}{(T, D_0, D_1) \xRightarrow{(2,i,s,m)} (T, D_0, D_1)} \tag{3}$$

Rule (1) handles creation of rules. Notice that operations of kind $(+, r)$ are always possible for any D_1, since coproduct collections have an infinite number of empty slots. Rules (2) and (3) handle deletion of rules. For a statement $(-, r)$, there may exist or not a rule isomorphic to r in D_1. If there is an isomorphic match m, then the matched rule is deleted by means of a second-order rewriting. Otherwise, the rewriting behaves like a no-operation, and D_1 is kept unchanged. We have tagged with the number 3 all operations that modify the cardinality of the active rules in D_1 to distinguish them from rule modifications induced by second-order rewriting. Another distinct is that, instead of marking the rewriting by means of the respective 2-rule, we keep the create-delete statement in its place. This is without loss of generality, since the creation or deletion 2-rule is totally defined by the respective rewriting statement.

Definition 123 (Create-delete rewriting sequence)**.** *Given a create-delete sequence* $s = s_1, s_2, \ldots, s_n$ *and a starting first-order grammar* $\mathcal{G}_0$*, we define a create-delete rewriting sequence induced by* s *over* $\mathcal{G}$*, written* $CD(\mathcal{G}_0, s)$*, as*

$$\mathcal{G}_0 \xRightarrow{2,i_1,s_1,m_1} \mathcal{G}_1 \xRightarrow{2,i_2,s_2,m_2} \mathcal{G}_2 \xRightarrow{2,i_3,s_3,m_3} \mathcal{G}_3 \quad \cdots \quad \mathcal{G}_{n-1} \xRightarrow{2,i_n,s_n,m_n} \mathcal{G}_n$$

Create-delete rewriting sequences allow the representation of a finite number of modifications on coproduct collections. From this, we introduce *create-delete-modify simple second-order graph grammars*, as follows.

Definition 124 (Create-delete-modify simple second-order graph grammar)**.** *A create-delete-modify simple second-order graph grammar (CDM-S-SOGG) is a tuple* $(T, D_0, D_1, D_2, \eta_2, C_2)$ *where*

- $(T, D_0, D_1, D_2, \eta_2)$ *is a S-SOGG,*
- C_2 *is a T-typed create-delete sequence.*

Definition 125 (Derivation of CDM-S-SOGG)**.** *Let* $\mathcal{G}^2$ *be a CDM-S-SOGG* $(T, D_0, D_1, D_2, \eta_2, C_2)$ *such that*

$$\mathcal{G}_0 = (T, D_0, D_1)$$

,

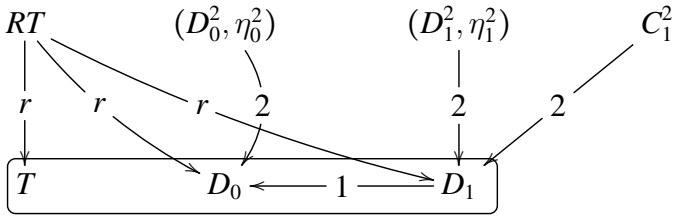
$$CD(\mathcal{G}_0, C_2) = \mathcal{G}_0 \xRightarrow{2,i_1,s_1,m_1} \mathcal{G}_1 \xRightarrow{2,i_2,s_2,m_2} \mathcal{G}_2 \xRightarrow{2,i_3,s_3,m_3} \mathcal{G}_3 \quad \cdots \quad \mathcal{G}_{n-1} \xRightarrow{2,i_n,s_n,m_n} \mathcal{G}_n$$

and

$$\mathcal{G}_n = (T, D_0, D_1')$$

We define the derivations of $\mathcal{G}^2$ *to be all derivations of the S-SOGG* $(T, D_0, D_1', D_2, \eta_2)$ *(i.e., same second-order layer* (D_2, η_2) *but starting with the first-order grammar* $\mathcal{G}_n$*) prefixed by the derivation* $CD(\mathcal{G}_0, C_2)$*.*

The previous definitions account for specifications with an extra sequence C_2 containing creation and deletion statements. Those creations and deletions are to be executed before the application of any 2-rule in D_2. This idea allows us to introduce and remove rules in the first-order specification (T, D_0, D_1). We can also integrate this idea with other models, giving rise, for instance, to CDM version of C-SOGGs (we omit this definition because they can straightforwardly derived from the previous one). We can introduce retyping for models with create-delete sequences in the same way as before: we just need to be careful to specify that create-delete sequences are to be executed after retyping, i.e., in the associated retyped CDM-S-SOGG or CDM-C-SOGG. For the last example, we will consider a full CDM-RC-SOGG model (i.e., create-delete-modify retyping-aware complete second-order graph grammar). In this model, the affect of the second-order layer over the first-order components can be represented in the following diagram (where C_1^2 represents the create-delete sequence).



Example 126 (CDM-RC-SOGG)**.** *The first-order system we are considering in our examples up to now, depicted originally in Figure 2.7, has some issues regarding the deletion of messages. The first is that is possible for the rule* getDATA *to be applied more than once, loading several data items on the same message node. When this message is sent back to the client by means of* receiveMSG*, it is impossible to delete it with* deleteMSG *because of*

the extra data items connected to the message node and the DPO dangling condition. The second point is that, once the message returns to the client, there is nothing that forbids it to be sent again to collect more data, instead of unloading its data and being erased. In order to correct those issues, we can design a second-order layer that performs the following actions:

1. *create a new self-edge type, to mark new messages that have not been sent to servers yet, and add one self-edge of this type to each message in the initial graph. These edges must be deleted by the application of rule* sendMSG.

2. *add a new rule that would transfer the data from a received message into the client,* without deleting the message. *This new rule, named* unloadMSG, *transfers to the client all but the last data node of the message, allowing* deleteMSG *to handle the last one.*

The corrections may be represented by a create-delete-modify retyping-aware complete second-order graph grammar (CDM-RC-SOGG). The specification is shown if Figure 5.8. The retyping span adds a new self-edge type to message nodes. The only rule addMark *in the initial graph preparation layer* (D_0^2, η_0^2) *adds a self-edge instance to every message node instance. The only 2-rule* addMarkSend *in the rule modification layer* (D_1^2, η_1^2) *modifies the first-order rule* sendMSG *to make it test and delete message self-edges. Finally, there is a unique operation in the create-delete layer* C_1^2 *specifying that* unloadMSG *must be added to the collection of first-order rules. The corrected system only sends messages once, but allow them to collect an arbitrary number of data items.*

This last example presents all kinds of modifications we can do simultaneously over a first-order specification: we are changing its type graph, the initial condition, and the collection of rules (both by adding a new rule and modifying existing ones). By a matter of design, modifications in both graph and rules are performed through rule-based rewriting.

5.4.5 Summary of models

In this section we have arrived at several so-called second-order models that differ mainly in how they affect the first-order components. The names of the models attempt to be descriptive, and in order to summarize their characteristics we present them in the following table:

Models / Modifications	Retype(D_0)	D_0	Retype(D_1)	D_1	Cardinality(D_1)
S-SOGG				$\surd$	
C-SOGG		$\surd$		$\surd$	
RS-SOGG	$\surd$		$\surd$	$\surd$	
RC-SOGG	$\surd$	$\surd$	$\surd$	$\surd$	
CDM+	$=$	$=$	$=$	$=$	$+\surd$

Roughly speaking, *simple* models are the ones which only modify the rules but not the initial graph(s), while *complete* models affect both the initial graph(s) and the collection of rules. *Retyping-aware* are the models that allow modifications in the type graph, and retyping has a conservative effect: it deletes all instances of the deleted types, but does not add instances of the created types. This notion of retyping is traditional in the graph transformation area. Finally, *create-delete-modify* models are equiped with a fixed

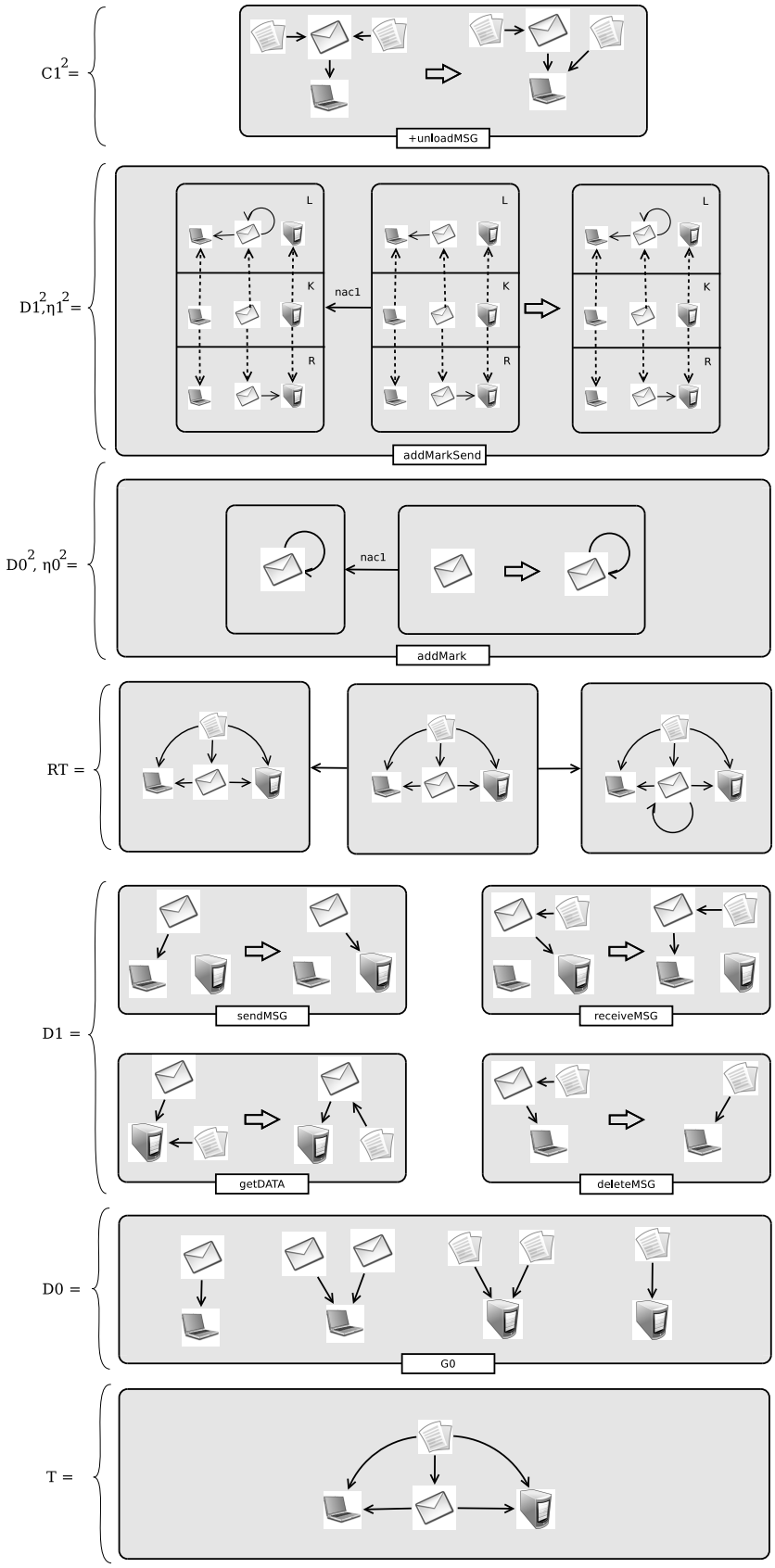


Figure 5.8: Create-delete-modify retyping-aware complete second-order graph grammar correcting issues of the base system.

sequence of deletion and creation of first-order rules, modifying the cardinality of the original collection of rules of the first-order system. The most generic model would be a *create-delete-modify retyping-aware complete second-order graph grammar*, abbreviated as CDM-RC-SOGG, for which all other models are special cases. We can see an example of such system in Figure 5.8.

5.5 Execution strategies for second-order graph grammars

Now that we have defined several kinds of second-order graph grammars and their respective derivations, we can discuss their respective execution model. As with the original graph grammar models, we can take their collection of (concrete) derivations as the simplest semantic model of a SOGG. Notice, however, that there are now distinct kinds of one-step derivations, tagged by the first number n of their label (n, x, y, z).

The simplest execution model of a conventional graph grammar comprises of a non-deterministic choice of all possible rewritings available at a given state. Since the state in SOGGs is a whole first-order system $\mathcal{G} = (T, D_0, D_1)$, there may be derivations tagged by 1 or 2 from it, and we need to specify if non-deterministic choice is applied uniformly across both tags, or if there will be some sorte of *priority* between them. This choice may depend on the intended use of the model. For instance, if we consider the 2 layer to represent a model transformation over the initial first-order system, we expect that 2-tagged rewriting steps have priority over any 1-tagged derivation. Additionally, in this case it is important that the 2-tagged possible rewritings terminate. On the other hand, if we are modelling a system with dynamic self-modifying rules, interleaving between 1-tagged and 2-tagged derivations are actually desired. A third possible case would be to only apply 2-tagged derivations if there are no possible 1-tagged derivations possible. In this case, the second-order layer would represent some kind of recovery modification on rules that would restore the system to a working status in case for some reason its execution gets stuck.

Because our main application scenario is the representation of model transformations by means of 2-tagged derivations, the notion of priority comes naturally.

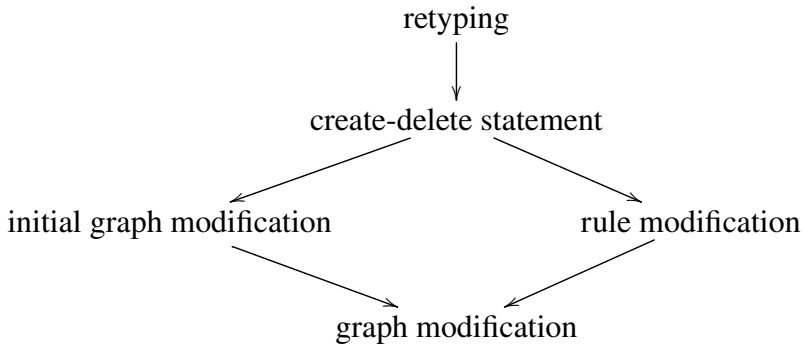
Definition 127 (Scheduling of SOGGs)**.** *Let $\mathcal{G}^2$ be a SOGG graph grammar (any of the varitions). We define the following scheduling types:*

Non-deterministic: *no priority between* 1*-tagged and* 2*-tagged derivations.*

Priority: 2*-tagged derivations have priority over* 1*-tagged ones.*

Reverse priority: 1*-tagged derivations have priority over* 2*-tagged ones.*

In the models of the previous section we have defined that retyping is performed before any derivation, and creation-deletion statements are always executed before any modification statements. Due to this choice, if we assume *priority scheduling* for the execution of a given model, we arrive at the following scheme, where the highest elements have higher priority.



However, we will not fix any of the possible scheduling as definitive since, as mentioned before, SOGGs may be used to model distinct scenarios requiring distinct scheduling policies. Regarding the derivation layer tags, one of the advantages of their use is that we can characterize very straightforwardly which derivations correspond to *model transformations*, as shown in the next definition.

Definition 128 (Model-transformation derivation)**.** *Let $\mathcal{G}^2$ be a SOGG (any of the models), and let ρ be a (possibly infinite) derivation*

$$\mathcal{G}_0 \overset{(n_0,x_0,y_0,z_0)}{\Longrightarrow} \mathcal{G}_1 \overset{(n_1,x_1,y_1,z_1)}{\Longrightarrow} \mathcal{G}_2 \overset{(n_2,x_2,y_2,z_2)}{\Longrightarrow} \dots$$

of $\mathcal{G}^2$. We say ρ is a model-transformation derivation *iff*

1. *for all $i \in \mathbb{N}$, $n_i \geq n_{i+1}$ and*

2. *there is a state $\mathcal{G}_k \in \rho$ such that, for all possible derivations $\mathcal{G}_k \overset{(j,x_k,y_k,z_k)}{\Longrightarrow} \mathcal{G}'$) we have $j = 1$ (they are all 1-tagged).*

We name $\mathcal{G}_k$ the evolved system *of ρ.*

Model-transformation derivations correspond to the execution of a programmed transformation over a first-order system immediately followed by the execution of the resulting system. Note that the definition of model-transformation derivation is independent of the scheduling of the model, although all model-transformation derivations respect the ordering of the priority scheduling.

Example 129. *We show in Figure 5.9 a model transformation derivation*

$$\rho = \mathcal{G}_0 \overset{(2,t1,\mathsf{addLog},m1)}{\Longrightarrow} \mathcal{G}_1 \overset{(2,t3,\mathsf{addLog},m2)}{\Longrightarrow} \mathcal{G}_2 \overset{(2,t2,\mathsf{addLog},m3)}{\Longrightarrow} \mathcal{G}_3 \overset{(1,g0,t1,m4)}{\Longrightarrow} \mathcal{G}_4 \overset{(1,g0,t2,m5)}{\Longrightarrow} \mathcal{G}_5$$

of the RS-SOGG depicted in Figure 5.6 (place-transition net example as second-order graph grammar). We have three 2-tagged derivations, representing the execution of 2-rule addLog *to each of the original rules of the grammar. The evolved system of ρ is $\mathcal{G}_3$, because due to application conditions we cannot apply* addLog *any more. We have yet two 1-tagged derivations at the end of ρ representing two initial execution steps of the evolved system $\mathcal{G}_3$.*

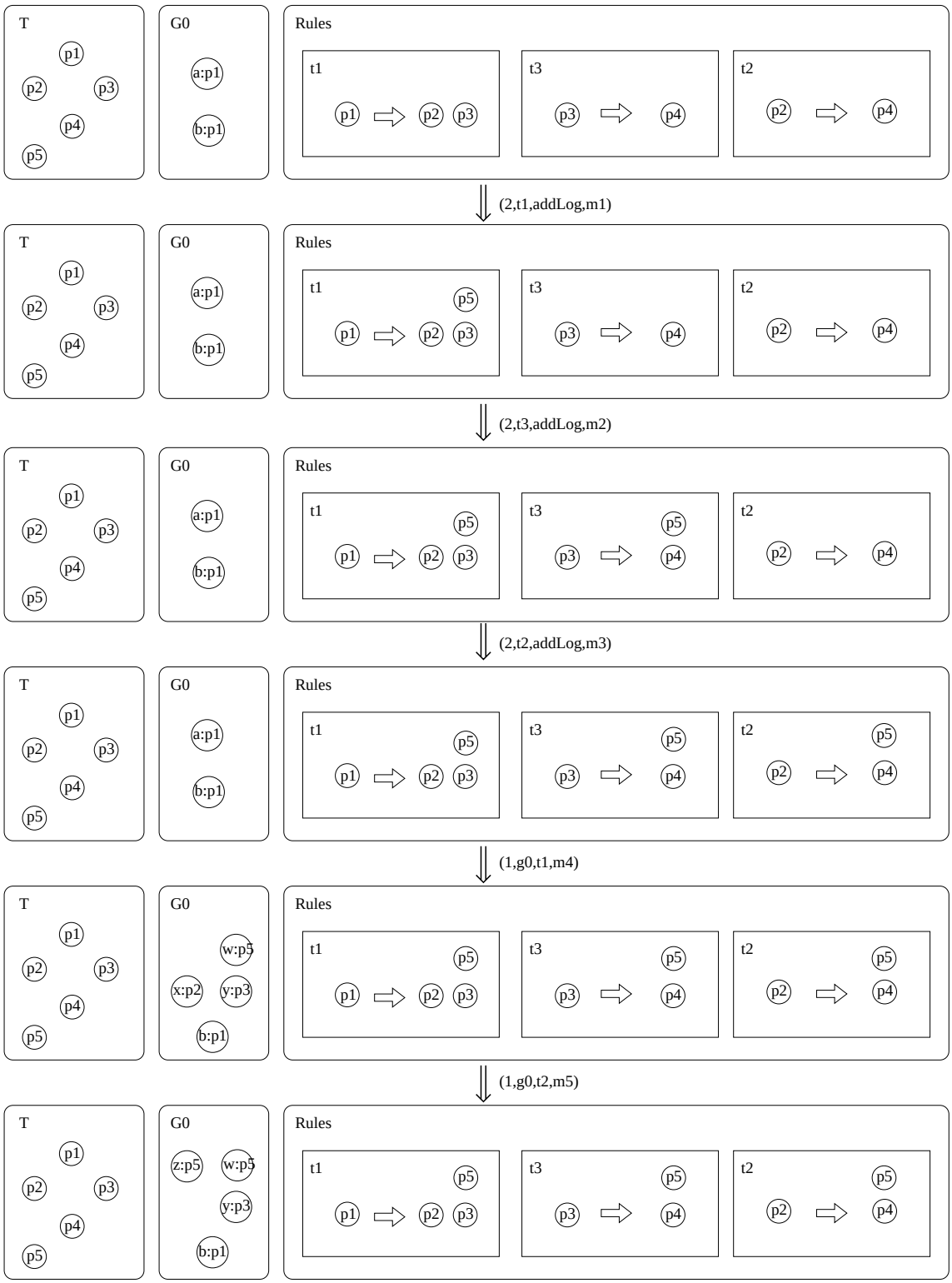
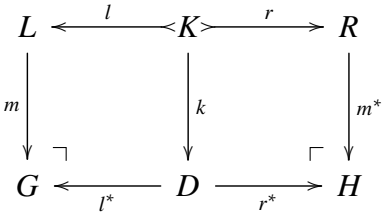


Figure 5.9: Model-transformation derivation of the place-transition system with log.

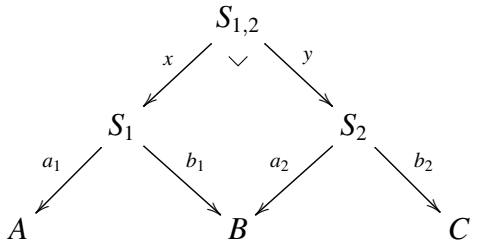
5.6 Model evolution represented by spans

One of the characteristics of the DPO approach for graph rewriting is that there is an explicit *interface graph* in rules, i.e., within each graph rule $L \leftarrow K \rightarrow R$ the graph K denotes the matched elements that are not modified by the rule application. Considering an application of the rule as the DPO diagram shown below, where we notice that between graphs G and H there is a span $s = G \xleftarrow{l^*} D \xrightarrow{r^*} H$.



The graph D consists of all elements maintained by the rule application, either by being left out of the match m (i.e., out of the region $m(L)$ in G) or by being preserved by the rule (within $l \circ m(K)$ in G). This span represents the overall evolution of the graph G into H, without referring to the particular rule that induced this evolution. For instance, given only s we could not know the rule that produced it, since K could be either empty, isomorphic to D or some graph in between. The span s records precisely what has been removed from G (represented by $(G - l^*(D))$) and what has been added to D (represented by $(H - r^*(D))$) by the rule application. In order to specify what has been added and deleted by a whole derivation composed of a series of subsequent rewritings, we need to recall the standard categorial notion of *span composition*.

Definition 130 (Span composition)**.** *Let $s_1 = A \xleftarrow{a_1} S_1 \xrightarrow{b_1} B$ and $s_2 = B \xleftarrow{a_2} S_2 \xrightarrow{b_2} C$ be two spans in a category $\mathscr{C}$ with pullbacks. The composition span $s_1 \star s_2$ of the two spans is obtained by calculating the pullback of b_1 and a_2 and taking $s_1 \star s_2 = A \xleftarrow{x;a_1} S_{1,2} \xrightarrow{y;b_2} C$, as shown in the diagram below.*



Since pullback calculation is associative, from a sequence of consecutive spans $t = s_1; s_2; \ldots; s_n$ where $n \geq 2$, we obtain a composition of the whole sequence as

$$\star(t) = (\ldots((s_1 \star s_2) \star s_3) \ldots \star s_n)$$

5.6.1 Evolutionary spans for models without retyping

Suppose we have a sequence of graph rewritings $G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} G_2 \ldots G_n$ from which we obtain the spans $s_1 = G_0 \xleftarrow{l_1^*} D_1 \xrightarrow{r_1^*} G_1$, $s_2 = G_1 \xleftarrow{l_2^*} D_2 \xrightarrow{r_2^*} G_2$, …, $s_n = G_{n-1} \xleftarrow{l_n^*} D_n \xrightarrow{r_n^*} G_n$ taking the base of each DPO diagram. By means of span composition we obtain a final span $\star(s_1; \ldots; s_n) = G_0 \leftarrow S_{1,\ldots,n} \rightarrow G_n$ representing the overall modification from a derivation (of any length) between G_0 and G_n. That is the intuition for our next definitions,

with the distinction that, instead of graphs, we will consider first-order graph grammars as objects. Given a model-transformation derivation, we will be able to track what has been removed, maintained and added to the original system in the same way as the case of graphs. For such, we need to extend the notion of span representing evolution from plain graphs towards a complete first-order graph grammar $\mathcal{G} = (T, D_0, D_1)$. For reason of simplicity, we will define evolution initially for the S-SOGG model, keeping in mind that definitions for the other models that are not retyping-aware can be obtained by extension.

Definition 131 (Span between coproduct collections)**.** *Let $c_1 = \{i_k\}_{k\in\mathbb{N}}$ and $c_2 = \{j_k\}_{k\in\mathbb{N}}$ be coproduct collections in a category $\mathscr{C}$. A coproduct collection span is a collection of spans $\{s_k : dom(i_k) \leftarrow d_k \rightarrow dom(j_k)\}_{k\in\mathbb{N}}$ in $\mathscr{C}$. An identity coproduct collection span is a coproduct collection span where all morphisms of all component spans s_k are identities in $\mathscr{C}$. Composition of coproduct collection spans is also written $\star$ and it is given by component-wise composition of the spans between the domains of injections.*

Definition 132 (Span between first-order graph grammars)**.** *Let $\mathcal{G}_1 = (T, D_0, D_1)$ and $\mathcal{G}_2 = (T, D_0', D_1')$ be first-order graph grammars. A first-order graph grammar span is a pair (s_0, s_1) where*

- *s_0 is a coproduct collection span between D_0 and D_0';*
- *s_1 is a coproduct collection span between D_1 and D_1';*

An identity (first-order) graph grammar span is a pair (s_0, s_1) where both components are identity coproduct collection spans. Composition of graph grammar spans is also denoted by $\star$ and it is defined by component-wise composition of the two coproduct collection spans.

Now that we have defined what means a span between coproduct collections and graph grammars, we can state how to obtain an *evolutionary span* from second-order rewritings, coproduct rewritings and S-SOGG derivations.

Definition 133 (Evolutionary span from one-step rewriting)**.** *We define how to obtain evolutionary spans from the following kinds of rewriting:*

first-order rewriting $G_1 \overset{p,m}{\Longrightarrow} G_2$*: we take the base of the DPO diagram in T-***Graph** *as the evolutionary span.*

second-order rewriting $p_1 \overset{\alpha,m}{\Longrightarrow} p_2$*: we take the base of the DPO diagram in T-***Span** *as the evolutionary span.*

coproduct collection rewriting $D_x \overset{inj_k,r,m}{\Rightarrow} D_x'$ *(in some category $\mathscr{C}$): the evolutionary span is $\{s_k\}_{k\in\mathbb{N}}$ where*

- *if $n = k$, then s_n is the base of the DPO diagram $dom(inj_n) \overset{r,m}{\Longrightarrow} a$ (which induced the coproduct collection rewriting)*
- *if $n \neq k$, the s_n is the identity span $dom(inj_n) \overset{id}{\leftarrow} dom(inj_n) \overset{id}{\rightarrow} dom(inj_n)$.*

As an example, given the coproduct rewriting $D_1 \overset{inj_2,\alpha,m}{\Rightarrow} D_1'$ such that $dom(inj_2) = r_2$, $\mathscr{A}(D_1) = \{r_1, r_2\}$ and $\mathscr{A}(D_1') = \{r_1, r_2''\}$, we can see in the diagram of Figure 5.10 a representation of its evolutionary span together with the rewriting that induced it.

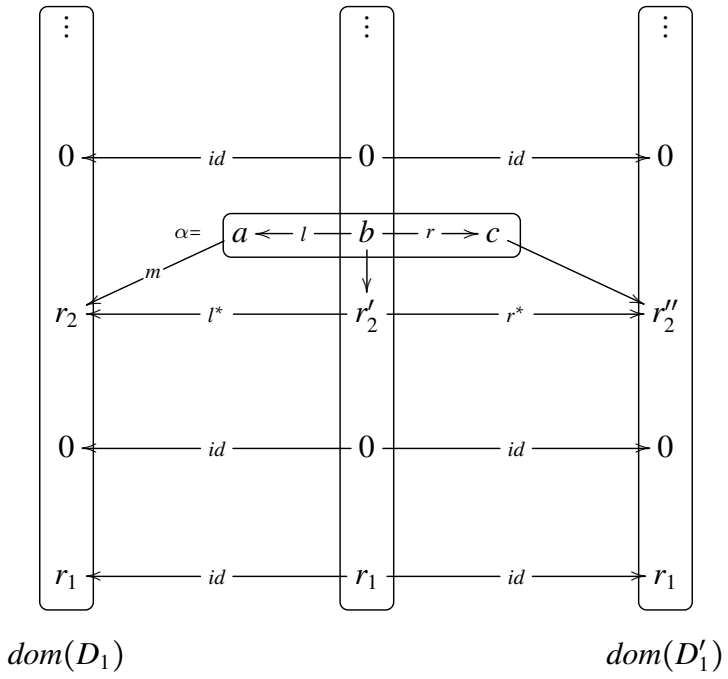


Figure 5.10: Evolutionary span between coproduct collections of rules D_1 and D'_1.

one-step derivation of S-SOGG $\mathcal{G}_1 \overset{(n,x,y,z)}{\Longrightarrow} \mathcal{G}_2$, *where* $\mathcal{G}_1 = (T, D_0, D_1)$ *and* $\mathcal{G}_2 = (T, D'_0, D'_1)$*: the respective evolutionary span* (s_0, s_1) *between* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *is*

- *case* $n = 1$ *(first-order):* s_0 *is the evolutionary span induced by the coproduct rewriting* $D_0 \overset{x,dom(y),z}{\Rrightarrow} D'_0$*, and* s_1 *is the identity span.*
- *case* $n = 2$ *(second-order):* s_0 *is the identity span and* s_1 *is the evolutionary span induced by the coproduct rewriting* $D_1 \overset{x,dom(y),z}{\Rrightarrow} D'_1$.

This last definition states how to construct an evolutionary span from a given one-step derivation $\mathcal{G}_1 \overset{(n,x,y,z)}{\Longrightarrow} \mathcal{G}_2$ of a simple second-order graph grammar. Given a larger derivation, we can calculate the overall evolutionary span by graph grammar span composition, which is obtained component-wise. This way we arrive at a single *span between graph grammars* representing the evolution of the whole derivation.

Definition 134 (Evolutionary span of S-SOGG derivations)**.** *Let*

$$\rho = \mathcal{G}_1 \overset{(n_1,x_1,y_1,z_1)}{\Longrightarrow} \mathcal{G}_2 \overset{(n_2,x_2,y_2,z_2)}{\Longrightarrow} \mathcal{G}_3 \overset{(n_3,x_3,y_3,z_3)}{\Longrightarrow} \ \ldots \ \mathcal{G}_n$$

be a finite derivation of a given S-SOGG, and let t *be the sequence of graph grammar spans*

$$t = \quad \mathcal{G}_1 \xleftarrow{} \mathcal{D}_1 \xrightarrow{} \mathcal{G}_2 \xleftarrow{} \mathcal{D}_2 \xrightarrow{} \mathcal{G}_3 \quad \ldots \quad \mathcal{G}_{n-1} \xleftarrow{} \mathcal{D}_{n-1} \xrightarrow{} \mathcal{G}_n$$

induced, respectively, by each one-step derivation of ρ*. We name the graph grammar span* $\mathcal{G}_1 \overset{m}{\leftarrow} \mathcal{D} \overset{n}{\rightarrow} \mathcal{G}_n$ *resulting from* $\star(t)$ *the* evolutionary span of ρ.

If ρ is a model-transformation derivation, we have an *evolved system* $\mathcal{G}_k$ for some $k \leq n$. By calculating the evolutionary span between the original first-order system $\mathcal{G}_1$ and $\mathcal{G}_k$, we have a formal and precise account of everything that has been maintained, removed and added by the subderivation

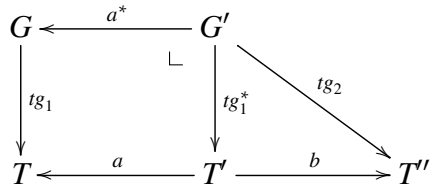
$$\mathcal{G}_1 \overset{(n_1,x_1,y_1,z_1)}{\Longrightarrow} \mathcal{G}_2 \overset{(n_2,x_2,y_2,z_2)}{\Longrightarrow} \mathcal{G}_3 \quad \ldots \quad \mathcal{G}_k$$

This procedure for calculating evolutionary spans applies to models S-SOGG, C-SOGG, CDM-S-SOGG and CDM-C-SOGG. Notice that we do not require any modification in the definition to obtain evolutionary spans from derivations of *create-delete-modify* models because of the characterization of creation and deletion of rules as particular cases of coproduct rewriting. However, retyping is not being covered by this notion of evolution, since the derivations of a retyping-aware model are the ones of the respective retyped model, which start with the retyped version of the original first-order system. Because retyping may delete instances, we want to consider those possible deletions in our notion of evolutionary span, as the next section details.

5.6.2 Evolutionary spans for models with retyping

This section deals with the calculation of evolutionary spans for models with retyping. As before, we will trace a parallel with a similar situation with conventional graphs, and then generalize towards first-order graph grammars. We take the model RS-SOGG as an example, for which suitable definitions for the other models may be directly derived.

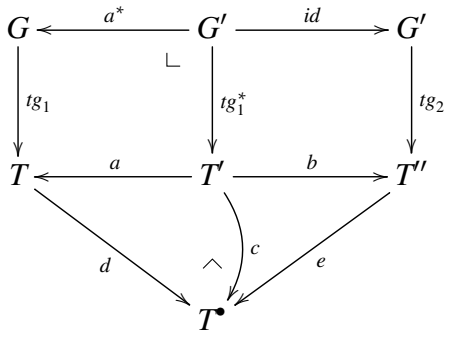
Recalling the previous sections, we can represent the retyping operation induced by the span $s = T \overset{a}{\leftarrow} T' \overset{b}{\rightarrow} T''$ over the T-typed graph $tg_1 : G \rightarrow T$, by the following diagram in the category **Graph**



generating the T''-typed graph $tg_2 : G' \rightarrow T''$. In the retyping-aware models, this principle is used to modify the initial collection of graphs and rules in order to fit them into the type T''. The problem now is how to relate tg_1 with tg_2, since the first lies in T-**Graph** and the second, in T''-**Graph**. Moreover, the retyping of tg_1 may have deleted some elements of G during the calculation of G', which is represented in the diagram by the morphism a^*. One solution for those issues is to calculate the pushout of the span $s = T \overset{a}{\leftarrow} T' \overset{b}{\rightarrow} T''$, which creates a type $T^\bullet$ consisting of the elements of T' together with the additions represented by both a and b. Using the extended type graph $T^\bullet$, we can define the evolutionary span of the retyping induced by s.

Definition 135 (Evolutionary span from graph retyping)**.** *Let $tg_1 : G \rightarrow T$ be an T-typed graph, and $s = T \overset{a}{\leftarrow} T' \overset{b}{\rightarrow} T''$ be a monic span in* **Graph**. *We define the respective* evolutionary span from graph retyping *by the $T^\bullet$-typed span $t = tg_1; d \overset{a^*}{\leftarrow} tg_1^*; c \overset{id}{\rightarrow} tg_2; e$*

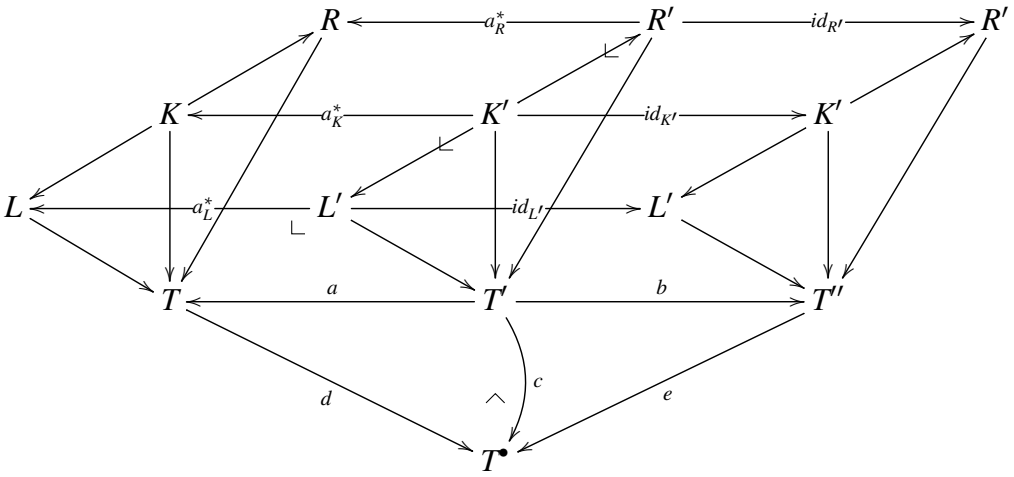
shown in the diagram below in the category **Graph**.



where $PO(s) = T \xrightarrow{d} T^{\bullet} \xleftarrow{e} T''$.

We can define a similar notion of evolutionary span from rule retyping by considering monic graph span instead of graphs in the definition.

Definition 136 (Evolutionary span from rule retyping)**.** *Let* $p_1 = L \leftarrow K \rightarrow R$ *be an T-typed graph rule, and* $s = T \xleftarrow{a} T' \xrightarrow{b} T''$ *be a monic span in* **Graph**. *We define the respective* evolutionary span from rule retyping *by the* $T^{\bullet}$*-typed span of rules* $d^{\triangleright}(p_1) \xleftarrow{a^*} c^{\triangleright}(p_1') \xrightarrow{id} e^{r}hd(p_2)$ *as shown in the diagram below in the category* **Graph**,

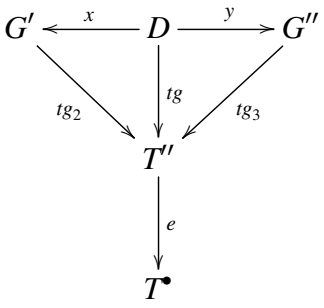


where $PO(s) = T \xrightarrow{d} T^{\bullet} \xleftarrow{e} T''$, p_1' *is a* T'*-typed graph rule with graph instances* $L' \leftarrow K' \rightarrow R'$ *and* p_2 *is a* T''*-typed graph rule with graph instances* $L' \leftarrow K' \rightarrow R'$.

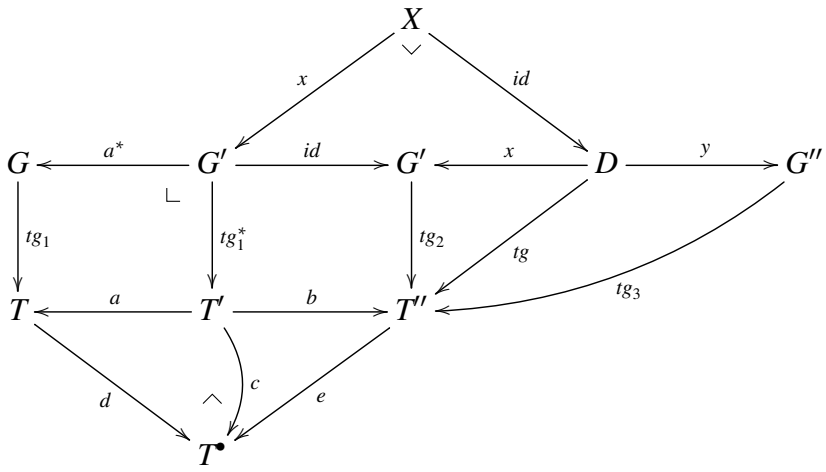
We can extend the notion of evolutionary span from graph retyping and rule retyping towards coproduct collections of graphs and rules in a component-wise way. It is known that retyping functors induced by graph spans *preserve colimits*, and, consequently, coproducts. This means that if we have a coproduct collection c over type T, $s^{\triangleleft\triangleright}(c)$ is still a coproduct collection over type T''.

Now, let us return to the case of graphs for the sake of the example. Let us suppose we have a T''-typed evolutionary span $u = tg_2 \xleftarrow{x} tg \xrightarrow{y} tg_3$ obtained from a derivation ρ. If we

retype this span using the functor $e^{\triangleright} : T''$-**Graph** $\rightarrow T^{\bullet}$-**Graph** as shown below,



we obtain a $T^{\bullet}$-typed span that may be composed with the evolutionary span obtained from retyping. By composing both the retyping evolutionary span and the rewriting evolutionary span, we end up with a span representing the overall modification that we are applying over the original first-order structure. This operation is depicted in the diagram below.



The base for calculating evolutionary spans for retyping-aware models is based on this particular situation with graphs, but generalized towards graph grammar spans.

Definition 137 (Evolutionary span of RS-SOGG derivations)**.** *Consider a RS-SOGG* $R = (T, D_0, D_1, RT, D_2, \eta_2)$ *such that* $RT = T \xleftarrow{a} T' \xrightarrow{b} T''$ *and the respective retyped S-SOGG is* $R_{RT} = (T'', D_0'', D_1'', D_2, \eta_2)$*. Let us name* $\mathcal{G}_0 = (T, D_0, D_1)$ *and* $\mathcal{G}_1 = (T'', D_0'', D_1'')$*. Given a* T''*-typed finite derivation*

$$\rho = \mathcal{G}_1 \overset{(n_1,x_1,y_1,z_1)}{\Longrightarrow} \mathcal{G}_2 \overset{(n_2,x_2,y_2,z_2)}{\Longrightarrow} \mathcal{G}_3 \overset{(n_3,x_3,y_3,z_3)}{\Longrightarrow} \ \ldots \ \mathcal{G}_n$$

of R_{RT}*, we execute the following steps to obtain a final evolutionary span:*

1. *calculate the pushout* $PO(s) = T \xrightarrow{d} T^{\bullet} \xleftarrow{e} T''$*;*
2. *calculate the* $T^{\bullet}$*-typed evolutionary span from retyping of* $\mathcal{G}_0$ *into* $\mathcal{G}_1$*, and name it* u*;*
3. *calculate the* T''*-typed evolutionary span from derivation* ρ*, and name it* v_0*.*
4. *calculate the* $T^{\bullet}$*-typed evolutionary span* $v = e^{\triangleright}(v_0)$

The resulting evolutionary span of R is the composition of $T^{\bullet}$*-typed graph grammar spans* $u \star v$*.*

This last definition tell us how to compose the span obtained from a given derivation in the retyped model with the possible changes induced by the rewriting process. This process calculates the evolutionary spans for models RS-SOGG, RC-SOGG, CDM-RS-SOGG and CDM-RC-SOGG.

Example 138 (Evolutionary span)**.** *In Figure 5.11 we present the evolutionary span calculated from the derivation depicted in Figure 5.9. Because the derivation only adds elements to the specification, the interface of the span corresponds to the initial graph grammar $\mathcal{G}_0$. The right-hand side corresponds to the final first-order system, namely $\mathcal{G}_3$ in the original derivation. Notice that the type graph of evolutionary span is the pushout of the retyping span of the original specification, which contains the node type p_5.*

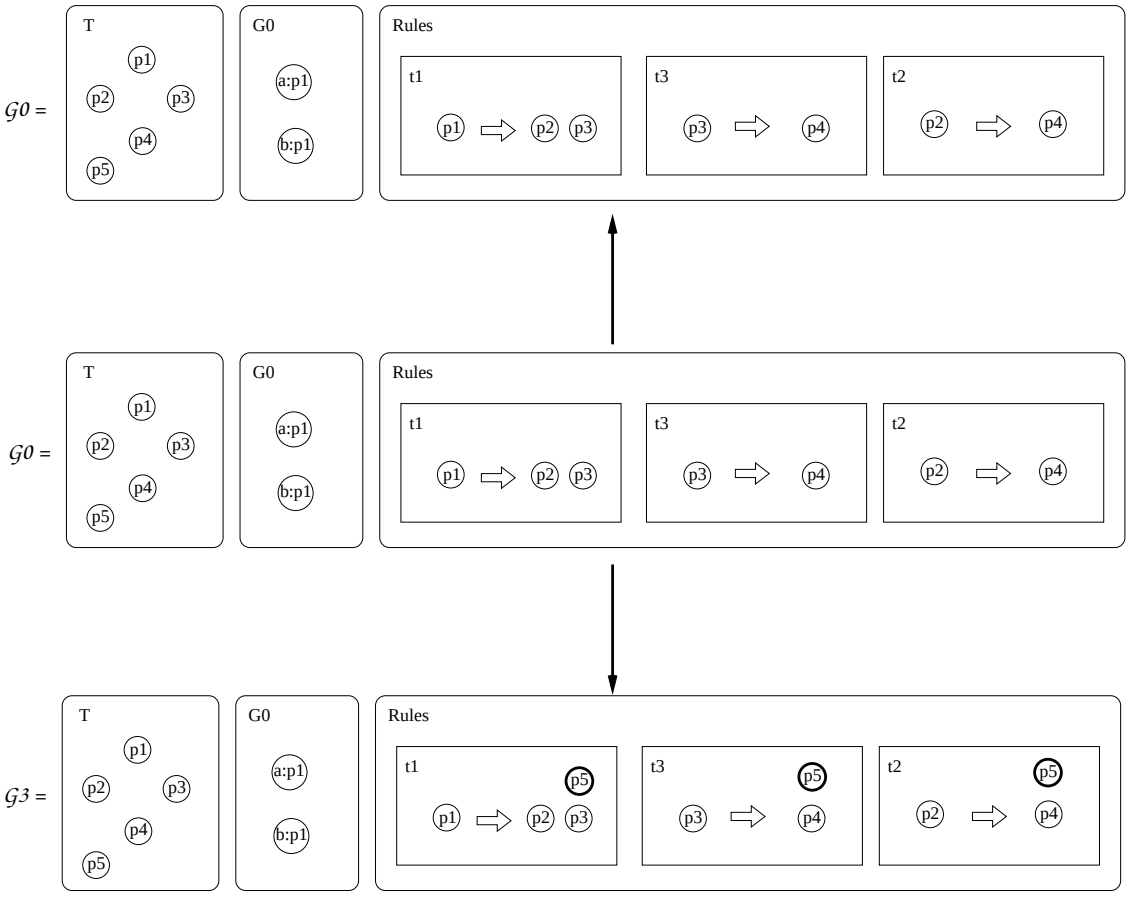


Figure 5.11: Example of evolutionary span.

5.7 Summary

The objective of this chapter is the definition of graph grammar specifications based on both first- and second-order graph rewriting. We started with a discussion about how to represent collections of rules, and how to modify the rule collection based on the application of a given 2-rule. This discussion gave rise to the notion of *coproduct collection* and *coproduct rewriting*. Based on this representation, we redefined first-order graph grammar, since they will comprise the state to be modified in second-order graph grammars. Then we introduced several kinds of second-order graph grammars, which differ mainly

on the ability to affect specific parts of the first-order system. After the presentation of the models, we discuss their operational semantics focusing on the presence or absence of *priority* between derivations of higher or lower order. We also defined what we have called *model-transformation derivations*, which represents a terminating model transformation followed by the execution of the *evolved system*. Finally, we introduced the notion of evolutionary span that registers what exactly has been added, removed and preserved from a given specification by means of 2-tagged derivations.

6 INTER-LEVEL INTERACTION

In Chapter 5 we have defined visual models equipped with two levels of rewriting: first-order and second-order. One of the advantages of our approach is the fact that we can represent both first-order and second-order rewritings in the same context, i.e, as as diagrams in the same category T-**Graph**. This allows us to represent their interaction by means of elements (morphisms) in this category. This chapters deals particularly with a notion of conflict and dependence between second-order and first-order rewritings. Based on this definitions, we propose an extension of critical pair analysis for second-order graph grammars. Finally, given a particular model-transformation, represented by a derivation, we discuss how to relate first-order critical pairs of the original system with first-order critical pairs of the modified system. This relationship allows us to foresee the effect of the model transformation over conflicts and dependencies of the original specification, which affect decisively its semantics. Hence, the original contributions of this chapter are the following:

- definition of a notion of conflict between first-order and second-order rewritings, and its extension to derivations of second-order graph grammars.
- an extension of the critical pair analysis technique for second-order graph grammars.
- determination of the evolution of critical pairs of the base system due to model transformations.

6.1 Inter-level conflicts and dependencies

In this section we determine how to describe conflicts and dependencies between second-order and first-order rewritings of a second-order specification. By the sake of simplicity, we will consider our analysis on simple second-order graph grammars (S-SOGG). We claim that results for the remaining models can be easily extrapolated from the interactions we foresee in this case.

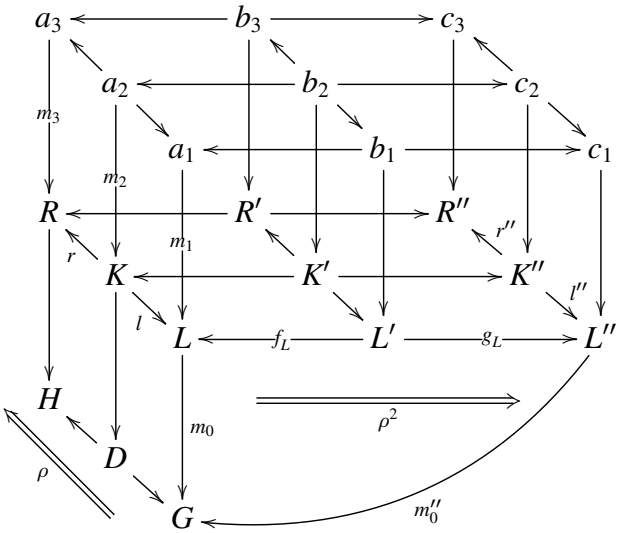
We start considering a S-SOGG $(T, D_0, D_1, D_2, \eta_2)$ and an arbitrary state of it, i.e., a first-order graph grammar $\mathcal{G} = (T, D_0', D_1')$. If we take two arbitrary one-step derivations departing from $\mathcal{G}$, we can have the following combinations.

1. two first-order rewritings: $\mathcal{G}_1 \xleftarrow{(1,g_1,r_1,m_1)} \mathcal{G} \xrightarrow{(1,g_2,r_2,m_2)} \mathcal{G}_2$
2. two second-order rewritings: $\mathcal{G}_1 \xleftarrow{(2,r_1,\alpha_1,m_1)} \mathcal{G} \xrightarrow{(2,r_2,\alpha_2,m_2)} \mathcal{G}_2$
3. one first-order, one second-order rewriting: $\mathcal{G}_1 \xleftarrow{(1,g_1,r_1,m_1)} \mathcal{G} \xrightarrow{(2,r_2,\alpha_2,m_2)} \mathcal{G}_2$

For each case, we will obtain a distinct characterization of conflict. In the two first cases, the notion of conflict is derived from conflicts of graph rewriting and span rewriting (with NACs), respectively. The third case is new, and requires us to ponder about what exactly means a conflict between a second-order and a first-order modification. We will address this issue now, and later we will provide a complete description for conflicts in all three cases.

Initially, we observe that there is a intrinsic hierarchy between the two levels of rewriting. Since first-order rewritings do not modify the collection of rules in any sense, it is quite obvious that they cannot interfere in the execution of any second-order rewritings. This causes the nature of possible inter-level conflicts to be, by definition, asymmetric: from higher-order to lower-order. A second observation refers to what a conflict represents. Consider that we are in the third case (one first-order and one second-order derivation) and we have $r_1 = r_2$. This means that the rule $dom(r_1)$ being used to modify g_1 is the same one being modified by 2-rule $dom(\alpha_2)$. Since conflicts are defined by one of the derivations disabling the other, we may say that there is a conflict between $\mathcal{G} \overset{(1,g_1,r_1,m_1)}{\Longrightarrow} \mathcal{G}_1$ and $\mathcal{G} \overset{(2,r_2,\alpha_2,m_2)}{\Longrightarrow} \mathcal{G}_2$ if the latter in some way *forbids* the application of the former, given that we maintain as much as possible the instantiation provided by the match m_1. In other words, the notion of conflict between higher-order and lower-order refers to *preservation of applicability* for a given rule $dom(r_1)$ and match m_1 after the possible modifications done by $dom(\alpha_2)$ and m_2. Representing the situation diagrammatically, we arrive at the following definition of independence.

Definition 139 (Inter-level parallel independence)**.** *Let $\rho = G \overset{r,m_0}{\Longrightarrow} H$ and $\rho^2 = r \overset{\alpha,m}{\Longrightarrow} r''$ (where $m = (m_1, m_2, m_3)$, $r = L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ and $r'' = L'' \overset{l''}{\leftarrow} K'' \overset{r''}{\rightarrow} R''$) be, respectively, a first-order derivation and a second-order derivation, shown as diagrams in T-***Graph*** as follows.*



We say that ρ and ρ^2 are (inter-level) parallel independent iff exists $m_0'' : L'' \to G$ such that

1. $f_L; m_0 = g_L; m_0''$

2. *m_0'' satisfies application conditions for $L'' \overset{l''}{\leftarrow} K'' \overset{r''}{\rightarrow} R''$*

The existence of m_0'' in Definition 139 reflects the case where the elements added in the graph L by the second-order rewriting can be matched also in G, provided we maintain the instantiations induced by m_0. It is required we test m_0'' for application conditions along l'' because only the existence of a match is not enough to ensure rule application in the DPO approach.

Definition 140 (Inter-level conflict)**.** *A first-order rewriting $G \overset{r,m_0}{\Longrightarrow} H$ and a second-order rewriting $r \overset{\alpha,m}{\Longrightarrow} r''$ are conflicting iff they are not inter-level parallel independent.*

Summarizing, we have a conflict between two derivations in distinct levels if either the second-order rule makes the new rule fail to have a suitable match for G "compatible" with the original match m_0, or all compatible matches harm some application condition of the modified rule. To provide a better sense for what is a conflict of this kind, we provide some examples as follows.

Example 141 (Inter-level conflict)**.** *Figure 6.1 depicts four situations that cause inter-level conflicts. Each one focus on some part of the diagram shown in Definition 139.*

(a) here the original rule deletes a node and its self-edge. By means of second-order deletion, we modify the rule to make it delete only the node. Notice that there is an obvious match $m_0; f_l$ for rule $r' = L' \overset{l'}{\leftarrow} K' \overset{r'}{\rightarrow} R'$, which, however, cannot be applied because the edge dangling condition is harmed. This problem may be propagated towards rule $r'' : L'' \overset{l''}{\leftarrow} K'' \overset{r''}{\rightarrow} R''$ (and match m_0'') because by condition 1 in Definition 139 we must have $f_L; m_0 = g_L; m_0''$.

(b) in this case, two nodes p_1 and p_2 are mapped to the same node x_1. In the original rule, both p_1 and p_2 are preserved. The second-order rewriting deletes the preimage of p_1 in the interface, changing preservation into deletion. The rule r' cannot be applied with match $f_L; m_0$ because this harms the identification condition. This problem may also propagate to the final rewritten rule r'', as in the previous case.

(c) this situation is somewhat symmetric to condition (b), because it involves non injectivity of the match and changes only in the interface graph. However, the problem happens in the rightmost part of the second-order rewriting (creation of elements) and affects two elements that were originally deleted by the rule. Although there exists a match $m_0'' : L'' \to G$, the second-order rewriting converts deletion of p_1 into preservation, and this causes the match m_0'' to harm the identification condition.

(d) this last situation is somehow more obvious than the previous ones, and refers to the second-order rewriting augmenting the LHS of the rule in such a way it fails to have a match for G. In the particular case, a self-edge for node p_1 is created in graph L'' although there are no edges in G.

Based on this relationship between rewritings, we can now define a notion of independence between rewritings of first-order specifications, shown in the next definition.

Definition 142 (Parallel independence for S-SOGG derivations)**.** *Given a a S-SOGG $(T, D_0, D_1, D_2, \eta_2)$, we have parallel independence for two rewritings departing from $\mathcal{G} = (T, D_0', D_1')$ iff (by cases)*

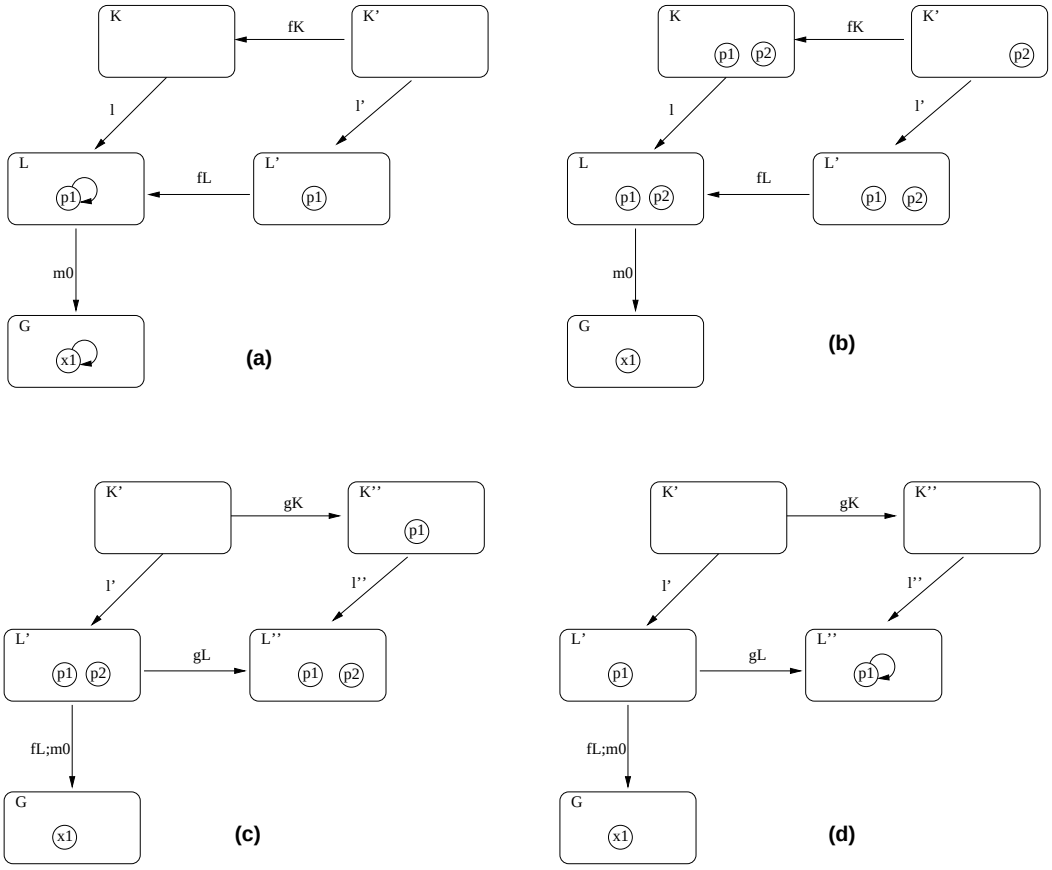


Figure 6.1: Situations causing inter-level conflicts between first-order and second-order rewritings.

1. *two first-order rewritings* ($\mathcal{G}_1 \overset{1,g_1,r_1,m_1}{\Longleftarrow} \mathcal{G} \overset{1,g_2,r_2,m_2}{\Longrightarrow} \mathcal{G}_2$)*:*

$$g_1 \neq g_2 \;\; or \;\; (g_1 = g_2 \; and \; G_1 \overset{dom(r_1),m_1}{\Longleftarrow} dom(g_1) \overset{dom(r_2),m_2}{\Longrightarrow} G_2 \\ are\ parallel\ independent\ graph\ rewritings)$$

2. *two second-order rewritings* ($\mathcal{G}_1 \overset{2,r_1,\alpha_1,m_1}{\Longleftarrow} \mathcal{G} \overset{2,r_2,\alpha_2,m_2}{\Longrightarrow} \mathcal{G}_2$)*:*

$$r_1 \neq r_2 \;\; or \;\; (r_1 = r_2 \; and \; p_1 \overset{(\eta_2(\alpha_1),dom(\alpha_1)),m_1}{\Longleftarrow} dom(r_1) \overset{(\eta_2(\alpha_2),dom(\alpha_2)),m_2}{\Longrightarrow} p_2 \\ are\ parallel\ independent\ second\text{-}order\ rewritings)$$

3. *one first-order, one second-order rewriting* ($\mathcal{G}_1 \overset{1,g_1,r_1,m_1}{\Longleftarrow} \mathcal{G} \overset{2,r_2,\alpha_2,m_2}{\Longrightarrow} \mathcal{G}_2$)*:*

$$r_1 \neq r_2 \;\; or \;\; (r_1 = r_2 \; and \; (G_1 \overset{dom(r_1),m_1}{\Longrightarrow} G_2, dom(r_1) \overset{(\eta_2(\alpha_2),dom(\alpha_2)),m_2}{\Longrightarrow} r') \\ are\ inter\text{-}level\ parallel\ independent\ rewritings)$$

Notice that, in the definition, g_1, g_2, r_1 and r_2 are injections indexing graphs and rules in coproduct collections. Sub-condition $g_1 \neq g_2$ and $r_1 \neq r_2$ states that are independent all pairs of derivations affecting distinct injections. When both derivations affect the same injection, they are independent iff the rewritings on the injection domain are, respectively, parallel independent graph rewritings, parallel independent second-order rewritings or inter-level parallel independent rewritings.

Regarding dependencies, we also are required to ponder about their actual meaning. As with conflicts, they are essentially asymmetric, since first-order rewritings do not interfere in any aspect with the applicability of a posterior second-order rewriting. On the other hand, a second-order rewriting modifying a rule can make an originally impossible application feasible. For instance, consider the reverse of the situation (d) depicted in Figure 6.1: the removal of a self-edge over a node in the LHS. Clearly, this modification in the rule makes the existence of a match for the depicted graph G possible, allowing the respective first-order rewriting. Moreover, a second-order rewriting could fix application conditions of an invalid match. This reasoning suggests that the adequate notion of inter-level dependency may be seen as a conflict between the *inverse* of the second-order rewriting and the respective first-order rewriting, confirming that the symmetry we observe between conflicts and dependencies in traditional DPO graph rewritings extends toward the inter-level scenario.

Definition 143 (Inter-level sequential dependencies)**.** *Let* $\rho_1 = r'' \overset{(N,\alpha),m}{\Longrightarrow} r$ *be a second-order derivation with NACs and* $\rho_2 = G \overset{r,m_0}{\Longrightarrow} H$ *be a first-order derivation. We say that* ρ_1 *and* ρ_2 *are inter-level sequential dependent iff* ρ_1^{-1} *and* ρ_2 *are inter-level parallel independent.*

Based on the symmetry of conflicts and dependecies, notice that it is straightforward to replicate definition 139 to define sequential dependence of S-SOGG derivations.

6.2 Critical pair analysis in second-order graph grammars

In the previous section we have defined a proper notion of conflict and dependency for derivations of S-SOGGs. Based on those notions, we propose an extension of the critical pair analysis algorithm that handles the specificities of S-SOGGs.

The critical pairs of a specification are the collection of essential conflicts that may potentially occur during its execution. The relevant distinction between a conventional graph grammar and a S-SOGG that affect the calculation of critical pairs is the fact that, when forming a pair of active elements to test for potential conflicts, we have the combinations (rule,rule), (2-rule,2-rule) and (rule,2-rule). The main challenge lies in the inter-level case, for which we must provide a way of calculating all possible conflicts between them. In the context of graph transformation, we obtain the collection of critical pairs between two rules $p_1 = L \leftarrow K \rightarrow R$ and $p_2 = L' \leftarrow K' \rightarrow R'$ (without NACs) by calculating all possible overlaps between L and L'. Each overlap represents a possible intersection of a match m_1 of p_1 and a match m_2 of p_2. When considering NACs, however, the case gets slightly more complicated because now we cannot test only the identification of elements in the LHS of rules, but also contextual information which lies out of the match. We refer the reader to (LAMBERS et al., 2008) for specific details about how to calculate critical pairs on adhesive HLR systems with NACs. Notice that is the case of span rewriting with NACs, which accounts for the case of (2-rule,2-rule).

In our inter-level scenario, we have a 2-rule $\alpha = a \leftarrow b \rightarrow c$ and a rule $p = L \leftarrow K \rightarrow R$, and we want to find all possible situations that result in inter-level conflict. Inter-level interaction requires a graph rewriting induced by p and a modification of p by α. The search for all possible modifications of p by α is straightforward, since we only have to find all matches (in T-**Span**) between a and p, and select the ones satisfying application conditions for second-order rewriting. We obtain a rewritten rule $p'' = L'' \leftarrow K'' \rightarrow R''$ from each of those valid matches. To find all graph rewritings induced by p that could be inter-level conflicting with the calculated second-order rewritings is slightly more difficult. This happens because we cannot simply calculate all partitions of L and use them as the target graph G. If we did this, we would miss some conflicting situations. For example, suppose our second-order rewriting increases the LHS of the rule, i.e., the left-hand side graph L'' of p'' is bigger than L. It is possible to have a target graph G which is big enough to provide a unique match $m_0'' : L'' \rightarrow G$, however, it may not be the case that m_0'' satisfy the DPO dangling condition. This is clearly a critical pair, and our algorithm should be able to construct it. But since dangling conditions depend on extra elements out of the image of match m_0'', we need extra elements in G that are not present in L to characterize this conflict. This is the motivation for the following definition, which we name *dangling extension*. The idea is to extend the LHS of a grah rule in order to capture possible conflicts that arise from harming dangling conditions.

Definition 144 (Dangling extension)**.** *Let $p = L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ be a T-typed graph rule, where $t_L : L \rightarrow T$ is the typing morphism (in* **Graph***) of the left-hand side L. Consider the following constructions:*

1. *let us write $L_D = \{x \mid x \in Nodes(L) \wedge \mathcal{O}_l(x)\}$ the collection of all nodes in L deleted by the application of p.*

2. *let us write $DangT(x) = \{e \mid e \in Edges(T) \wedge (src(e) = t_L(x) \vee tgt(e) = t_L(x)\}$ the collection of all* edge types *in T that are connected to the node type $t_L(x)$.*

We define the dangling extension of L, written L^+, the graph obtained by creating, for each deleted node $x \in L_D$, one instance edge for each type edge in $DangT(x)$. For edges that are not self-loops, new instance nodes to serve as targets of sources are also created. We write $dext_L : L \hookrightarrow L^+$ the obvious inclusion of L into its dangling extension L^+.

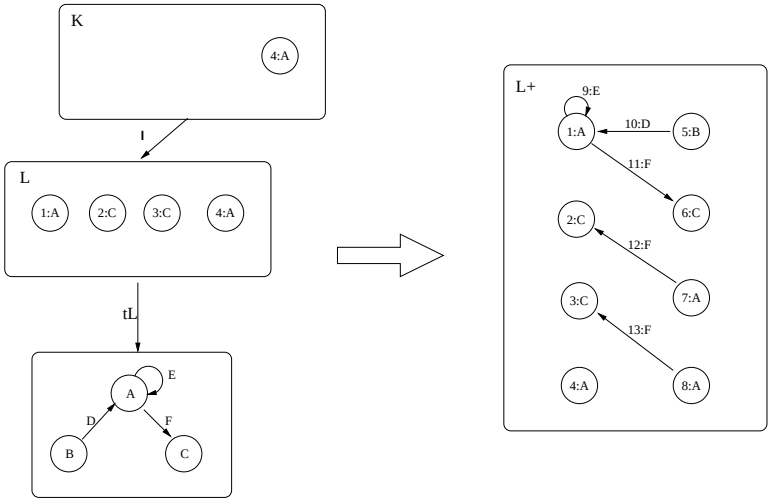


Figure 6.2: Example of dangling extension.

Example 145 (Dangling extension)**.** *Figure 6.2 presents an example of the operation named dangling extension. The left-hand side graph L contains four nodes, from which some are being deleted (1,2 and 3) and one is being preserved (4). To the right, we see the resulting dangling extension, which creates new edge instances based on the connectivity of the type of each node. Note that nodes 5,6,7 and 8 had to be created as sources or targets of the new edges.*

Definition 146 (Inter-level critical pair calculation)**.** *Given a 2-rule $\alpha = a \leftarrow b \rightarrow c$ and a rule $p = L \leftarrow K \rightarrow R$, the following steps calculate all possible inter-level critical pairs.*

1. *Calculate all second-order matches $m : a \rightarrow p$ satisfying application conditions for second-order rewriting, and collect them in the set $M_{\triangle}$.*

2. *For each second-order match $m \in M_{\triangle}$, we execute the following steps:*

   (a) *execute the second-order rewriting $p \overset{\alpha,m}{\Longrightarrow} p''$, obtaining an intermediate rule $p' = L' \leftarrow K' \rightarrow R'$ and a final rule $p'' = L'' \leftarrow K'' \rightarrow R''$.*

   (b) *calculate the pushout of diagram $L \xleftarrow{f_L} L' \xrightarrow{g_L;dext_{L''}} (L'')^+$, as shown in the diagram below*

$$\begin{array}{ccccc} L' & \xrightarrow{g_L} & L'' & \xrightarrow{dext_{L''}} & (L'')^+ \\ {\scriptstyle f_L}\downarrow & & & & \downarrow \\ L & & \xrightarrow[a_{L^\star}]{} & & L^\star \end{array}$$

   (c) *calculate the set of all subgraphs of $L^\star$ containing the graph L (via $a_{L^\star}$). In other words, the set*

$$Test_m = \{X \mid a_{L^\star}(L) \subseteq X \subseteq L^\star\}$$

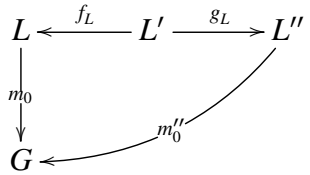
   Let us call a_X the arrow from L to each $X \in Test_m$ that factors through $a_{L^\star}$, as shown below.

$$\begin{array}{ccccc} L & \xrightarrow{a_X} & X & \hookrightarrow & L^\star \\ & \searrow & & \nearrow & \\ & & \xrightarrow[a_{L^\star}]{} & & \end{array}$$

(d) *for each graph $X \in Test_m$ we calculate the set $Par(X)$ of all partitions of X. Notice that each partition $P \in Par(X)$ can be represented (up to isomorphism) by an epimorphism $e_P : X \twoheadrightarrow P$.*

(e) *for each partition $P \in Par(X)$ of each graph $X \in Test_m$, calculate the arrow $a_X; e_P : L \to P$. Test each arrow for application conditions with respect to rule p, and collect all valid matches in the set $M^m_\triangledown$.*

3. *for each pair (m, m_0) where $m : a \to p \in M_\triangle$ and $m_0 : L \to G \in M^m_\triangledown$, calculate all factorizations $m''_0 : L'' \to G$ such that $f_L; m_0 = g_L; m''_0$, and collect them in set $Fact(m, m_0)$.*



If $Fact(m, m_0) = \varnothing$ or all $y \in Fact(m, m_0)$ harm some application condition for p'', then (m, m_0) is a critical pair.

4. *Return the collection of all pairs (m, m_0) which are critical pairs.*

The essence of the algorithm is fairly simple: initially calculate all second-order rewritings, then calculate a number of candidate first-order rewritings and finally test for inter-level conflicts between pairs of first-order and second-order rewritings. Step 1 calculates all valid second-order matches between the given 2-rule and rule, and collect them into a single set. For each second-order match, we calculate a collection of candidate first-order rewritings in step 2. The complex part is the generation of candidates for first-order rewritings, since we must care to obtain all possible conflicts for p'', which may be caused by

- *non-existence of match for p'':* the graph G does not contain enough elements to allow a match for p'' consistent with $f_L; m_0$.
- *problem with identification condition in p'':*, m''_0 exists but it identifies deleted elements with preserved ones.
- *problem with dangling condition in p'':* m''_0 tries to delete some node that has a connected edge which is out of $m''_0(L'')$. In order to capture those situations, we were forced to introduce the notion of *dangling extension*.

Step 2.*a* executes the second-order rewriting, obtaining the modified rule p''. Step 2.*b* obtains the graph $L^\star$, obtained from the amalgamated sum of the dangling extension of L'' and the elements in L which are not in L'. The graphs L and $L^\star$, respectively, contains the least number of elements and the maximum number of elements for the candidate target graphs. Actual tests for matches are partitions of graphs "in between" L and $L^\star$, calculated in steps 2.*c*, 2.*d* and 2.*e*. Step 3 basically runs over all the previously calculated second-order and first-order rewritings and tests for conflicts following Definition 140, and step 4 returns the results.

Although having a simple mechanics, this algorithm for finding critical pairs is not very efficient in several stages because the calculation of partitions, subgraphs and morphism

factorizations are usually quite expensive. Although they are feasible considering small typed graphs, where the typing mechanism reduces considerably the search space, those computational steps may be quite restrictive when considering examples with few types and large graph instances. Moreover, although our intuition says we are accounting for all kinds of conflicting situations, it still remains as future work to prove that this algorithm is complete in the sense it captures all existing critical pairs between rule p and 2-rule α. To improve its efficiency is also an interesting problem to solve. Assuming there are no problems with the algorithm, we can fit it in our definition of second-order critical pair analysis for C-SOGGs as follows.

Definition 147 (Second-order critical pair analysis for C-SOGG)**.** *Let $(T, D_0, D_1, D_0^2, D_1^2, \eta_0^2, \eta_1^2)$ be a complete SOGG. The following steps calculate all of its critical pairs (conflicts).*

- *for all pairs (p_1, p_2) such that $p_1, p_2 \in \mathscr{A}(D_1)$, calculate all critical pairs between graph rules $dom(p_1)$ and $dom(p_2)$.*
- *for all pairs (p_1, p_2) such that $p_1, p_2 \in \mathscr{A}(D_0^2)$, calculate all critical pairs between graph rules with NACs $(\eta_0^2(p_1), dom(p_1))$ and $(\eta_0^2(p_2), dom(p_2))$.*
- *for all pairs (α_1, α_2) such that $\alpha_1, \alpha_2 \in \mathscr{A}(D_1^2)$, calculate all critical pairs between the 2-rule $dom(\alpha_1)$ with NACs $(\eta_1^2(\alpha_1) \cup \mathscr{S}(dom(\alpha_1))$ and 2-rule $dom(\alpha_2)$ with NACs $(\eta_1^2(\alpha_2) \cup \mathscr{S}(dom(\alpha_2))$. For such, use the algorithm described in (LAMBERS et al., 2008).*
- *for all pairs (α, p) such that $p \in \mathscr{A}(D_1)$ and $\alpha \in \mathscr{A}(D_2)$, calculate all inter-level critical pairs between $dom(\alpha)$ and $dom(p)$, using the method described in Definition 146.*

We name this method second-order critical pair analysis *(SO-CPA).*

In Definition 147 we are essentially distinguishing four cases, for which we employ distinct methods that are adequate for each situation. The calculation of dependencies is essentially the same, considering the reverse rule of the first component of the pair. Second-order critical pair analysis of retyping-aware models is obtained from the SO-CPA on their respective retyped models, and for CDM+ models we consider the resulting D_0 and D_1 after rule deletion and creation for the calculation of critical pairs.

6.3 Evolution of critical pairs due to model transformation

One of the goals of defining SOGGs was to be able to study formally model transformations over first-order graph grammars. For such, we have introduced the concepts of *model-transformation derivation*, *evolved graph grammar* and also the construction of a *evolutionary span* comparing the initial graph grammar with the evolved one. However, we have not yet considered how to use them to foresee the effect of model transformations over the behavior of the first-order system. This problem is not easy at all: mostly often graph grammar have infinite behaviors, and thus it is (in general) not possible to represent all possible derivations in a single structure as we have done with the simple place-transition system in the first chapter. Moreover, the notion of behavior may change if we consider parallelism, for instance. In the general case, it is not possible to make non-trivial predictions about the effect of changes in graph grammars due to their expressive

power. Still, it is possible to gain some insight about the effect of the model transformation by focusing on how it affects the interaction between pairs of rules in both the original and evolved system. In this session we explore this idea, for which we trace the modifications caused by the model evolution over the critical pairs of the original system and of the new system. From this, we may discover that the model transformation has added, removed or maintained conflicts and dependencies.

We start by recalling how to calculate critical pairs for (first-order) graph grammars without NACs. For each pair of rules (p_1, p_2) where $p_1 = L_1 \leftarrow K_1 \rightarrow R_1$ and $p_2 = L_2 \leftarrow K_2 \rightarrow R_2$, we calculate the disjoint union $G = L_1 + L_2$. This graph represents the situation where the image of both matches are totally disconnected. By calculating the set $Part(G)$ of all partitions of G, we represent all possible overlaps between matches of both rules. From $Part(G)$, we remove overlaps that do not satisfy application conditions for p_1 or p_2, and we end up with a collection of *candidate rule match overlaps* for with we can classify into *conflicting* (critical pairs) or *non-conflicting* (common overlaps).

Definition 148 (Rule match overlap)**.** *Let $p_1 = L_1 \leftarrow K_1 \rightarrow R_1$ and $p_2 = L_2 \leftarrow K_2 \rightarrow R_2$ be graph rules without NACs. A rule match overlap for p_1 and p_2 is a pair (m_1, m_2) of jointly surjective graph morphisms $m_1 : L_1 \rightarrow X$ and $m_2 : L_2 \rightarrow X$ such that m_1 satisfies application conditions for p_1 and m_2 satisfies application conditions for p_2. The collection of all match overlaps for a pair of rules (p_1, p_2) is denoted $Overlaps(p_1, p_2)$.*

Now, consider that by means of model transformation we have modified both rules p_1 and p_2 into, respectively, p_1'' and p_2'', and, moreover, we have their respective evolutionary spans $p_1 \leftarrow p_1' \rightarrow p_1''$ and $p_2 \leftarrow p_2' \rightarrow p_2''$. We can see both spans by their components in the following diagrams in T-**Graph**.

$$\begin{array}{ccccc}
p_1 & & R_1 & \longleftarrow & K_1 & \longrightarrow & L_1 \\
\uparrow & & \uparrow & & \uparrow & & \uparrow \\
p_1' & = & R_1' & \longleftarrow & K_1' & \longrightarrow & L_1' \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
p_1'' & & R_1'' & \longleftarrow & K_1'' & \longrightarrow & L_1''
\end{array}
\qquad\qquad
\begin{array}{ccccc}
L_2 & \longleftarrow & K_2 & \longrightarrow & R_2 & & p_2 \\
\uparrow & & \uparrow & & \uparrow & & \uparrow \\
L_2' & \longleftarrow & K_2' & \longrightarrow & R_2' & = & p_2' \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
L_2'' & \longleftarrow & K_2'' & \longrightarrow & R_2'' & & p_2''
\end{array}$$

Notice that the diagrams can be read either vertically or horizontally, representing 2-rules in both cases. Based on the evolution of both rules, we can calculate how the match overlaps of the original pair (p_1, p_2) relate to match overlaps of the evolved pair (p_1'', p_2''). We achieve this by calculating all possible *evolutionary spans of match overlaps* as described by the following definition.

Definition 149 (Evolutionary span of match overlaps)**.** *Let*

$$e_1 = p_1 \leftarrow p_1' \rightarrow p_1''$$

and

$$e_2 = p_2 \leftarrow p_2' \rightarrow p_2''$$

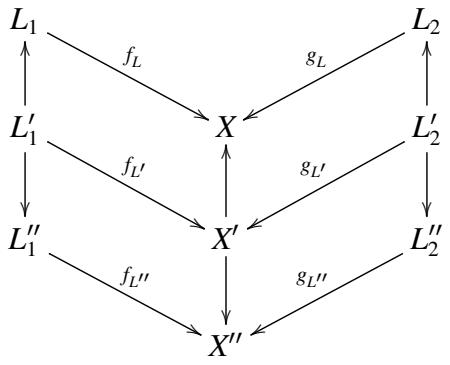
be two evolutionary spans of graph rules. An evolutionary span of match overlaps *for e_1 and e_2 is a pair (f, g) of jointly surjective graph span morphisms*

$$f : (L_1 \leftarrow L_1' \rightarrow L_1'') \rightarrow (X \leftarrow X' \rightarrow X'')$$

and

$$g : (L_2 \leftarrow L_2' \rightarrow L_2'') \rightarrow (X \leftarrow X' \rightarrow X'')$$

*as shown in the following diagram in category T-***Graph**



such that

1. $(X \leftarrow X' \rightarrow X'')$ *is an injective graph span.*
2. (f_L, g_L) *is a match overlap for* (p_1, p_2).
3. $(f_{L''}, g_{L''})$ *is a match overlap for* (p_1'', p_2'').

We write $OverlapEvolution(e_1, e_2)$ *to denote the collection of all candidate overlaps between* e_1 *and* e_2.

Each evolutionary span of match overlap represents a connection between a match overlap of the original pair (p_1, p_2) and a match overlap of the evolved pair (p_1'', p_2''). By calculating $OverlapEvolution(e_1, e_2)$, we can define a formal relation between the collections $Overlaps(p_1, p_2)$ and $Overlaps(p_1'', p_2'')$.

Definition 150 (Evolution relation between overlaps)**.** *Let* $e_1 = p_1 \leftarrow p_1' \rightarrow p_1''$ *and* $e_2 = p_2 \leftarrow p_2' \rightarrow p_2''$ *be two evolutionary spans of graph rules. We define the evolution relation* $\mathscr{R} \subseteq Overlaps(p_1, p_2) \times Overlaps(p_1'', p_2'')$ *by*

$$a \mathscr{R} b \;\; \textit{iff} \;\; \exists c (a \leftarrow c \rightarrow b \in OverlapEvolution(e_1, e_2))$$

Since each candidate overlap of (p_1, p_2) and of (p_1'', p_2'') can be either conflict-free or a critical pair, by means of $\mathscr{R}$ we can trace the effect of the overall evolution of rules in terms of increasing or decreasing the level of parallelism of the system. If a conflict-free overlap of the original system is related through $\mathscr{R}$ to a critical pair of the evolved system, the software evolution may have (potentially) reduced the ability to parallel execution. On the other hand, if a critical pair in the original system is only related to conflict-free overlaps in the evolved system, we may have *increased* the potential for parallelism of the system. We name *critical pair evolution* (CPE) this method of detecting the changes in critical pairs of the first-order layer through the relation $\mathscr{R}$.

6.4 Summary

This chapter dealt with the problem of defining an adequate notion of conflict between first-order and second-order derivations in SOGGs. We started with the notions of inter-level parallel independence and inter-level conflict. Then, we have proposed two analysis

techniques based on the interaction of the two execution layers. The first is an extension of the critical pair analysis for second-order graph grammars, which we named *second-order critical pair analysis*. The second method, called *critical pair evolution*, defines formally how rule match overlaps of the original system relate to rule match overlaps of the evolved system. This can be used to identify if a model evolution is increasing or decreasing the parallelism of the base system.

7 ASPECT-ORIENTED GRAPH GRAMMARS

Aspect-oriented programming (AOP) (KICZALES et al., 1997) is a paradigm which aims to solve the *code entanglement* problem in object-oriented software. For this, it employs a *rule-based*, *invasive* code combination process known as aspect weaving. There are similarities between aspect-oriented concepts and the principle of graph transformation, which may be explored in some ways, such as, for instance, using analysis techniques from the graph rewriting area to study models with aspects. Based on those similarities we propose a characterization of aspect-oriented concepts for first-order graph grammars, arriving at the model we name *aspect-oriented graph grammars*. This characterization involves second-order constructions that have been introduced in previous chapters.

This chapter is organized as follows: initially we revise the basic notions behind aspect-oriented programming and aspect-oriented modelling. Then, we discuss the similarities between aspect-oriented languages and models based on graph rewriting. Finally we define the aspect-oriented graph grammar model by means of second-order rewriting principles, and discuss how to analyse them using the techniques discussed in the previous chapter. The original contributions of this chapter are the following:

- the definition of aspect-oriented concepts for graph grammars through the framework of second-order graph grammars.
- the usage of analysis techniques developed relating model transformation and base system rewriting to study aspect-oriented constructions.

7.1 Aspect-oriented programming

Several characteristics of the object oriented paradigm favor good practices in software development. Restriction permissions for components, such as the `private` modifier in Java and C++, help to improve the encapsulation of objects. The inheritance mechanism for classes encourages the reuse of code, an so on. However, it is known that the implementation of a given family of requirements may not be properly modularized by the abstractions of the object-oriented paradigm.

To illustrate the main idea, we will refer to Figure 7.1, which shows the source code for the Apache Tomcat Web Server (KERSTEN, 2002). In the picture, each column represents a class, while the length of the column accounts for the amount of lines of code within it. The shadowed portion within the class refers to the lines devoted to implement XML parsing, a functionality which is properly contained. Possible changes in the structure of XML parsing cause only local adjustments, which implies that this implementation requirement is properly modularized. Now consider the statements for logging in the same application, shown by Figure 7.2. Hypothetically, suppose that the log policy is “register

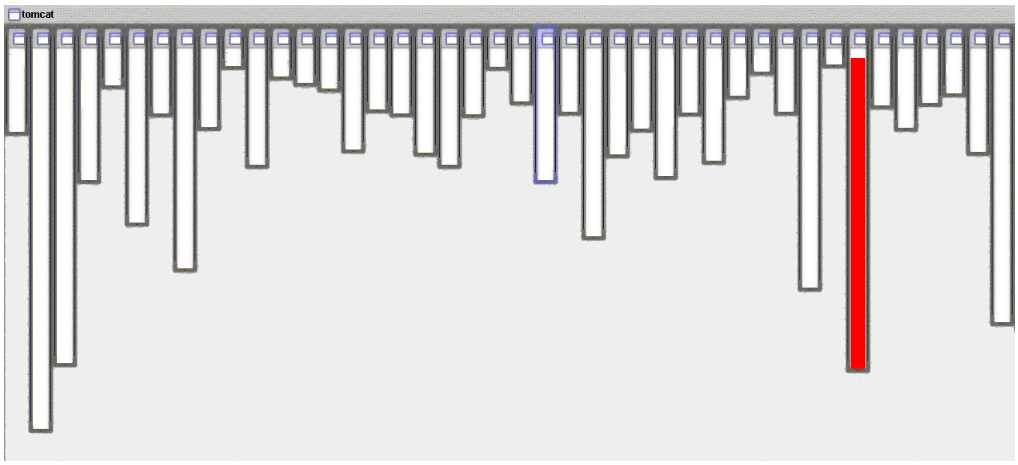


Figure 7.1: Code for XML parsing in the Apache Tomcat web server.

all operations involving opening and closing connections". This policy may be implemented by inserting method calls throughout the code whenever another statement reads or modifies connection handlers. Notice that this insertion of statements depends on how the handlers are accessed within method bodies. As a consequence, changes in the logging policy may require manual adjustments throughout several classes, which is usually an error-prone and time consuming task. Moreover, if there are other requirements besides logging that also depend on such code patterns, the bodies of methods may become excessively cluttered due to the excess of extra statements, hiding their original purposes.

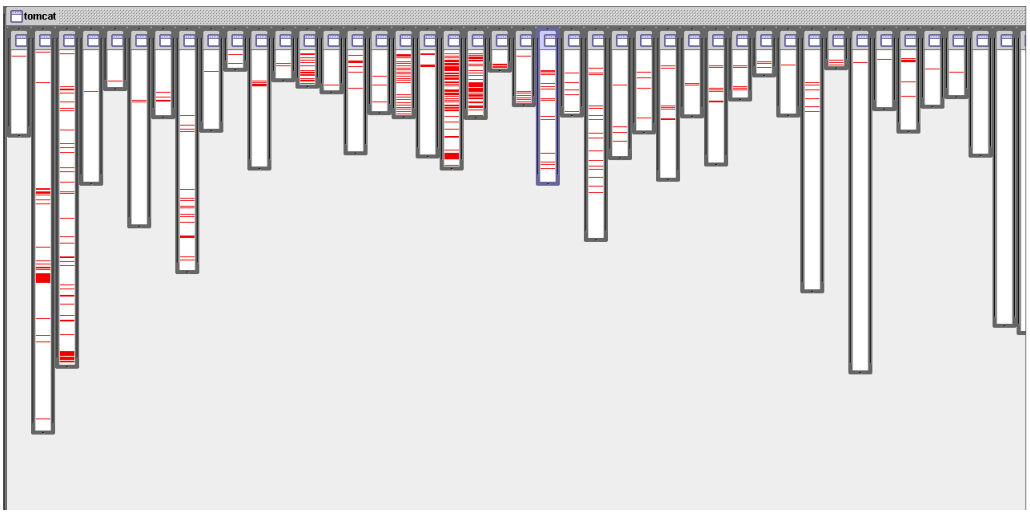


Figure 7.2: Logging statements in the Apache Tomcat web server.

This effect of code scattering happens not only to logging requirements, but also to session handling, security policies, distributed object management, among others policies. Such implementation requirements are commonly referred as *crosscutting concerns*. Roughly speaking, what the aspect-oriented paradigm proposes is the creation of new abstractions to implement modularly crosscutting concerns. For instance, in the logging case, there should be a unique application module, called an aspect, where all the conditions, rules and statements for logging should be specified, as depicted in Figure 7.3. Notice that there are no log-related statements within other system modules. According to (FILMAN; FRIEDMAN, 2000), AOP is characterized by *quantification* and *obliviousness*. Quantifi-

cation refers to the ability to range over the structure of the application selecting interesting points of combination. Obliviousness refers to the fact that aspects are weaved to other system modules implicitly. The module developer is oblivious about the effect of aspects over its own code. Aspect-oriented languages must provide new language constructions to reason about the own language structures, and combination patterns to determine how they should be transformed. The main abstractions of the aspect-oriented paradigm are the following:

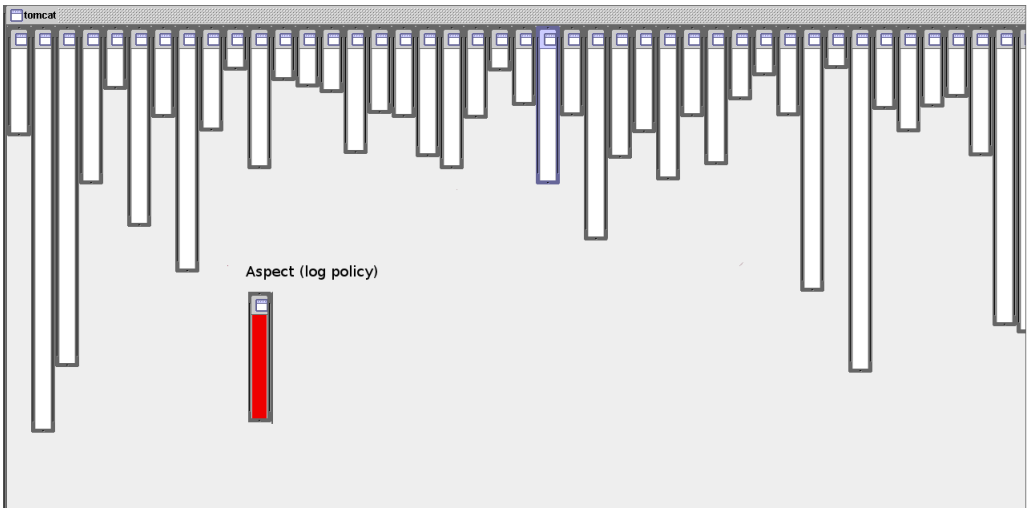


Figure 7.3: Log policy using aspects in the Apache Tomcat web server.

- *join points:* a subset of the language transitions steps that can be affected by aspects. In most object-oriented languages, join points refer to transition such as method calls (before dispatch), method executions (after dispatch), field accesses, field modifications, constructor calls, exception throws, etc. Notice that may be some transitions, such as iterations of `for` loops, that may not be join points. The definition of join points depends on the language structure, and may be also affected by the language implementation.

- *pointcuts:* a particular set of join points, used to determine a crosscutting pattern. An example of pointcut may be "all method calls where result is of type `int`". AOP languages define an *pointcut expression language* to determine pointcuts. For instance, the previous pointcut is written in AspectJ as `"int *.*(..)"`.

- *advices:* code modification rules. Advices have the format `pointcut → effect`. The effect describes some kind of code modification to be executed when in the join points of the specified pointcut. In AspectJ, the effect is specified by a tuple `(type,code)`, where `type` may be `before`, `after` or `around`, while `code` refers to Java statements. The aspect weaver inserts the given code, before, after or in replacement of the join point code.

- *inter-type declarations:* are extensions to the static structure of the system, normally new methods, fields or subclass relationships which are introduced implicitly by aspects. These modifications over the structure of the application are usually needed by code inserted by advices.

- *aspects:* modules grouping all definitions of pointcuts, advices and inter-type declarations that deal with one specific crosscutting concern.

The AOP paradigm also relies on an operation named *aspect weaving* to combine aspect specifications with the rest of the application, resulting in the final system, as shown in Figure 7.4.

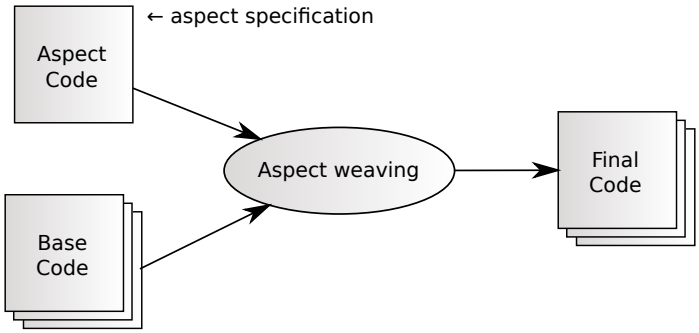


Figure 7.4: Aspect weaving.

The aspect weaving operation can be done at compile-time, load-time or at run-time, according to the implementation of the aspect weaver (the aspect combination module of the language). As a simple example of weaving, consider the AspectJ source code depicted in Figure 7.5 (AspectJ is the most popular AOP extension for the Java programming language). The AspectJ weaver receives both the base code and the aspect code. Then, it applies the advices within the aspect, merging the code in advices every time it finds a match for their pointcuts in the base code. In the example of Figure 7.5, the aspects simply introduces a print command right before the start of the execution of join point. The pointcut selects any method without parameters sent to an object of class `A`. Although the result of the combination is shown as a source-code transformation, the weaver actually performs the code combination in bytecode level.

Despite the advantages of the paradigm, the implicit way aspects are composed with other system modules may lead to some problems. Since aspects may modify the body of methods in every class of the application, the encapsulation provided by the object-oriented paradigm is harmed. Moreover, there are also no guarantees that the weaving of two aspects may lead to the same system independently of the order they are combined. As the number of aspects grows, it may be very difficult to reason about the program, specially due to interactions between the original code and the inserted one for each aspect. To overcome some of those problems and be able to understand the system behavior, the developer needs both adequate development tools and reasoning models for aspect-oriented languages and diagrams. On the implementation side, integrated development environments such as Eclipse already offer views to expose the aspect interference in the source code. On the formal perspective, several calculi have been proposed to define aspect weaving over programming languages (MASUHARA; KICZALES; DUTCHYN, 2002; JAGADEESAN; JEFFREY; RIELY, 2003; WAND; KICZALES; DUTCHYN, 2004; JAGADEESAN; JEFFREY; RIELY, 2006; CLIFTON; LEAVENS, 2006; HUI; RIELY, 2007; FRAINE; SÜDHOLT; JONCKERS, 2008). The purpose of those approaches is to provide a precise semantics for aspect-oriented constructions over well-established object-oriented or functional core languages. In most of the cases, they specify an operational semantics and type system for dynamic advice weaving, reproducing the behavior of some actual aspect-oriented language such as AspectJ.

Base code:

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

+

Aspect:

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

⇓

Weaved code

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

  void b() {
             System.out.println("Method without parameters\n");
             ... body of b ...  }

  void c(int x) { ... body of c ... }
}
```

Figure 7.5: Example of aspect weaving in AspectJ.

In order to identify and characterize aspects in the whole software development cycle, it is required to represent concepts such as advices and pointcuts in class diagrams, sequence diagrams and other related models. *Aspect-oriented modeling* refers to the identification and representation of aspect-oriented constructions in the context of system modeling. As shown in (SCHAUERHUBER et al., 2006), there are several approaches. Some of them, such as (ALDAWUD; ELRAD; BADER, 2003; ZHANG, 2005; FUENTES; SÁNCHEZ, 2006, 2007; JÚNIOR; CAMARGO; CHAVEZ, 2009), represent aspects using the meta-object facility (MOF) mechanism of UML. There are approches, such as (EVERMANN, 2007), which provide a detailed embedding of some aspect-oriented language within UML diagrams, referring to the weaving semantics of the language. Finally, there are some graph-transformation approaches, such as (WHITTLE et al., 2009; MEHNER; MONGA; TAENTZER, 2006), where the aspect weaving process is given by means of graph rewriting.

7.2 Comparison of aspect weaving and graph transformation

In this section we compare the main concepts from both aspect-orientation and graph rewriting. One of the first (if not the first) reference to the similarities of both approaches is given by (ASSMANN; LUDWIG, 1999), just after the name aspect-orientation became known. One common point is that aspect weaving manipulates programs, which in turn may be seen as abstract syntax trees, and, consequently, also as graphs. Advices are program rewriting rules to be applied over a base program, aiming to affect all join points matching the advice pointcut. From this, the analogy to be drawn becomes fairly straightforward, as it is show in Figure 7.6.

Concept	AOP	Graph rewriting
base system	abstract syntax tree	graph
place of combination	join points	region of a graph
pattern to be found	pointcut	graph (LHS of rule)
matching	$\in$	graph morphism
rewriting rule	advice	graph transformation rule
typing extensions	inter-type declarations	type graph extension
aspects	collection of advices and inter-type declarations	type graph extension and a collection of graph rules
combination	aspect weaving	graph rewriting

Figure 7.6: Analogy between aspect weaving and graph rewriting.

The same analogy is explored in (WHITTLE et al., 2009) and (MEHNER; MONGA; TAENTZER, 2006), which propose to implement aspect weaving for UML-like models through graph rewriting. Both approaches employ the critical pair analysis technique for ensuring that there are no conflicts among aspects affecting the same model. We can say that our proposed approach follows the same general lines. However, the difference lies in the fact that we intend to work with higher-order graph grammars, i.e. to consider graph grammars as base systems. Models such as UML activity diagrams have to be properly formalized before attempting to address its semantics with the one for the weaving layer, possibly with a distinct principle than graph rewriting. By considering graph grammars themselves as a model, we have the same principle for describing both the underlying model and also aspect weaving process, which makes possible to relate them using in the same context. As examples, we can refer to the notion of inter-level conflict, that relates a second-order rewriting with a first-order one, and also the notion of critical pair evolution.

7.3 Aspect-oriented graph grammars

This section describes how to use second-order graph grammars to represent aspect-oriented concepts over graph grammar models. Roughly speaking, we characterize an aspects as a second-order layer over an initial first-order graph grammar $\mathcal{G}$. We also describe how to combine several aspects simultaneously. The final weaved graph grammar is associated with the evolved grammar of a model-transformation derivation. If the aspect weaving process is confluent, then the evolved grammar is unique. We then use the developed techniques of second-order critical pair analysis, and critical pair evolution to study the effect of the aspects over the base system.

Example 151 (Base system)**.** *Figure 7.7 introduces the base system used as a working*

example. It is essentially a version of the client-server system with self-loops in messages to ensure only one data node will be loaded for each message.

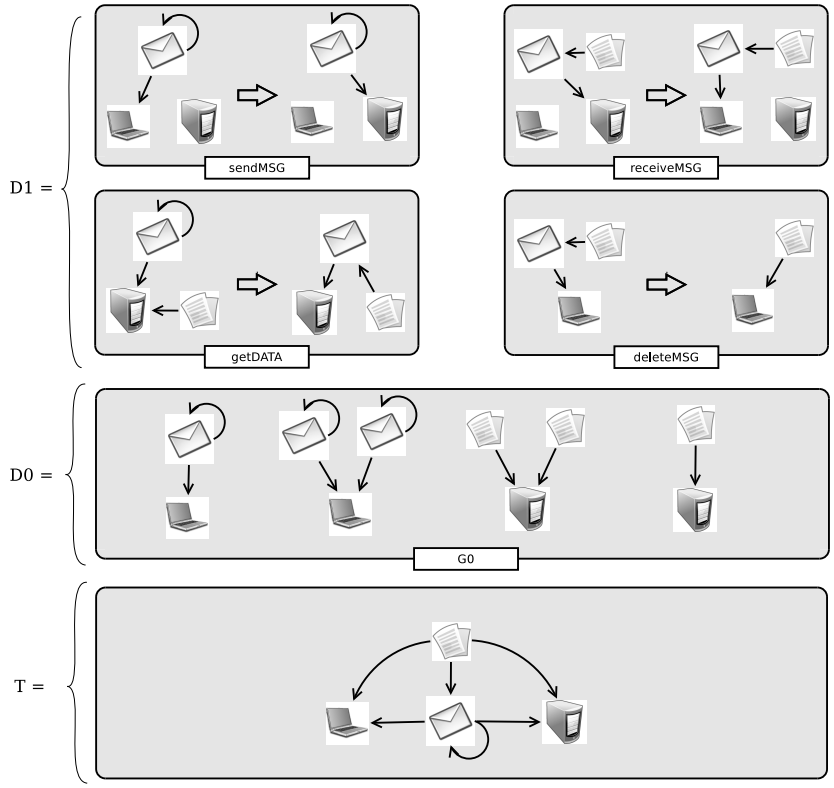


Figure 7.7: Base system.

The main purpose of the aspect-oriented paradigm is to solve the problem of lack of modularity for the code that handles crosscutting concerns. In order to illustrate crosscutting concerns in the context of graph grammars, we propose two simple modifications to the system of Figure 7.7 the inclusion of a *logging* object (to log executions) and of a domain policy for server access.

Logging: Suppose we want to register every execution step within the system in order to have access to the execution history. For instance, it is very common that servers store information about the start and the end of each client connection, both for profiling and security reasons. In the context of graph grammars, this would mean that we have to record each production application, or derivation step.

Domain policy: Let us suppose that, due to a large number of clients and servers, the modeler wishes to divide the traffic into regions (or domains). This means that message passing will only occur within the bounds of a group of servers and client within the same domain. We need to represent domains using nodes and edges, and also enforce the restriction of only sending messages across elements within the same domain to all rules that transfer messages between clients and servers.

Notice that those requirements involve modifications in an arbitrary number of rules of the first-order system. In the first case (logging), all rules should be modified; in the second case (domains), we must modify all rules that satisfy a given criteria, in the case

the ones that transfer messages. If we consider that the first-order system may be distinct and may have a quite large number of rules, we may think of these transformations as *crosscutting concerns* to be applied over the whole base system. Apart from modifications in rules, we also require extension to the type graph and to the initial graph. Note also that those modifications are the ones we can represent by means of a second-order layer. One of them, the logging policy, has been already implemented in the second-order graph grammar of Figure 5.7. The introduction of domains for sending and receiving messages can also be implemented in a similar way. From this observation, we define an aspect for a given first-order graph grammars as a *second-order layer* over it, which implements a model-transformation.

Definition 152 (Graph grammar aspect)**.** *Let $\mathcal{G} = (T, D_0, D_1)$ be a first-order graph grammar. An* aspect *A for $\mathcal{G}$ is a tuple*

$$A = (RT, D_0^2, D_1^2, \eta_0^2, \eta_1^2, C_1^2)$$

such that $(T, D_0, D_1, RT, D_0^2, D_1^2, \eta_0^2, \eta_1^2, C_1^2)$ is a CDM-RC-SOGG specification. We will also use the notation $(\mathcal{G}, A)$ to represent this CDM-RC-SOGG.

Referring to the elements of Definition 152, we establish the following nomenclature to match the concepts of aspect-oriented programming with the terminology of second-order graph grammars.

- the retyping span *RT* is the *inter-type declaration* of *A*.
- the graph rules described by (D_0^2, η_0^2) are the *first-order advices*. For each first-order advice, the LHS graph is the respective *pointcut*.
- the 2-rules described by (D_1^2, η_1^2) are the *second-order advices*. Their respective LHS rules are the *pointcuts*. Creation and deletion rules of kind $0 \leftarrow 0 \rightarrow r$ and $r \leftarrow 0 \rightarrow 0$, induced by create-delete statements in C_1^2, are also considered second-order advices.

Now we describe implementations of both logging and domain policies for the base system.

Example 153 (Logging aspect)**.** *Figure 7.8 depicts the definition of the logging aspect for the base system.*

Example 154 (Domain aspect)**.** *Figure 7.9 depicts an aspect that implements domain restriction on message sending. The type graph is extended with a type* D*, representing a domain type, and edges from both client and server type towards it. Clearly, a server or client belongs to a particular domain instance if there is an edge instance connecting them. There three rules, named* newDomain*,* addClient *and* addServer*, that create a domain division between the elements of the initial graph. Rule* newDomain *creates a domain node connecting a particular server and client. Then, rules* addClient *and* addServer *add a new stray client or server, respectively, to a domain. Negative application conditions for these three rules assure they only connect to domains servers and clients that are not already connected. Notice also that the creation of domains and connection of clients and servers occur in a non-deterministic way, and will depend on the execution of those creation rules.*

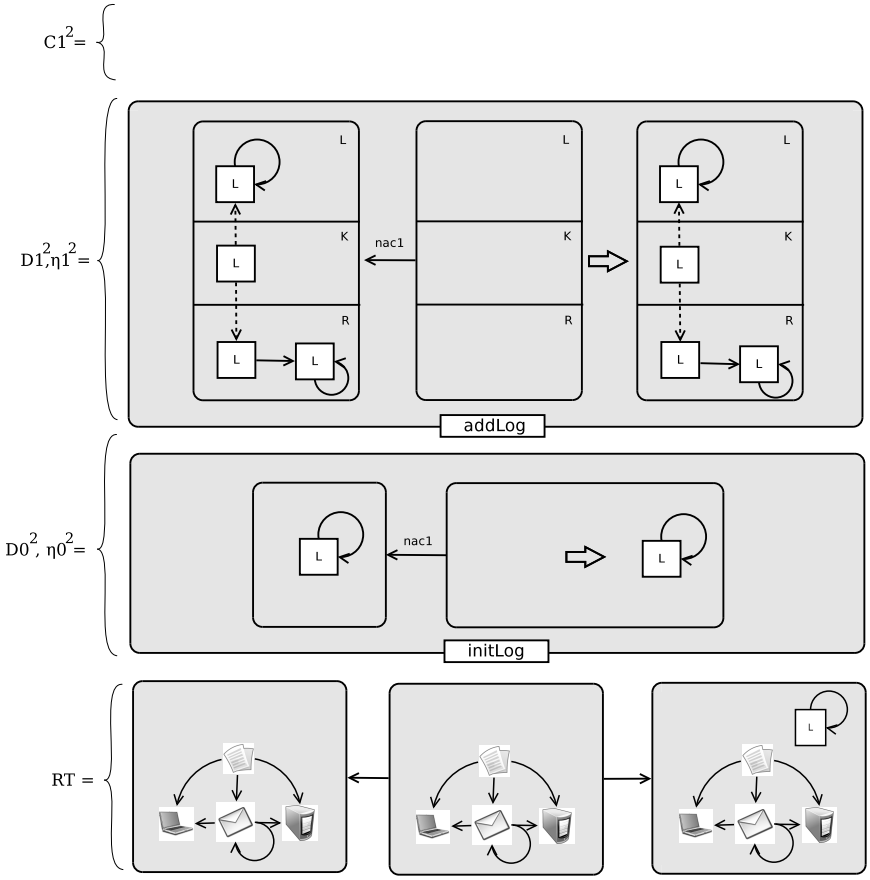


Figure 7.8: Logging aspect.

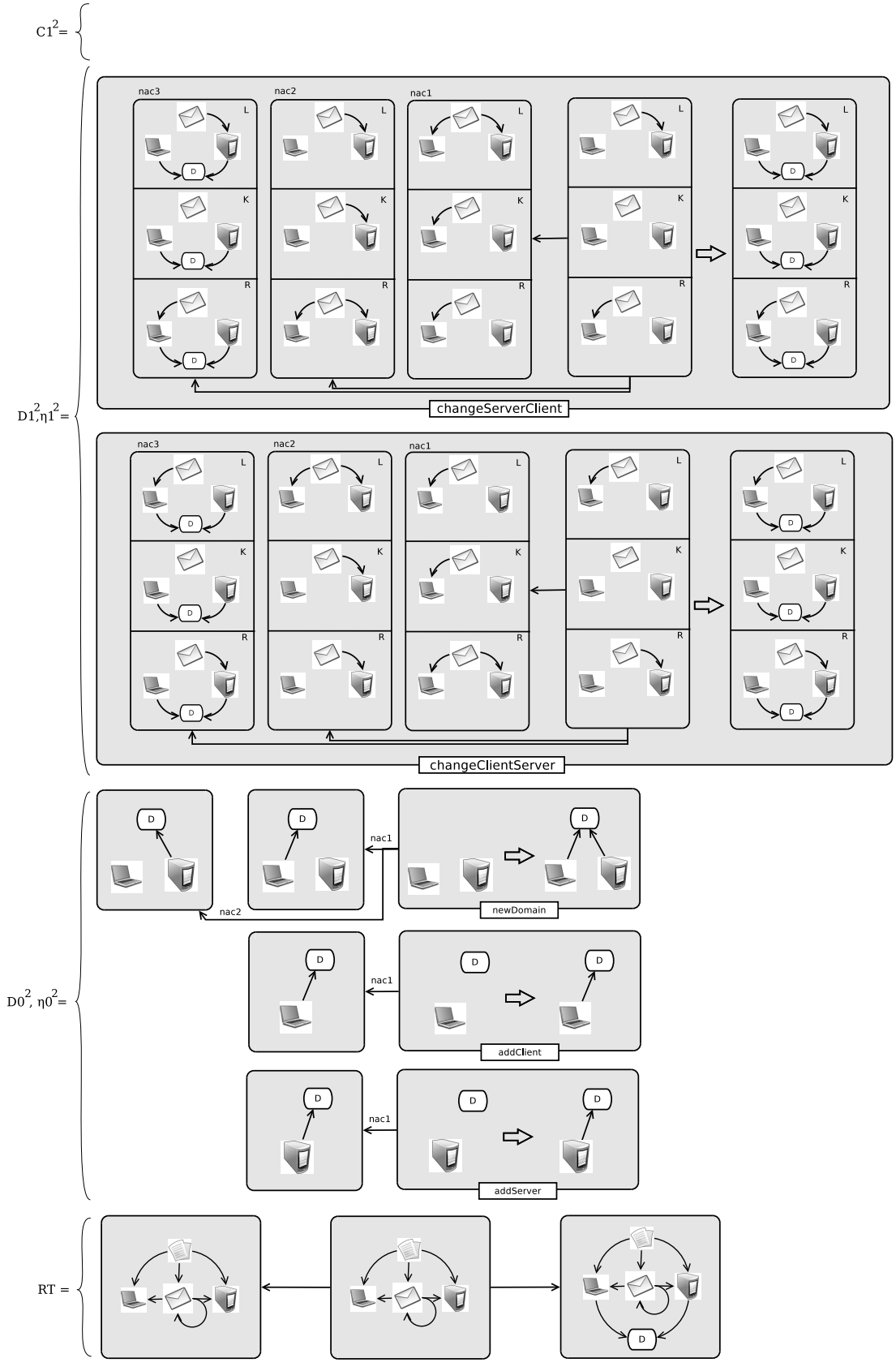


Figure 7.9: Domain aspect.

If we were using numerical attributes, we could establish a more orderly procedure, dividing servers and clients using a numerical expression involving the identification of each node. Because we are working with the simplest model without attributes, this characterization will simply put all clients and servers into exactly one domain each, but the number of domains and their correpondent clients and servers is non-deterministic. For the second-order part of the transformation, we have two 2-rules: changeClientServer *and* changeServerClient*. The pointcut of* changeClientServer *matches against rules that remove a message from a client and put it into a server, represented by the deletion and creation of edges. NACs* nac1 *and* nac2 *assure that we do not match against rules that preserve the edges, while NAC* nac3 *ensure that the modification should be run only once for each matched rule. The other rule* changeServerClient *works similarly, but for rules transferring messages from servers to clients.*

Now that we have defined what a graph grammar aspect is, we can construct a base system together with a collection of aspects, each one handling one particular requirement. We name this model an *aspect-oriented graph grammar*.

Definition 155 (Aspect-oriented graph grammar)**.** *An aspect-oriented graph grammar (AOGG) is a tuple* $(\mathcal{G}, \Delta)$ *where* $\mathcal{G}$ *is a first-order graph grammar and* Δ *is a sequence* $A_1, \ldots, A_n$ *of aspects for* $\mathcal{G}$*.*

Example 156 (Aspect-oriented graph grammar)**.** *We have an AOGG* $(\mathcal{G}, [A_1, A_2])$ *formed by the base system of Figure 7.7 together with the logging aspect* A_1 *of Figure 7.8 and the domain aspect* A_2 *of Figure 7.9.*

This definition of an AOGG quite straightforward. However, we do not have an execution model from which we obtain a weaved first-order graph grammar. The next definition shows how to *merge* a sequence of aspects into a single aspect. By combining the first-order system with the aspect to create a CDM-RC-SOGG, we can use the notion of *evolved graph grammar* to describe the result of the *aspect weaving* process.

Definition 157 (Merging of two graph grammar aspects)**.** *Let* $\mathcal{G} = (T, D_0, D_1)$ *a first-order graph grammar and* A_1, A_2 *be aspects for* $\mathcal{G}$ *such that*

$$A_1 = (ST, E_0^2, E_1^2, \theta_0^2, \theta_1^2, F_1^2)$$

and

$$A_2 = (VT, H_0^2, H_1^2, \zeta_0^2, \zeta_1^2, J_1^2)$$

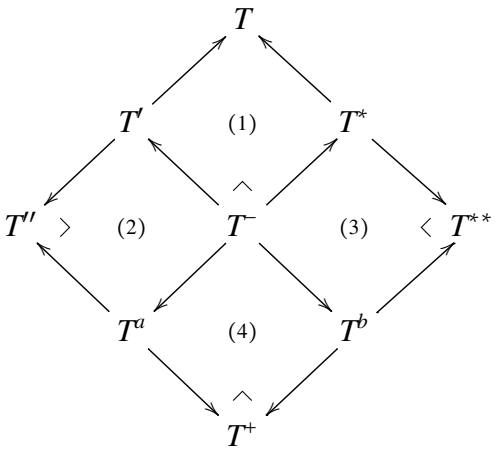
We define the merged aspect $A_1 \oplus A_2$ *for* $\mathcal{G}$ *as the tuple*

$$(RT, D_0^2, D_1^2, \eta_0^2, \eta_1^2, C_1^2)$$

calculated as follows.

- $RT = T \leftarrow T^- \rightarrow T^+$ *is obtained from* $ST = T \leftarrow T' \rightarrow T''$ *and* $VT = T \leftarrow T^* \rightarrow T^{**}$ *by*

the following steps.



1. *calculate the pullback of* $T' \rightarrow T \leftarrow T^*$ *(1)*
2. *calculate the pushout complement of* $T'' \leftarrow T' \leftarrow T^-$ *(2)*
3. *calculate the pushout complement of* $T^{**} \leftarrow T^* \leftarrow T^-$ *(3)*
4. *calculate the pushout of* $T^a \leftarrow T^- \rightarrow T^b$ *(4)*
5. *the morphisms of RT are the diagonals of squares (1) and (4).*

From this calculation, we obtain the spans $a = T'' \leftarrow T^a \rightarrow T^+$ *and* $b = T^{**} \leftarrow T^b \rightarrow T^+$.

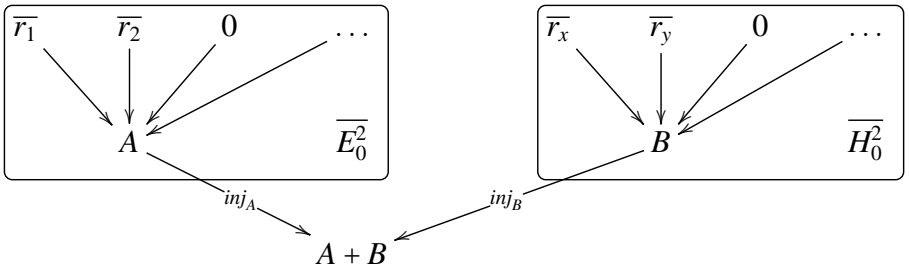
- *calculate the retypings*

$$a^{\triangleleft\triangleright}(E_0^2, E_1^2, \theta_0^2, \theta_1^2, F_1^2) = (\overline{E_0^2}, \overline{E_1^2}, \overline{\theta_0^2}, \overline{\theta_1^2}, \overline{F_1^2})$$

and

$$b^{\triangleleft\triangleright}(H_0^2, H_1^2, \zeta_0^2, \zeta_1^2, J_1^2) = (\overline{H_0^2}, \overline{H_1^2}, \overline{\zeta_0^2}, \overline{\zeta_1^2}, \overline{J_1^2})$$

converting all constructions to the same type T^+. *The object* $A + B$ *of the coproduct collection* D_0^2 *is obtained by calculating the coproduct of the vertices* A *of* $\overline{E_0^2}$ *and* B *of* $\overline{H_0^2}$ *in category* T^+-**Graph**. *The injections of* D_0^2 *are obtained by post-composition with* inj_A *(for injections of* $\overline{E_0^2}$*), and by post-composition with* inj_B *(for injections of* $\overline{H_0^2}$*, as shown in the following diagram.*



The calculation of D_1^2 *proceeds in the same way in category* T^+-**Span** *by calculating the coproducts of vertices of* $\overline{E_1^2}$ *and* $\overline{H_1^2}$. *The domains of functions* $\eta_0^2 = \overline{\theta_0^2} + \overline{\zeta_0^2}$ *and* $\eta_1^2 = \overline{\theta_1^2} + \overline{\zeta_1^2}$ *are updated in the same way.*

- $C_1^2 = \overline{F_1^2}; \overline{J_1^2}$, *i.e., the concatenation of the retyped create-delete sequences* $\overline{F_1^2}$ *and* $\overline{J_1^2}$.

The first stage of merging two aspect is to fit their types. The procedure described above obtains a span that represents *removing* all the type elements deleted either by the first of the second span, and adding the elements introduced by both. Notice that the pushout complements calculated in squares (2) and (3) are always possible: identification conditions hold because all arrows in the diagram are mono, and dangling conditions hold because we cannot represent by $T' \to T$ or $T^\star \to T$ the removal of a node without removing all incident edges. The coproduct collection of the final system is obtained by calculating the coproduct of the retyped collections involved. The retyped sequence of create-delete statements of both aspects is concatenated to compose the create-delete sequence of the final system. Now we can extend the notion of aspect merging towards a sequence of aspects.

Definition 158 (Merging of a sequence of aspects)**.** *Let* $[A_1, A_2, \ldots, A_n]$ *be a non-empty sequence of aspects for* $\mathcal{G}$*. The merged aspect from the sequence is given by*

$$(\ldots((A_1 \oplus A_2) \oplus A_3) \ldots \oplus A_n)$$

The notion of merging a sequence of aspects over the same graph grammar is simply the subsequent application of binary merging. When we have a single aspect A (possibly the result of merging several other aspects) acting over a single base system $\mathcal{G}$, we can use the underlying CDM-RC-SOGG $(\mathcal{G}, A)$ to provide an execution model for the rule transformation induced by A, and to obtain an *weaved graph grammar*. In other words, the aspect weaving operation for AOGGs is given by obtaining the *evolved gramar* from *model-transformation derivations* of the respective CDM-RC-SOGG.

Definition 159 (Weaved graph grammar)**.** *Consider a single aspect A for graph grammar* $\mathcal{G}$*. We say the first-order* $\mathcal{G}''$ *is a weaved first-order graph grammar of A over* $\mathcal{G}$ *iff* $\mathcal{G}''$ *is an* evolved grammar *from some* model-transformation derivation *of the CDM-RC-SOGG* $(\mathcal{G}, A)$.

Example 160 (Weaved graph grammar)**.** *Consider the AOGG* $(\mathcal{G}_0, [A_1, A_2])$ *formed by the base system of Figure 7.7 together with the logging aspect* A_1 *of Figure 7.8 and the domain aspect* A_2 *of Figure 7.9. The CDM-RC-SOGG* $(\mathcal{G}, A_1 \oplus A_2)$ *has the following model-transformation derivation*

$$\mathcal{G}_0 \overset{(2,g0,newDomain,m_1)}{\Longrightarrow} \mathcal{G}_1 \overset{(2,sendMSG,changeClientServer,m_2)}{\Longrightarrow} \mathcal{G}_2 \overset{(2,g0,newDomain,m_3)}{\Longrightarrow}$$
$$\mathcal{G}_3 \overset{(2,receiveMSG,addLog,m_4)}{\Longrightarrow} \mathcal{G}_4 \overset{(2,receiveMSG,changeServerClient,m_5)}{\Longrightarrow} \mathcal{G}_5 \overset{(2,sendMSG,addLog,m_6)}{\Longrightarrow}$$
$$\mathcal{G}_6 \overset{(2,getDATA,addLog,m_7)}{\Longrightarrow} \mathcal{G}_7 \overset{(2,deleteMSG,addLog,m_8)}{\Longrightarrow} \mathcal{G}_8$$

from which the evolved graph grammar *is* $\mathcal{G}_8$*. Therefore, the grammar* $\mathcal{G}_8$*, shown in Figure 7.10, is a* weaved graph grammar *for AOGG* $(\mathcal{G}_0, [A_1, A_2])$.

Notice that weaved graph grammars may not be unique. If the 2-tagged rewritings are not confluent, the we may have several distinct weaved graph grammars for the same AOGG. For instance, in the logging and domain AOGG, the domain preparation of the initial graph is non-confluent, and therefore the initial conditions of the weaved system will depend on a particular application of rules newDomain, addClient and addServer.

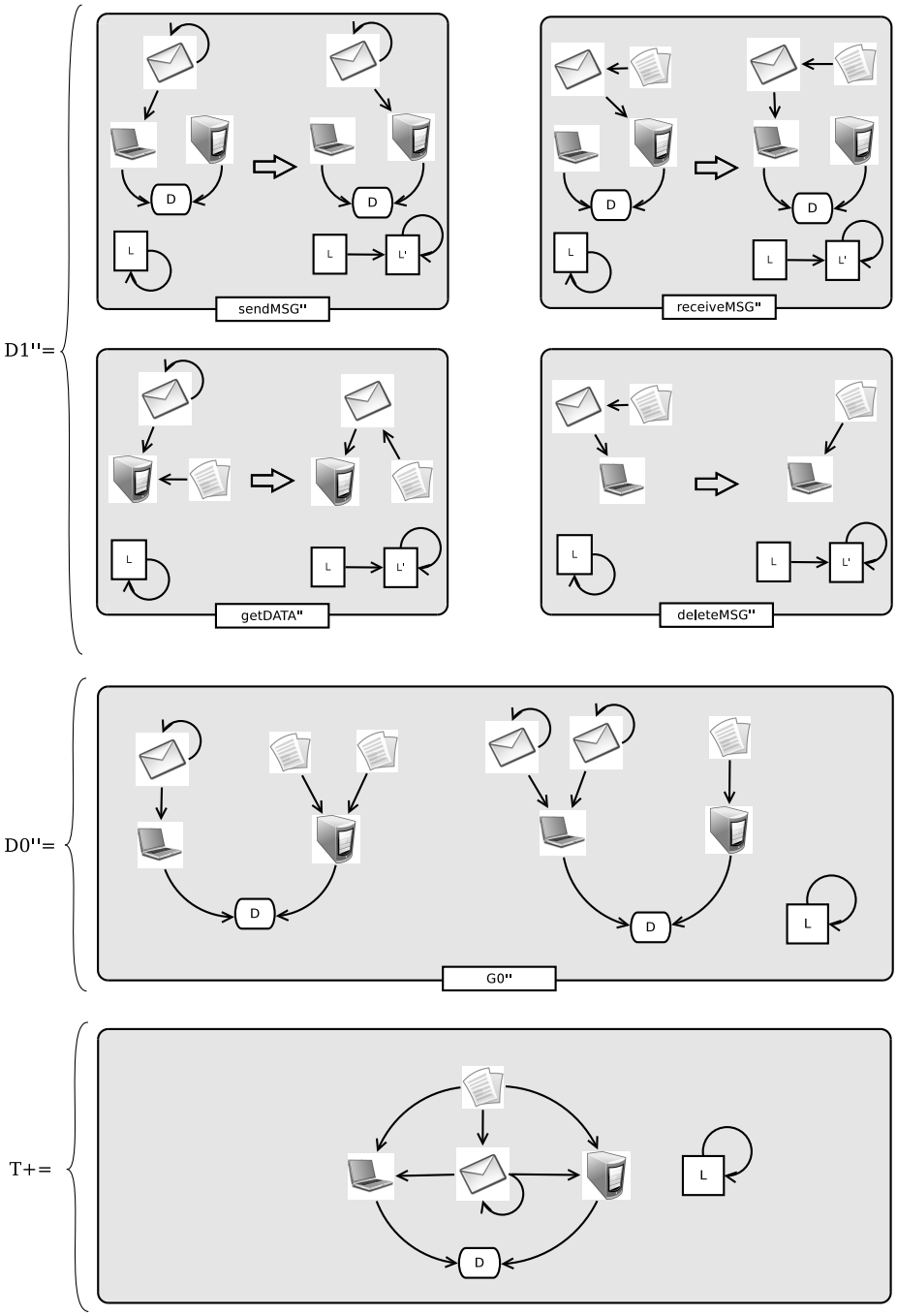


Figure 7.10: Example of weaved graph grammar: base system with logging and domains.

7.4 Analysis of aspect-oriented graph grammars

The characterization of aspects for graph grammars, and aspect-weaving through 2-tagged derivations allows us to study the aspect weaving process by the same techniques used for second-order graph grammars. We will use as running example the AOGG of the base system with logging and domains, for which the associated CDM-RC-SOGG is

$$(\mathcal{G}_0, A_1 \oplus A_2) = (T, D_0, D_1, T \leftarrow T \rightarrow T^+, D_0^2, D_1^2, \eta_0^2, \eta_1^2, \varepsilon)$$

Although we do not present full results from second-order critical pair analysis (SO-CPA) and critical pair evolution (CPE), in this section we explain how they can be use to foresee the behaviour of the model transformation and its effect on the first-order semantics. The following topics may be analysed:

Confluence of aspect weaving: because the aspect weaving comes from model-transformation derivations, we can test for confluence by checking the lack of conflicts between elements of (D_0^2, η_0^2), that implement modifications in the initial graph, and (D_1^2, η_0^2), that implement modifications in the first-order rule collection. Both analysis are calculated in second-order critical pair analysis. If we have a merged aspect and keep track of the origin of a given advice, SO-CPA results may show potential conflicts between advices of distinct aspects and also between advices from the same aspect. In our example, an easy inspection shows that there are no conflicts between changeServerClient, changeClientServer and addLog (not considering the same rule twice). Thus, the rewriting of the second-order advices is confluent due to local Church Rosser. However, there are conflicts of kind produce-forbid between `newDomain`, `addClient` and `addServer`, which are first-order advices from the domain aspect. This points to possibility of not having confluence in the rewritings of the domain aspect. On the other hand, there are no conflicts between initLog and any of the first-order advices of the domain aspect, and thus the *aspects are not conflicting*. The conclusion is that the aspect weaving process of the example is not confluent due to internal conflicts of the domain aspect, but the logging and domain aspects as a whole do not interfere with each other because there are not conflicts across their advices.

Reduction or increase in first-order rewriting possibilities: modifications in rules may affect the possibilities of rewriting. For instance, if we increase the LHS in such a way that there is not a match any longer for a given graph, we forbid a rewriting. On the other hand, if we remove some elements of the LHS, we may allow a rewriting that would otherwise fail to have a match. The first situation would be represented by means of an inter-level conflict, and the other, by an inter-level dependency. By using SO-CPA, we can calculate all inter-level critical pairs between a rule in D_1 and a 2-rule in D_1^2, and foresee those situations that affect the shape of the execution of the first-order system. For instance, let us consider the rule sendMSG of the base system and the 2-rule changeClientServer. There is a possible second-order rewriting involving them that adds a domain node and two edges to the LHS, RHS and interface of `sendMSG`. This way, a rewriting that was once possible across a server and a message is not possible if they do not belong to the same domain. This interaction would be perceived by the existence of the critical pair shown in Figure 7.11, calculated from the inter-level critical pair analysis procedure.

Reduction or increase in parallelism: another way that 2-rules may affect the first-order system is by reducing or increasing the possibility of parallel execution. We have an exam-

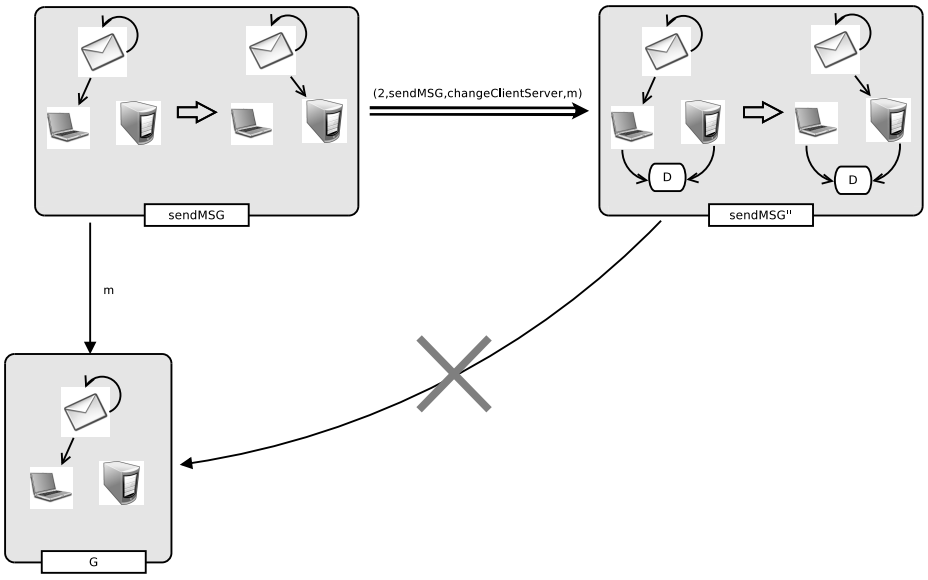


Figure 7.11: Inter-level critical pair of the example aspect-oriented graph grammar.

ple of such situation in the logging aspect: every modified rule deletes a self-edge of a log node, and creates a new log node with a self-edge, connected through another (new) edge to the original node. In the original system, there were no possible conflicts between the application of sendMSG and receiveMSG, which would always allow rewritings induced by both rules to occur simultaneously. This was the case because neither of the possible overlaps of their matches are critical pairs. In the evolved system, however, we have a potential conflict between both rules since both may try to delete the same self-edge. By means of relation $\mathscr{R}$, we obtain for each overlap in the original system both conflicting and non-conflicting overlap in the evolved system, as shown in Figure 7.12. If we consider the additional information that *only one self-edge in loops* occurs in the initial conditions, and that all rules *delete and re-create this unique self-edge*, we arrive at the conclusion that the non-conflicting overlap shown in Figure 7.12 is not possible, and therefore it only remains that all rewritings induced by sendMSG" and receiveMSG" in the evolved system are conflicting. More than this, this implementation of logging has a drastic serializing effect on the whole specification, making impossible simultaneous execution of any two first-order rules. This effect would appear only when considering the *event structure* of the resulting graph grammar, but, as we have shown, we can foresee it from critical pair evolution (CPE) together with information from the initial conditions.

This list of possibilities for analysis is, of course, not extensive. Due to the characterization of the second-order rewriting by means of adhesive HLR system with NACs, any available technique for this context could be used in order to test for other properties – termination, for instance. It would also be possible to think about adapting techniques such as model-checking from graph grammars towards second-order graph grammars, since the execution model of the former has been properly defined and it is possible to extract the required LTS in the same way we do when exploring the space state of conventional graph grammars.

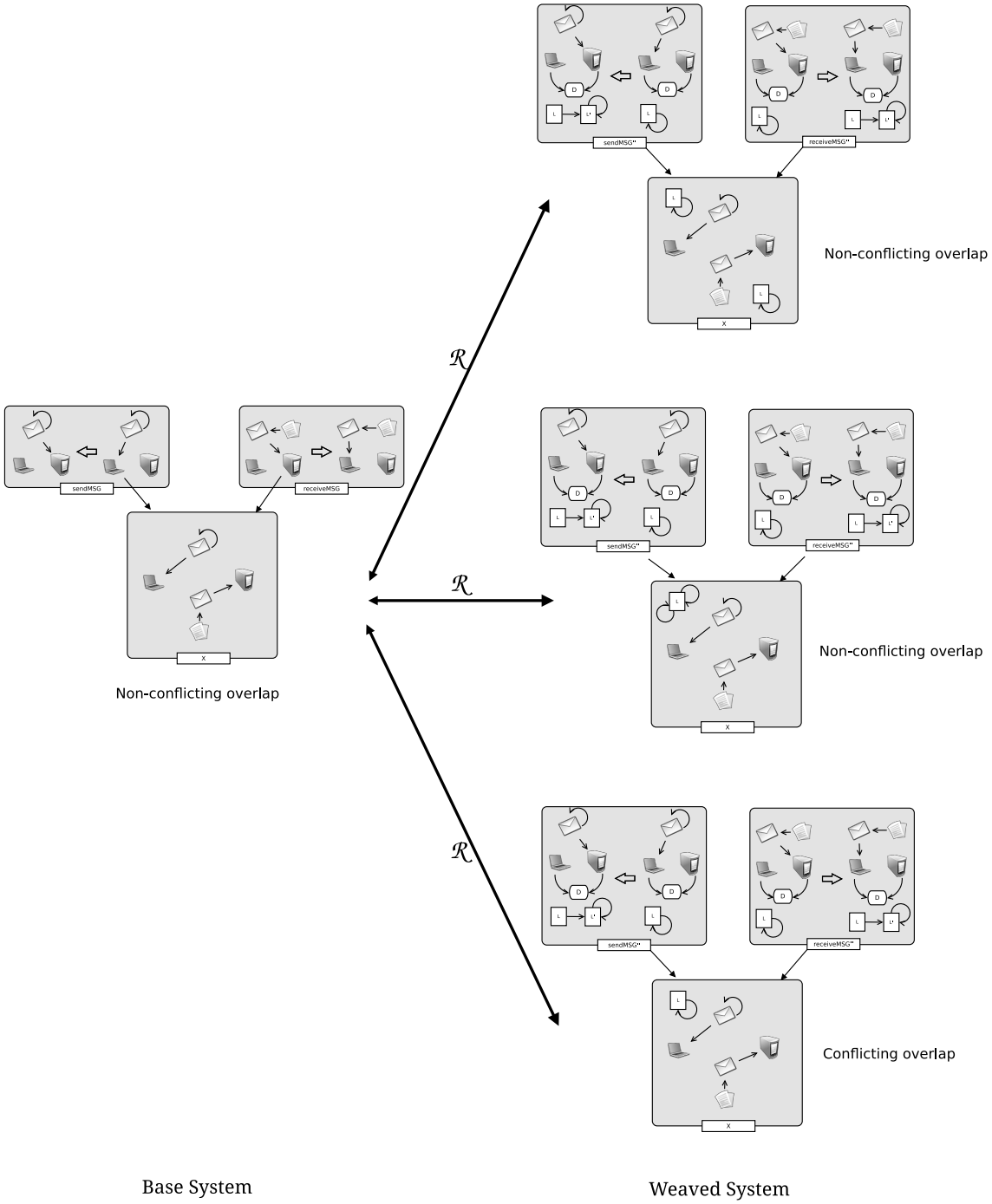


Figure 7.12: Example of evolution of rule match overlap between base and weaved system.

7.5 Summary

This chapter connected the concepts of aspect-oriented programming and graph grammars. It began by reviewing the main components of aspect-oriented systems, and then provided an analogy between aspect weaving and graph transformation. Later, we have introduced the notion of aspects for graph grammars, which gave rise to the notion of *aspect-oriented graph grammars*. In those models, the weaving process is characterized and a second-order layer acting over a first-order specification. The combination of several aspects into a single one is properly defined, as it is the result of aspect weaving by means of the notions of *model-derivation transformation* and *evolved graph grammar*. Finally, we briefly discussed how analysis techniques envisioned for second-order graph grammars could be explored to study properties of the model transformation induced by aspect weaving, and their interaction with the base system semantics.

8 RELATED WORK

In this chapter we present a set of proposals in the literature that are related to this thesis, and discuss both their similarities and distinctions. We start by comparing our approach to other proposals that define rule-based modification of rules. Subsequently, we discuss some proposals for higher-order concepts over place-transition nets. Then, we talk about the formalism known as triple graph grammar that uses triple rules that are similar in structure to our 2-rules. Finally, we present some proposals that relate the areas of aspect-oriented programming and graph transformation, either by using graph transformation as an aspect weaving mechanism or by using graph transformation tools to model and study idealized aspect-oriented languages.

8.1 Modification of graph transformation rules

Because higher-order has not been extensively studied in the context of graph transformation systems, there are few proposals to compare our approach to. This does not mean we do not find a taste of higher-order ideas in the field: concepts such as meta-modelling and rule schema strongly suggest a notion of models that affect the structure of other models. However, only few approaches treat directly the notion of rule-based modification of rules, and those are the ones we discuss in this section.

One of the first references we find in the literature is (GÖTTLER, 1999), that defines a model called *two-level graph grammars*. The intuition is the same of our second-order graph grammars: there are *meta-rules*, which are the ones that modify parts of other rules, and also *hyperrules*, representing the rules being modified. In our nomenclature, *meta-rules* would correspond to 2-rules, and *hyperrules*, to normal rules. In Göttler's work, hyperrules are said to differ from actual rules because they cannot be applied until all modifications of meta-rules are applied. That represents, in our setting, the notion of *priority* of second-order modifications over first-order modifications. The similarities, however, end in the intuitive aspect of both proposals. Göttler presents and algorithmic approach based on set theory to describe the graph rewriting approach. His approach does not correspond directly neither to the DPO approach nor to the SPO approach directly, because it mixes characteristics of both (deletion in unknown context and explicit notion of preservation). Moreover, it also presents a peculiar way of representing graph transformation rules by means of a single graph divided into region representing the left-hand side, right-hand side and connection information (in fact, two representations are introduced: Y notation and X notation). The way it rewrites rules is as follows: it encodes *hyperrules* as a graph in Y notation. Meta-rules, also represented by Y notation using a slight distinct mark (double lines), affect the hyperrules as they would affect conventional graphs. Once no more modifications are possible, hyperrules may be considered rules and be applied to graphs.

To illustrate the approach let us consider the elements depicted in Figure 8.1, adapted from (GÖTTLER, 1999). In the upper, left part of the picture we find an example of a meta-rule that deletes a node of kind *A* (leftmost part of the Y diagram), creates two nodes of kind *b* connected by an edge (rightmost part of the Y diagram) and updates the connections according to the following policy (uppermost part of the Y diagram): edges from the deleted *A* to other *A* become pairs of arrows between the non-deleted *a* and one of the new *b*s, and arrows from *A* to *c* become arrows from *c* to the other *b*. In the upper, rightmost part of the picture, we have a representation of a *hyperrule* h, which deletes two nodes and an edge of a graph and adds three nodes *c*, *d* and *e*, connected in a triangular way. There are two occurrences of *A*-typed nodes in this hyperrule: in the left-hand side region (marked as 1), and in the connectivity part (marked as 2). Hence, those are the two places in which the meta-rule m can modify h. The result of both rewritings are shown in the lower part of the picture, where in the left we have the result of applying m over the node 1 and in the right, the result of applying m over the node 2.

The difficulties of adopting the notions introduced by Göttler for the DPO approach come from its notion of rewriting, which does not match the categorical description of the rewriting process. Moreover, the notion of higher-order is obtained by means of encoding rules as graphs, and the rewriting may occur in any region of the encoded rule. In the example, the same meta-rule affected the left-hand side and in the connectivity description. This contrast with our approach using 2-rules, where modifications for each component of the rule (left-hand side, interface or right-hand side) are specified in a separated way. Moreover, we use the abstract and vastly studied generalization of the double-pushout approach, the framework of adhesive HLR system with NACs. This gives us several theoretical results regarding the second-order layer of rewriting that would have to be verified under the Göttler's notion of rewriting.

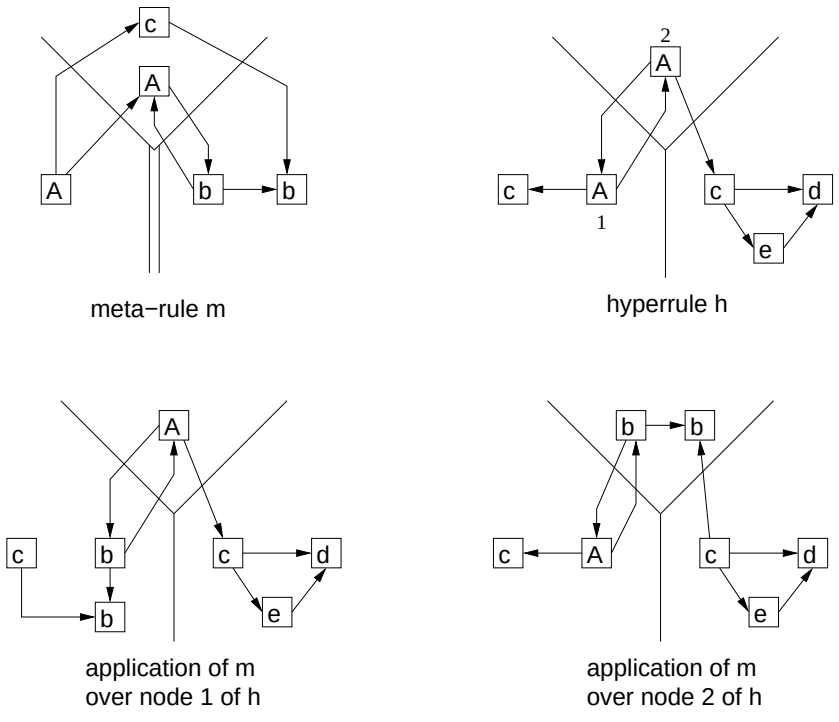


Figure 8.1: Example of meta-rule and hyperrule in Y-notation.

Parisi-Presicce (2001) also discusses modifications in graph rules and in graph grammar specifications. His work is based on the generalization of the DPO approach known as High-Level Replacement systems, which specified a series of properties a category should respect to allow the application of the DPO rewriting approach and the respective theory. Later, it was discovered that all HLR axioms could be deduced from adhesive categories, and thus the latter became the framework of choice for generalizing DPO rewriting. A HLR system is defined as a tuple $(\mathscr{C}, \mathscr{M}, P, \pi)$ where $\mathscr{C}$ is a category satisfying a set of so-called HLR axioms, $\mathscr{M}$ is a collection of monomorphisms in $\mathscr{C}$ (used to construct rules), P is a set of rule names and π is a function associating rule names to actual rule spans. Essentially, such systems consist of a context $(\mathscr{C}, \mathscr{M})$ and a named collection of rules. In this work, Parisi-Presicce discusses some ways of modifying HLR systems, which he divides in three groups:

(i) modular transformation,

(ii) global transformations and

(iii) local transformations.

Parisi-Presicce defines a notion of morphism between HLR systems that essentially consists of two elements: (i) a functor between the categories of the two systems and (ii) a mapping of rules names compatible with the functor. Based on this notion of morphism we obtain a category of HLR systems, that is used to define the two first kinds of modifications. Modular transformations compose two HLR systems H_1 and H_2 by means of a common interface H_0 through a pushout calculation of the diagram $H_1 \leftarrow H_0 \rightarrow H_2$. The idea is to combine the rules of both systems without replicating the interface H_0, as shown in diagram (a) of Figure 8.2. Global transformations follow the same idea, but they also account for the deletion of rules. This is achieved by considering DPO diagrams in the category of HLR systems as a rewriting step, as shown in diagram (b) of Figure 8.2.

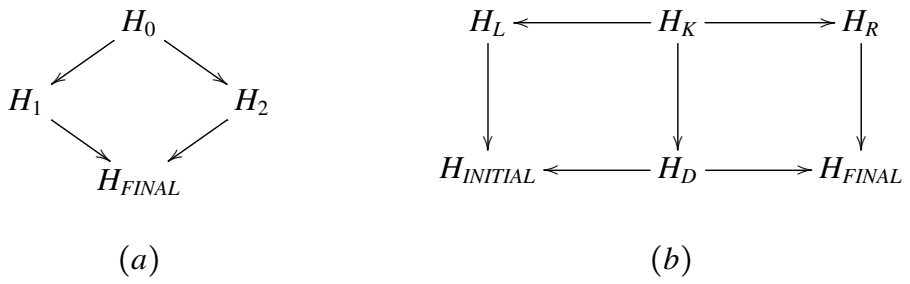


Figure 8.2: Modular transformation and global transformation.

Local transformations are of distinct nature, since they focus on modifications in the structure of rules within a given specification. Although Parisi-Presicce does not define a notion of 2-rule, he proposes to employ conventional graph rewritings over parts of another rule to modify it. He identifies three kinds of local modifications:

- *specialization:* each rewriting $G \overset{p_1,m}{\Longrightarrow} H$ where $p_1 = L_1 \leftarrow K_1 \rightarrow R_1$ produces a base span $p_2 = G \leftarrow D \rightarrow H$ where both arrows are in $\mathscr{M}$ due to HLR properties. This span, if seen as a rule, deletes and create the same elements as p, however, it requires a greater context (G) to execute. We can name p_2 as an specialization of p_1, as shown in the diagram (a) of of Figure 8.3.

- *analogy:* this case occurs when we provide a rewriting of the interface of a rule, removing some elements and replacing them for others. Given a base rule $p_1 = L_1 \leftarrow K_1 \rightarrow R_1$, and a rewriting $K_1 \stackrel{p,m}{\Longrightarrow} K_2$ on its interface where $p = L \leftarrow K \rightarrow R$, we obtain a rule $p_2 = L_2 \leftarrow K_2 \rightarrow R_2$, as show in diagram (b) of Figure 8.3.

- *inheritance:* this case corresponds to a rule $L \leftarrow K \rightarrow R$ modifying the RHS of an original rule $p_1 = L_1 \leftarrow K_1 \rightarrow R_1$, from which we calculate the pullback $PB(K_1 \rightarrow R_1 \leftarrow D) = K_1 \leftarrow K_2 \rightarrow D$ and obtain the rule $p_2 = L_2 \leftarrow K_2 \rightarrow R_2$ as shown in diagram (c) of Figure 8.3.

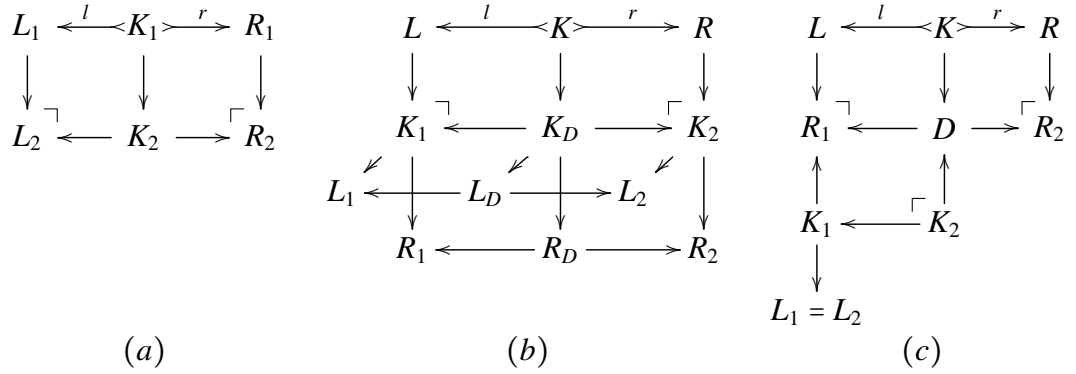


Figure 8.3: Local modifications: specialization, analogy and inheritance.

Parisi-Presicce's global and modular transformation affect essentially the collection of rules, adding or removing them without modifying their structure in any sense. In contrast, local modifications act directly over the rule structure, without mentioning how to handle sets of rules. In his approach he does not define how to combine both kinds of modification into a single framework. In our approach, both what he defines as *local modifications* and *global modifications* are simultaneously achieved by the notion and *coproduct collection rewriting*, which in turn makes use of *second-order rewriting*. Addition and removal of rules is possible through special 2-rules in CDM models, and local modifications are carried out by conventional 2-rules. From the three kinds of local modifications Parisi-Presicce mentions, two are directly translated into 2-rules notation: specialization and analogy. Inheritance, however, cannot be represented generically, since calculation of what must be deleted from K_1 to obtain K_2 depends on the match $m : L \rightarrow R_1$. In 2-rules, the amount to be deleted from the interface graph is fixed. Figure 8.4 depicts a representation for representing specialization (a) and analogy (b) as 2-rules. In diagram (c) we have a schema representing the problematic graphs (marked by "?"), that cannot be defined in advance in order to represent the modification induced by inheritance as a single 2-rule.

In Fernandez et al. (2007), a higher-order calculus to describe first-order rewriting systems such as graph grammars is proposed. However, their higher-order syntax refers to an external calculus, and not a higher-order version of graph rewriting as in our work. This approach is rather generic and allows representing other kinds of first-order rewriting models such as term graph rewriting.

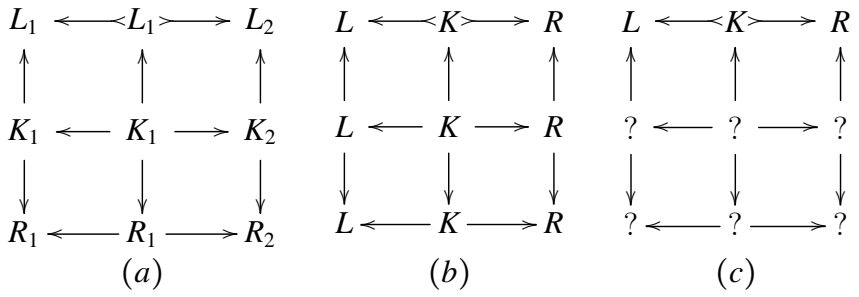


Figure 8.4: Representation of local modifications using 2-rules.

8.2 Petri-nets with dynamic structure

In the area of place-transition nets, the notion of dynamic changes in the structure has been subject of study for quite a long time. One of the first characterizations is the model known as *self-modifying nets* (VALK, 1978), where the main idea is to make the behaviour of transitions depend on the current marking. Another variation of place transition nets that allow variation in its structure is shown in (ASPERTI; BUSI, 2009). Here, concepts from lambda-calculus such as variables and scope were introduced to represent the structural variability of the model.

A model that shares similarities with SOGGs is called *reconfigurable nets* (BADOUEL; OLIVER, 1998, 1999; BADOUEL et al., 2003; LLORENS; OLIVER, 2004). This model consists of a base place-transition system equipped with a set of rules R that, when triggered, modify the structure of the net. The evolution of the system may be given either by triggering transitions or by applying rules. Notice that this is exactly the kind of situation we model with our SOGGs, where both structural modifications (second-order rewritings) and base system execution (first-order rewritings) are possible from a given state. Both the transition triggering and rule application are described using set-based terminology, as it is conventional for place-transition nets. Hence, it cannot be directly generalized for DPO graph rewriting.

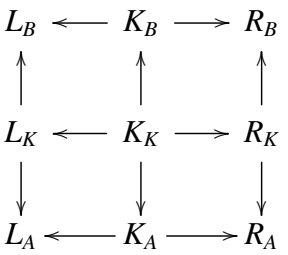
There are approaches such as (HOFFMANN; MOSSAKOWSKI; PARISI-PRESICCE, 2005; EHRIG; HOFFMANN; PADBERG, 2006) that employ *graph transformation* to describe modifications in place-transition systems. This way we may maintain the semantics of the base system by relying on an external set of graph rules to perform modifications on system structure. For example, that is exactly what we have considered for the place-transition net shown in the introduction. Following this idea, reconfigurable nets were given a algebraic foundation through the framework of (weak) adhesive HLR rewriting systems in (PRANGE et al., 2008). Later, negative application conditions for rules transforming the net structure were considered (REIN et al., 2008) by means of the framework of (weak) adhesive HLR rewriting systems with NACs (LAMBERS et al., 2008). This characterization imported several results such as Local Church Rosser, Parallelism Theorem and others to the theory of reconfigurable nets. Notice that this the same framework we have used to substantiate our notion of second-order rewriting. The important difference of our approach is that, although place-transition nets can be represented as special kinds of graph grammars, there is not a straightforward graph-based representation of a graph grammar as it is the case for place-transition nets. In some way, what we have achieved by introducing the notion of *coproduct collections* is a representation for a col-

lection of rules or graphs based on their disjoint union (coproduct). Notice that rewritings induced by *coproduct rewriting* occur within the limit of a particular injection, forbidding situations where a given rewriting would affect parts of two distinct rules in the collection. Furthermore, since our approach is based exclusively on the algebraic DPO approach for both the first-order and second-order levels, it is easier to relate both levels as we have done, for instance, in our definition of inter-level conflict.

8.3 Triple graph grammars

Triple graph grammars were originally proposed in (SCHÜRR, 1995) to address the problem of keeping consistency across changes in two models A and B that are related in a known way. For example, we can consider A to be a UML class diagram and B a relational entity-relationship diagram, where the correspondence associates classes with entities, attributes with relationship, and so on. Whenever occurs a rule-based modification in model A, we have to modify model B accordingly in order to preserve their correspondence. For such, triple graph rules are defined to be spans of spans (just like our 2-rules) for which the lower part correspond to a transformation in model A, the higher part correspond to a transformation in model B and the central part acts as an interface K between the models, as shown below.

Triple graph rule:



From the point of view of structure, triple graph rules and 2-rules are very similar. However, because the spans being rewritten in triple graph grammars are not interpreted as graph rules but rather correspondences between two models, they have different constraints. For example, the question of maintenance of injectivity of the span being rewritten is not mandatory for rewritings in triple graph grammars, unlike rule rewriting where it is so central it was necessary to develop the notion of *minimum rule-preserving set of NACs* to enforce it in *second-order rewritings*. A broader vision of the area of triple graph grammars can be found in (KINDLER; WAGNER, 2007; SCHÜRR; KLAR, 2008; HERMANN et al., 2010).

8.4 Aspects and graph rewriting

As we have mentioned previously, there have been recent approaches relating aspect-oriented concepts and the principle of graph transformation. Both (MEHNER; MONGA; TAENTZER, 2006) and (WHITTLE et al., 2009) propose to model the base system as UML diagram encoded as a graph, and aspects over it as sets of graph rules. They also propose to use of critical pair analysis to study aspect weaving: if there are conflicts between rewriting rules, the rewriting may not be confluent and thus the order aspects are combined may matter. Because of this, the modeller must ensure that distinct advices do not contain conflicts. Due to local Church Rosser, the absence of conflicts suffices to ensure confluence of the aspect weaving process. Because these proposals deal with

UML models, for which formal semantic is only recently being standardized (OBJECT MANAGEMENT GROUP, 2011), it is not easy to relate transitions of the first-order semantics with the modifications introduced by the application of advices. In our approach, since DPO rewriting is used in both levels, we managed to achieve a notion of interaction between rewriting in both levels naturally.

A different kind of interaction between aspects and graph rewriting comes from (AKSIT; RENSINK; STAIJEN, 2009), where graph rewriting is used to implement the execution rules of an idealized aspect-oriented calculus. Then, the graph rewriting tool Groove (RENSINK, 2004) is employed to generate the space-state of the rewriting, from which the aspect weaving is studied under the shape of a labelled transition system.

9 CONCLUSIONS

This thesis addresses the problem of representing the interaction between model transformation and system execution in the context of graph grammars. For such, we have employed the algebraic DPO rewriting principle to represent both modifications in graphs (first-order) and modifications in graph rules (second-order). Following this principle, we have tested some alternatives and fixed a notion of *second-order rewriting* to modify graph rules based on span rewriting with negative application conditions. Furthermore, we proposed a family of second-order graph grammar models, and introduced a notion of conflict between first-order and second order rewritings through the algebraic framework. Two analysis techniques for second-order graph grammars were proposed: the first consists of an extension of critical pair analysis, and the second consists on tracing the effect of evolution on the critical pairs of the base system.

The hypothesis we have been considering is that higher-order principles can be effective in the modelling and analysis of systems undergoing programmed modifications. Since we have proposed working with the graph transformation framework, we had to develop a new notion of second-order rewriting for the DPO approach, and based on this notion, construct second-order models. Only then we could test if such second-order systems may represent transformations adequately, and how we may study their behaviour. Here are some observations obtained from the development process of this thesis:

- *Modelling:* the proposed second-order graph grammars are quite flexible, allowing modifications of the type graph, the initial graph and also the collection of first-order rules. This is done both by performing local modifications in existing rules and by adding/removing rules to/from the rule collection. Throughout the thesis we have presented several examples of transformations such as introduction of logging, introduction of domains for message sending and even the implementation of corrections on the base system that were successfully represented as a second-order layer. It was not hard to introduce the concepts of *model-transformation derivation* and *evolutionary span* to capture the notion of model transformation over the first-order system, induced exclusively by the second-order layer of the grammar. The straightforward way that aspect-oriented constructions were represented by means of second-order graph grammars is another indication of the convenience of second-order graph grammars.

- *Analysis:* the fact we have used the same rewriting principle for both rewriting levels allowed us to achieve a notion of *inter-level conflicts and dependencies* in a rather natural way, representing the effect of second-order rewritings over first-order ones. Based on this, two static analysis techniques were proposed: one is the extension

of critical pair analysis for second-order graph grammars, and the other traces how critical pairs of the original base system relate to critical pairs of the modified system. Their expected results have been presented in the examples of the chapter that discussed aspect-oriented graph grammars.

Based on this argumentation, we claim that higher-order concepts are indeed helpful to the modelling and analysis of systems undergoing programmed modifications. By creating this notion of second-order transformation for the vastly studied DPO approach, we started a theoretical investigation of "inter-level" situations. For instance, we may investigate how to build third-order or forth-order systems, or whether the idea of higher-order as we have defined can be generalized. Those are some of the topics we foresee as future work.

9.1 Contributions

Although the notion of higher-order is pervasive in computer science, as shown by the relevance of higher-order terms in lambda calculus, historically it has not been treated in the same way within the graph transformation literature. We can find traces of it in concepts such as meta-models, rule schemes, and even direct rule-based modification of rules in restricted settings. However, the investigation of higher-order for the graph transformation paradigm is certainly not as mature such as other concepts such as, for example, the notion of conflict, application conditions for rules or the usage of attributed graphs. This fact contrasts with the literature on place-transition nets, for instance, where higher-order models have been vastly explored throughout the years. We believe this thesis, through a characterization of both second-order rewriting, second-order specification and interaction between first- and second-order rewritings, has contributed to a better understanding of higher-order constructions in the context of graph rewriting. Its usefulness has been demonstrated through the application of the proposed analysis techniques for a representation of aspects using graph grammars. Here we present a more detailed account of the original contributions proposed by thesis:

1. Discussion regarding DPO rewriting of rules in the categories T-**Span** and T-**Rules**, and definition of second-order rewriting *correcting* and *avoiding* rule invalidation.

2. Characterization of second-order rewriting by means of span rewriting with NACs, implemented through the notion of *minimal rule preservation set of NACs*, and the algorithm for calculating them based on the 2-rule structure.

3. Characterization of span rewriting with NACs in the framework of adhesive HLR systems with NACs, providing the application of standard results from DPO graph transformation to the context of second-order rewriting.

4. A notion of *coproduct collections* and *coproduct collection rewriting* to model modifiable collections of rules in first-order graph grammars.

5. Definition of a family of second-order graph grammars, comprising changes in the type graph and initial graph, and their respective operational semantics.

6. Definition of a notion of *model-transformation derivation* and *evolutionary span* to summarize changes in the whole model due to higher-order rewritings.

7. A notion of conflict between first-order and second-order graph rewritings, and two analysis techniques for second-order systems: *second-order critical pair analysis* and *critical pair evolution*.

8. A representation of aspects over first-order graph grammars as second-order layers, and the aspect weaving process by means of model-transformation derivations.

9.2 Future work

There are several questions that emerged from the development of this research and that could not be properly investigated within the limits of this text. Those topics relate to the basic framework introduced by this thesis, and range from clarifications on some particular topics to extrapolations of the second-order characterization to other contexts. The open topics that we conceived as future work are the following ones:

- *Negative application conditions in the first-order layer:* one of the assumptions we have been working with is that the rules of the first-order grammars do not have negative application conditions. The presence of application conditions in graph rules would trigger a series of questions:

  1. how to represent a collection of rules with NACs in a compatible way with coproduct rewriting?
  2. should we be able to match against rules depending on the their NACs? How could this be done?
  3. how to update first-order NACs to conform them to an arbitrary rule rewriting?
  4. how do NACs would affect the proposed analysis techniques?

  Since our notion of second-order rewriting relies heavily on on NACs, this is an interesting theoretical research track to pursue.

- *Models of third or higher order:* through the text we have focused mainly in second-order systems, since those suffice for representing model transformations of interest. However, we may consider re-using the principles we have defined to define 3-rules that would modify 2-rules, 4-rules modifying 3-rules and so on. How those layers would interact? What would be distinct from from the known situation between the first and second layers?

- *Proof of correctness of the inter-level critical pair algorithm:* although we have provided an algorithm for calculating inter-level critical pairs, we have not proved it actually captures all possible conflicting situations between a second-order and a first-order rewriting. Besides this, improvements in the efficiency of the algorithm would also be very desirable.

- *Implementation of a second-order graph grammar tool:* the investigation of higher-order in graph transformation has provided models and techniques of practical relevance since they can represent simultaneously system execution and model transformation. To perform practical experiments in large scenarios, it would be required a tool to specify, run and verify second-order graph grammars. Initial work has begun in implementing a system with those characteristics using the functional programming language Haskell. The interpreter, however, is still in early stages, and it is

not suitable for general use yet. Such implementation, when finished, would allow practical modelling and verification experiments that are not possible at the moment.

- *Investigation of SOGGs in the modelling of dynamic systems*. The main interpretation for the 2-tagged layer in this thesis was that it represented a model transformation over the base. That is why the focus has been given to *model-transformation derivations* and evolutionary spans. However, the non-deterministic nature of SOGGs also allow the modelling and execution of dynamic systems where rules transformations occur simultaneously with graph transformation. I would be an interesting scenario to explore practical situations of dynamic systems using SOGGs and apply our analysis techniques in such contexts.

- *Generalization of second-order for arbitrary adhesive HLR systems:* in this thesis we have worked mainly within the scope of typed graphs and typed graph rules. A natural sequence to this first attempt of providing higher-order is to generalize the requirements to investigate if the principles that guided us in the construction of a second-order layer for graph grammars can be reused for other scenarios that fit the framework of *adhesive HLR systems*, such as attributed graph grammars.

- *Second-order constructions in other algebraic approaches for graph rewriting:* all the formal investigation of this thesis was within the limits of the DPO approach for graph rewriting and its generalization. There comes the question if this same notions could be reproduced in other algebraic characterizations of graph rewritings, such as the single-pushout approach (LÖWE, 1993) or the sesqui-pushout approach (CORRADINI et al., 2006). Which results would still hold? Which ones would have to be modified or could not hold at all due to some particular characteristic of the rewriting mechanism?

REFERENCES

AKSIT, M.; RENSINK, A.; STAIJEN, T. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In: ACM INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD 09), 8., New York, NY, USA. **Proceedings…** ACM, 2009. p.39–50.

ALDAWUD, O.; ELRAD, T.; BADER, A. UML profile for aspect-oriented software development. In: INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED MODELING WITH UML (AOM 03), 3., Boston, USA. **Proceedings…** [S.l.: s.n.], 2003.

ASPERTI, A.; BUSI, N. Mobile petri nets. **Mathematical Structures in Computer Science**, New York, NY, USA, v.19, p.1265–1278, 2009.

ASSMANN, U.; LUDWIG, A. Aspect weaving by graph rewriting. In: INTERNATIONAL CONFERENCE ON GENERATIVE COMPONENT-BASED SOFTWARE ENGINEERING (GCSE 99), 3., Erfurt. **Proceedings…** Springer, 1999. (Lecture Notes in Computer Science, v.1799).

BADOUEL, E. et al. Modeling concurrent systems: reconfigurable nets. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS (PDPTA 03). **Proceedings…** CSREA Press, 2003. p.1568–1574.

BADOUEL, E.; OLIVER, J. **Reconfigurable Nets, a Class of High Level Petri Nets Supporting Dynamic Changes**. [S.l.]: INRIA, 1998. Research Report. (PI-1163).

BADOUEL, E.; OLIVER, J. **Dynamic Changes in Concurrent Systems**: modelling and verification. [S.l.]: INRIA, 1999. Research Report. (RR-3708).

BALASUBRAMANIAN, D. et al. The graph rewriting and transformation language: GReAT. **Electronic Communications of the EASST**, [S.l.], v.1, 2006.

BALDAN, P. et al. Unfolding grammars in adhesive categories. In: INTERNATIONAL CONFERENCE ON ALGEBRA AND COALGEBRA IN COMPUTER SCIENCE (CALCO 09), 3., Berlin, Heidelberg. **Proceedings…** Springer, 2009. p.350–366.

BALDAN, P.; KÖNIG, B.; RENSINK, A. Graph grammar verification through abstraction. In: GRAPH TRANSFORMATIONS AND PROCESS ALGEBRAS FOR MODELING DISTRIBUTED AND MOBILE SYSTEMS. **Proceedings…** [S.l.: s.n.], 2005.

BARENDREGT, H. P. Lambda calculi with types. In: ABRAMSKY, S.; GABBAY, D. M.; MAIBAUM, T. S. E. (Ed.). **Handbook of Logic in Computer Science: Vol 2. Background: Computational Structures**. Oxford, UK: Oxford University Press, 1992. p.117–309.

BERGMANN, G. et al. Incremental pattern matching in the VIATRA model transformation system. In: INTERNATIONAL WORKSHOP ON GRAPH AND MODEL TRANSFORMATIONS, New York, NY, USA. **Proceedings...** ACM, 2008. p.25–32.

BORCEUX, F. **Handbook of Categorical Algebra**. Cambridge, UK: Cambridge University Press, 1994. v.1.

CLIFTON, C.; LEAVENS, G. T. MiniMAO$_1$: an imperative core language for studying aspect-oriented reasonings. **Science of Computer Programming**, Amsterdam, The Netherlands, v.63, n.3, p.321–374, 2006.

CORRADINI, A. et al. The category of typed graph grammars and its adjunctions with categories of derivations. In: INTERNATIONAL WORKSHOP ON GRAPH GRAMMARS AND THEIR APPLICATION TO COMPUTER SCIENCE, 5., London, UK. **Proceedings...** Springer, 1996. p.56–74. (Lecture Notes in Computer Science, v.1073).

CORRADINI, A. et al. Sesqui-pushout rewriting. In: INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION (ICGT 06), 3., Natal, Rio Grande do Norte, Brazil. **Proceedings...** Springer, 2006. p.30–45. (Lecture Notes in Computer Science, v.4178).

CORRADINI, A.; MONTANARI, U.; ROSSI, F. Graph processes. **Fundamenta Informaticae**, Amsterdam, The Netherlands, v.26, n.3-4, p.241–265, 1996.

DJOKO, S. D.; DOUENCE, R.; FRADET, P. Aspects preserving properties. In: ACM SIGPLAN SYMPOSIUM ON PARTIAL EVALUATION AND SEMANTICS-BASED PROGRAM MANIPULATION, New York, NY, USA. **Proceedings...** ACM, 2008. p.135–145.

DOUENCE, R.; FRADET, P.; SÜDHOLT, M. Composition, reuse and interaction analysis of stateful aspects. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD 04), 3., New York, NY, USA. **Proceedings...** ACM, 2004. p.141–150.

EHRIG, H. et al. Adhesive high-level replacement categories and systems. In: INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION (ICGT 04), 2., Rome, Italy. **Proceedings...** Springer, 2004. v.3256.

EHRIG, H. et al. **Fundamentals of Algebraic Graph Transformation**. Berlin, Germany: Springer, 2005. (Monographs in theoretical computer science, an EATCS series).

EHRIG, H. et al. (Ed.). **Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 3: Concurrency, Parallelism, and Distribution**. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1999.

EHRIG, H. et al. (Ed.). **Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools**. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1999.

EHRIG, H.; HOFFMANN, K.; PADBERG, J. Transformations of Petri nets. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.148, n.1, p.151–172, 2006.

EHRIG, H.; PFENDER, M.; SCHNEIDER, H. J. Graph-grammars: an algebraic approach. In: ANNUAL SYMPOSIUM ON SWITCHING AND AUTOMATA THEORY, 14. **Proceedings…** [S.l.: s.n.], 1973. p.167–180.

ELLERMAN, D. A theory of adjoint functors–with some thoughts about their philosophical significance. **arXiv**, [S.l.], v.math/0511367v1, Nov 2005.

EVERMANN, J. A meta-level specification and profile for AspectJ in UML. In: INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED MODELING (AOM 07), 10., New York, NY, USA. **Proceedings…** ACM, 2007. p.21–27.

FERNÁNDEZ, M.; MACKIE, I.; PINTO, J. S. A higher-order calculus for graph transformation. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.72, n.1, p.45–58, 2007.

FERREIRA, A. P. L. **Object-Oriented Graph Grammars**. 2005. Tese (Doutorado em Ciência da Computação) — Instituto de Informática - UFRGS.

FILMAN, R. E.; FRIEDMAN, D. P. **Aspect-Oriented Programming is Quantification and Obliviousness**. [S.l.]: Research Institute for Advanced Computer Science, 2000. Technical Report.

FRAINE, B. D.; SÜDHOLT, M.; JONCKERS, V. StrongAspectJ: flexible and safe pointcut/advice bindings. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD 08), 7., New York, NY, USA. **Proceedings…** ACM, 2008. p.60–71.

FUENTES, L.; SÁNCHEZ, P. A generic MOF metamodel for aspect-oriented modelling. In: JOINT MEETING OF THE FOURTH WORKSHOP ON MODEL-BASED DEVELOPMENT OF COMPUTER-BASED SYSTEMS AND THE THIRD INTERNATIONAL WORKSHOP ON MODEL-BASED METHODOLOGIES FOR PERVASIVE AND EMBEDDED SOFTWARE (MBD/MOMPES 06), Potsdam, Germany. **Proceedings…** IEEE, 2006. p.10 pp.–124.

FUENTES, L.; SÁNCHEZ, P. Towards executable aspect-oriented UML models. In: INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED MODELING, 10., New York, NY, USA. **Proceedings…** ACM, 2007. p.28–34.

GEIS, R. et al. GrGen: a fast SPO-based graph rewriting tool. In: INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION (ICGT 06), 3., Natal, Brasil. **Proceedings…** Springer, 2006. p.383 – 397. (Lecture Notes in Computer Science, v.4178).

GOLDBLATT, R. **Topoi, the Categorial Analysis of Logic**. Mineola, NY, USA: Dover Publication, Inc., 2006. 486p.

GÖTTLER, H. Deriving productions from productions with an application to Picasso's œuvre. In: **Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools**. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1999. p.459–484.

HABEL, A.; MÜLLER, J.; PLUMP, D. Double-pushout graph transformation revisited. **Mathematical Structures in Computer Science**, New York, NY, USA, v.11, n.5, p.637–688, 2001.

HERMANN, F. et al. Formal analysis of functional behaviour for model transformations based on triple graph grammars. In: INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION (ICGT 10), 5. **Proceedings...** Springer, 2010. p.155–170. (Lecture Notes in Computer Science, v.6372).

HOFFMANN, K.; MOSSAKOWSKI, T.; PARISI-PRESICCE, F. Higher-order nets for mobile policies. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.127, p.87–105, 2005.

HUI, P.; RIELY, J. Typing for a minimal aspect language: preliminary report. In: WORKSHOP ON FOUNDATIONS OF ASPECT-ORIENTED LANGUAGES (FOAL 07), 6., New York, NY, USA. **Proceedings...** ACM, 2007. p.15–22.

JAGADEESAN, R.; JEFFREY, A.; RIELY, J. A calculus of untyped aspect-oriented programs. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP 03), Darmstadt, Germany. **Proceedings...** Springer, 2003. p.54–73. (Lecture Notes in Computer Science, v.2743).

JAGADEESAN, R.; JEFFREY, A.; RIELY, J. Typed parametric polymorphism for aspects. **Science of Computer Programming**, Amsterdam, The Netherlands, v.63, n.3, p.267–296, 2006.

JÚNIOR, J. U.; CAMARGO, V. V.; CHAVEZ, C. V. F. UML-AOF: a profile for modeling aspect-oriented frameworks. In: WORKSHOP ON ASPECT-ORIENTED MODELING (AOM 09), 13., New York, NY, USA. **Proceedings...** ACM, 2009. p.1–6.

KERSTEN, M. **AO Tools: state of the (AspectJ™) art and open problems**. Presented in the AOSD Tools Workshop in OOPSLA 2002. Available online at <http://kerstens.org/mik/publications/aoTools-ooplsa2002.pdf>. Access in January, 2012.

KICZALES, G. et al. Aspect-Oriented Programming. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP 97), 11. **Proceedings...** Springer, 1997. p.220–242. (Lecture Notes in Computer Science, v.1241).

KICZALES, G. et al. An overview of AspectJ. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP 01). **Proceedings...** Springer, 2001. p.327–353. (Lecture Notes in Computer Science, v.2072).

KINDLER, E.; WAGNER, R. **Triple Graph Grammars**: concepts, extensions, implementations, and application scenarios. Paderborn, Germany: Department of Computer Science, University of Paderborn, 2007. Technical Report. (D-33098).

KLEIN, T. et al. **From UML to Java And Back Again**. Paderborn, Germany: University of Paderbon, 1999. Technical Report.

KÖNIG, B.; KOZIOURA, V. Augur – a tool for the analysis of graph transformation systems. **Bulletin of the European Association for Theoretical Computer Science**, [S.l.], v.87, p.125–137, 2005.

LACK, S.; SOBOCIŃSKI, P. **Adhesive Categories**. Department of Computer Science, University of Aarhus: BRICS, 2003. Research Series, 25 pp. (RS-03-31).

LACK, S.; SOBOCINSKI, P. Toposes are adhesive. In: INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION (ICGT 06), 3., Natal, Brasil. **Proceedings...** Springer, 2006. p.184–198. (Lecture Notes in Computer Science, v.4178).

LAMBERS, L.; EHRIG, H.; OREJAS, F. Conflict detection for graph transformation with negative application conditions. In: INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION (ICGT 06), 3., Natal, Brazil. **Proceedings...** Springer, 2006. p.61–76. (Lecture Notes in Computer Science, v.4178).

LAMBERS, L.; EHRIG, H.; OREJAS, F. Efficient conflict detection in graph transformation systems by essential critical pairs. **Electronic Notes in Theoretical Computer Science**, Amsterdam, The Netherlands, v.211, p.17–26, 2008.

LAMBERS, L. et al. Parallelism and concurrency in adhesive high-level replacement systems with negative application conditions. **Electronic Notes in Theoretical Computer Science**, Amsterdam, The Netherlands, v.203, n.6, p.43–66, 2008.

LLORENS, M.; OLIVER, J. Structural and dynamic changes in concurrent systems: reconfigurable petri nets. **IEEE Transactions on Computers**, [S.l.], v.53, n.9, p.1147 – 1158, sept. 2004.

LÖWE, M. Algebraic approach to single-pushout graph transformation. **Theoretical Computer Science**, Essex, UK, v.109, n.1-2, p.181–224, 1993.

MACHADO, R.; FOSS, L.; RIBEIRO, L. Aspects for graph grammars. In: INTERNATIONAL WORKSHOP ON GRAPH TRANSFORMATION AND VISUAL MODELING TECHNIQUES (GT-VMT 09), 8., York, UK. **Proceedings...** ECEASST, 2009.

MACHADO, R.; HECKEL, R.; RIBEIRO, L. Modeling and reasoning over distributed systems using aspect-oriented graph grammars. In: INTERNATIONAL WORKSHOP ON RULE-BASED PROGRAMMING (RULE 09), 10., Brasília, Brazil. **Proceedings...** Open Publishing Association, 2010. p.39–50. (Electronic Proceedings in Theoretical Computer Science, v.21).

MACLANE, S. **Categories for the Working Mathematician**. New York, NY, USA: Springer, 1998. (Graduate Texts in Mathematics).

MASUHARA, H.; KICZALES, G.; DUTCHYN, C. Compilation semantics of aspect-oriented programs. In: WORKSHOP ON FOUNDATIONS OF ASPECT-ORIENTED LANGUAGES (FOAL 02), 1., Enschede, The Netherlands. **Proceedings...** Department of Computer Science: Iowa State University, 2002. v.02-06 (Technical Report).

MEHNER, K.; MONGA, M.; TAENTZER, G. Interaction analysis in aspect-oriented models. In: IEEE INTERNATIONAL CONFERENCE ON REQUIREMENTS ENGINEERING, 14. **Proceedings...** [S.l.: s.n.], 2006. p.69–78.

MENEZES, P. B.; HAEUSLER, E. H. **Teoria das Categorias para Ciência da Computação**. Porto Alegre, Brasil: Bookman, 2006. (Série Livros Didáticos – Instituto de Informática / UFRGS).

MOSTEFAOUI, F.; VACHON, J. Verification of Aspect-UML models using Alloy. In: INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED MODELING (AOM 07), 10., New York, NY, USA. **Proceedings...** ACM, 2007. p.41–48.

OBJECT MANAGEMENT GROUP. **Unified Modeling Language Version 2.0**. Available online at <http://www.omg.org/spec/UML/2.0/>. Access in January, 2012.

OBJECT MANAGEMENT GROUP. **Semantics of a Foundational Subset for Executable UML Models (FUML)**. Avaliable online at <http://www.omg.org/spec/FUML/>. Access in January, 2012.

PARISI-PRESICCE, F. On modifying high level replacement systems. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.44, n.4, p.16 – 27, 2001.

PIERCE, B. C. **Basic Category Theory for Computer Scientists**. 1.ed. Cambridge, MA, USA: The MIT Press, 1991. (Foundations of Computing).

PRANGE, U. et al. Transformations in reconfigurable place/transition systems. In: DEGANO, P.; NICOLA, R. D.; MESEGUER, J. (Ed.). **Concurrency, Graphs and Models**. [S.l.]: Springer, 2008. p.96–113.

REIN, A. et al. Negative application conditions for reconfigurable place/transition systems. In: INTERNATIONAL WORKSHOP ON GRAPH TRANSFORMATION AND VISUAL MODELING TECHNIQUES (GT-VMT 08), 7., Budapest, Hungary. **Proceedings...** ECEASST, 2008. v.10.

REISIG, W.; ROZENBERG, G. (Ed.). **Lectures on Petri Nets I: Basic Models**. 1.ed. Berlin, Germany: Springer, 1998. (Lecture Notes in Computer Science, v.1491).

RENSINK, A. The GROOVE simulator: a tool for state space generation. In: INTERNATION SYMPOSIUM ON APPLICATIONS OF GRAPH TRANSFORMATIONS WITH INDUSTRIAL RELEVANCE (AGTIVE 03), 2., Charlottesville, Virginia, USA. **Proceedings...** Springer, 2004. p.479–485. (Lecture Notes in Computer Science, v.3062).

RENSINK, A.; SCHMIDT, A.; VARRÓ, D. Model checking graph transformations: a comparison of two approaches. In: INTERNATIONAL CONFERENCE ON GRAPH TRANSFORMATION (ICGT 04), 2., Rome, Italy. **Proceedings...** Springer, 2004. p.226–241. (Lecture Notes in Computer Science, v.3256).

RIBEIRO, L. **Parallel Composition and Unfolding Semantics of Graph Grammars**. 1996. Tese (Doutorado em Ciência da Computação) — Technischen Universitat Berlin. vom Fachbereich 13 - Informatik,Berlin, de.

ROZENBERG, G. (Ed.). **Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 1: Foundations**. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1997.

RUDOLF, M. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In: INTERNATIONAL WORKSHOP ON THEORY AND APPLICATION OF GRAPH TRANSFORMATIONS (TAGT 98), 6., Paderborn, Germany. **Proceedings...** Springer, 2000. p.238–251. (Lecture Notes in Computer Science, v.1764).

SANJABI, S. B.; ONG, C.-H. L. Fully abstract semantics of additive aspects by translation. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD 07), 6., New York, NY, USA. **Proceedings…** ACM, 2007. p.135–148.

SCHAUERHUBER, A. et al. **A Survey on Aspect-Oriented Modeling Approaches**. Vienna, Austria: Vienna University of Technology, 2006. Technical Report.

SCHÜRR, A. Introduction to PROGRESS, an attribute graph grammar based specification language. In: INTERNATIONAL WORKSHOP ON GRAPH-THEORETIC CONCEPTS IN COMPUTER SCIENCE, 15., Castle Rolduc, the Netherlands. **Proceedings…** Springer, 1990. p.151–165. (Lecture Notes in Computer Science, v.411).

SCHÜRR, A. Specification of graph translators with triple graph grammars. In: INTERNATIONAL WORKSHOP ON GRAPH-THEORETIC CONCEPTS IN COMPUTER SCIENCE, 20., Herrsching, Germany. **Proceedings…** Springer, 1995. (Lecture Notes in Computer Science, v.903).

SCHÜRR, A.; KLAR, F. 15 Years of Triple Graph Grammars. In: EHRIG, H. et al. (Ed.). **Graph Transformations**. Berlin, Germany: Springer, 2008. p.411–425. (Lecture Notes in Computer Science, v.5214).

STAHL, T. et al. **Model-Driven Software Development - Technology, Engineering, Management**. 1.ed. [S.l.]: Wiley, 2006. 444p. (Wiley Software Patterns Series).

STEELE, G. L. **Common LISP: the Language**. 2.ed. Newton, MA, USA: Digital Press, 1990.

TAENTZER, G. AGG: a tool environment for algebraic graph transformation. In: INTERNATIONAL WORKSHOP ON APPLICATIONS OF GRAPH TRANSFORMATIONS WITH INDUSTRIAL RELEVANCE (AGTIVE 99), Kerkrade, The Netherlands. **Proceedings…** Springer, 2000. p.481–488. (Lecture Notes in Computer Science, v.1779).

TAENTZER, G. et al. Model transformations by graph transformations: a comparative study. In: MODEL TRANSFORMATIONS IN PRACTICE WORKSHOP AT MODELS 2005, Montego Bay, Jamaica. **Proceedings…** Springer, 2005. p.05. (Lecture Notes in Computer Science, v.3713).

TUCKER, D. B.; KRISHNAMURTHI, S. **A Semantics for Pointcuts and Advice in Higher-Order Languages**. Providence, Rhode Island, USA: Department of Computer Science, Brown University, 2002. Technical Report. (CS-02-13).

VALK, R. Self-modifying nets, a natural extension of Petri nets. In: COLLOQUIUM ON AUTOMATA, LANGUAGES AND PROGRAMMING, 5., Udine, Italy. **Proceedings…** Springer, 1978. p.464–476. (Lecture Notes in Computer Science, v.62).

WAND, M.; KICZALES, G.; DUTCHYN, C. A semantics for advice and dynamic join points in aspect-oriented programming. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, New York, NY, USA, v.26, n.5, p.890–910, 2004.

WHITTLE, J. et al. MATA: a unified approach for composing UML aspect models based on graph transformation. **Transactions on Aspect-Oriented Software Development**, [S.l.], v.6, p.191–237, 2009. (Lecture Notes in Computer Science, v.5560).

WHITTLE, J.; JAYARAMAN, P. K. MATA: a tool for aspect-oriented modeling based on graph transformation. In: INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED MODELING (AOM 07), 11., Nashville, TN, USA. **Proceedings…** Springer, 2007. p.16–27. (Lecture Notes in Computer Science, v.5002).

ZHANG, G. Towards Aspect-Oriented Class Diagrams. In: ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE, 12., Washington, DC, USA. **Proceedings…** IEEE Computer Society, 2005. p.763–768.

APPENDIX A – CATEGORY THEORY

Categories, morphisms and related concepts are used by the algebraic approach to graph transformation as a common language. Because of this, we provide in this appendix a reference for categorical concepts required for reading this work. A category is a mathematical structure that contains objects, i.e. mathematical entities such as sets, numbers and graphs equipped with a notion of *morphism*. Intuitively, morphisms can be seen as structure preserving transformations: functions, relations, graph homomorphisms, etc. The theory focus mainly in the morphisms and their relationship, being very useful to describe, characterize and study mathematical constructions in a generic and abstract way, independent of the internal object structure. It has a large influence in areas of Computer Science such as type theory, logic, functional programming languages, concurrency models and graph transformation. For an intuitive introduction to the subject, we suggest (PIERCE, 1991) and (MENEZES; HAEUSLER, 2006). A comprehensive presentation of the area given by the classic reference (MACLANE, 1998).

A.1 Basic definitions

Definition 161 (Category)**.** *A category* **C** *is a tuple* $\langle O, M, \partial_0, \partial_1, i, \circ \rangle$*, where*

- O *is a collection of objects;*
- M *is a collection of morphisms (or arrows) between objects;*
- $\partial_0 : M \to O$ *is a total function taking each morphism to its source object;*
- $\partial_1 : M \to O$ *is a total function taking each morphism to its target object;*
- $i : O \to M$ *defines an identity* i_A *to each object* $A \in O$*;*
- $\circ : M \times M \rightharpoonup M$ *is a partial binary operation named morphism composition.*

such that the following holds:

1. *composition is closed for compatible maps*

$$\partial_1(f) = \partial_0(g) \iff g \circ f \in M$$

   and it is associative

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

2. *identities are neutral elements of the composition:*

$$f \circ i_A = f = i_B \circ f$$

Notation 162. We may write both $g \circ f : A \to C$ or $f; g : A \to C$ to denote the composition of arrows $f : A \to B$ and $g : B \to C$. The syntax $\mathbf{C}[A, B]$ denotes the collection of all morphisms from A to B in category $\mathbf{C}$.

Example 163 (Set-based categories)**.** *The categories below are constructed using sets or structured sets.*

- **Set** : *sets as objects, total functions as arrows.*
- **Pfn** : *sets as objects, partial functions as arrows.*
- **Poset** : *partially ordered sets as objects, monotonic functions as arrows.*
- **Mon** : *monoids as objects, monoid homomorphisms as arrows.*

Some properties of functions such as being injective, surjective or bijective are generalized in Category Theory by the concepts of monomorphisms, epimorphisms and isomorphism, which are respectively analog to the original properties in the category **Set**. In the graph categories we address in this work the analogy also holds, however, it is important to remark that there are categories where this is not the case. As an example, we cite that epimorphisms in **Mon** are not necessarily surjective.

Definition 164 (Monomorphism)**.** *A $\mathscr{C}$-morphism $f : A \to B$ is mono, also called a monomorphism, iff $\forall g, h : X \to A,\ f \circ g = f \circ h \to g = h$*

$$X \underset{h}{\overset{g}{\rightrightarrows}} A \xrightarrow{f} B$$

Definition 165 (Epimorphism)**.** *A $\mathscr{C}$-morphism $f : A \to B$ is epi, also called an epimorphism, iff $\forall g, h : B \to X,\ g \circ f = h \circ f \to g = h$*

$$A \xrightarrow{f} B \underset{h}{\overset{g}{\rightrightarrows}} X$$

Notation 166. In diagrams, mono arrows are drawn with a tail: $A \rightarrowtail B$ and epi arrows are drawn with a double head: $A \twoheadrightarrow B$.

Definition 167 (Isomorphism)**.** *A $\mathscr{C}$-morphism $f : A \to B$ is iso, also called an isomorphism, iff $\exists f^{-1} : B \to A$ such that $f \circ f^{-1} = i_B$ and $f^{-1} \circ f = i_A$. If there is an isomorphism between A and B, we say that they are isomorphic objects.*

The definition of isomorphism rely on the existence of an inverse morphism. It is know that, if this inverse transformation exists, it is unique. Notice that if an arrow is iso it implies that it is both mono and epi. The converse, i.e. being mono and epi implies iso, does not hold in general categories, although it is known to be true in some categories such as **Set**.

In Category Theory, operations are specified by means of limits and colimits of diagrams, which can be seen as particular cases of *universal arrows*. The following definitions introduce the most common limits and colimits used throughout the text.

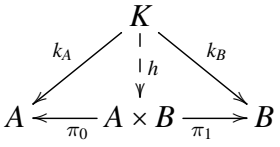
Definition 168 (Terminal object)**.** *An $\mathscr{C}$-object A is terminal iff $\forall X \in \mathscr{C}, \exists! f : X \to A$.*

Definition 169 (Initial object)**.** *An $\mathscr{C}$-object A is initial iff $\forall X \in \mathscr{C}, \exists! f : A \to X$*

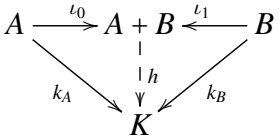
Notation 170. In diagrams, unique arrows are written with dashed line: $A \dashrightarrow B$. The initial object is usually denoted 1, and the initial object is usually denoted 0.

Notice that terminal and initial objects in a category do not need to be unique. However, if there are several terminal objects, all of them are isomorphic, and the same holds for initial objects. Because of this, we say that terminal and initial objects are *unique up-to-isomorphism*. As an example, all singletons in **Set** are terminal. On the other hand, there is a unique initial object in **Set**: the empty set $\varnothing$.

Definition 171 (Product)**.** *Let A and B be objects of $\mathcal{C}$. Every object K together with morphisms $k_A : K \to A$ and $k_B : K \to B$ is a pre-product of A and B. A product is a pre-product $\langle A \times B, \pi_0, \pi_1 \rangle$ such that for all pre-product $\langle K, k_A, k_B \rangle$ there is a unique arrow $h : K \to A \times B$ such that $\pi_0 \circ h = k_A$ and $\pi_1 \circ h = k_B$. This can be seen by the following diagram:*

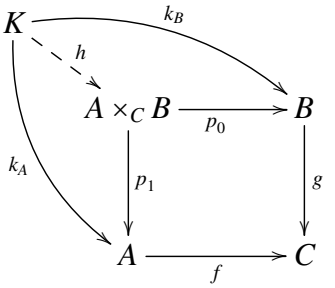


Definition 172 (Coproduct)**.** *Let A and B be objects of $\mathcal{C}$. Every object K together with morphisms $k_A : A \to K$ and $k_B : K \to B$ is a pre-coproduct of A and B. A coproduct is a pre-product $\langle A + B, \iota_0, \iota_1 \rangle$ such that for all pre-product $\langle K, k_A, k_B \rangle$ there is a unique arrow $h : A + B \to K$ such that $h \circ \iota_0 = k_A$ and $h \circ \iota_1 = k_B$. This can be seen by the following diagram:*



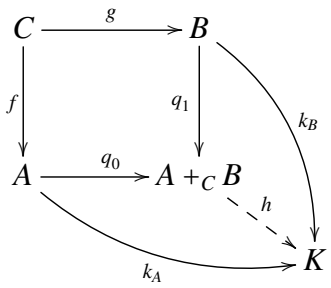
Definition 173 (Product and coproduct of arrows)**.** *Given two morphisms $f : A \to C$ and $g : B \to D$, the arrow product of f and g is defined as the unique arrow $f \times g : A \times B \to C \times D$ between the products of domains and codomains. Conversely, the arrow coproduct of f and g is defined as the unique arrow $f + g : A + B \to C + D$.*

Definition 174 (Pullback)**.** *Let $f : A \to C$ and $g : B \to C$ be morphisms of $\mathcal{C}$. Every object K together with morphisms $k_A : K \to A$ and $k_B : K \to B$ such that $f \circ k_A = g \circ k_B$ is a pre-pullback of $A \xrightarrow{f} C \xleftarrow{g} B$. A pullback is a pre-pullback $\langle A \times_C B, p_0, p_1 \rangle$ such that for each pre-pullback $\langle K, k_A, k_B \rangle$ there is a unique arrow $h : K \to A \times_C B$ such that $p_0 \circ h = k_A$ and $p_1 \circ h = k_B$, as shown in the following diagram:*



Definition 175 (Pushout)**.** *Let $f : C \to A$ and $g : C \to A$ be morphisms of $\mathcal{C}$. An object K together with morphisms $k_A : A \to K$ and $k_B : B \to K$ such that $k_A \circ f = k_B \circ g$ is a pre-pushout*

of $A \xleftarrow{f} C \xrightarrow{g} B$. A pushout is a pre-pushout $\langle A +_C B, q_0, q_1 \rangle$ such that for each pre-pushout $\langle K, k_A, k_B \rangle$ there is a unique arrow $h : A +_C B \to K$ such that $p_0 \circ h = k_A$ and $p_1 \circ h = k_B$, as shown in the following diagram:



It is know that limits and colimits, if existent, are unique up-to-isomorphism, i.e. if they are not unique, at least all candidates have isomorphic objects.

Definition 176 (Functor)**.** *Let $\mathscr{C} = \langle O, M, \partial_0, \partial_1, i, \bullet \rangle$ and $\mathscr{D} = \langle O', M', \partial'_0, \partial'_1, i', \bullet' \rangle$ be categories. A functor $f : \mathscr{C} \to \mathscr{D}$ is a pair (f_O, f_M) of total functions $f_O : O \to O'$ and $f_M : M \to M'$ such that*

1. *sources and targets are preserved: $\partial'_0 \circ f_M = f_O \circ \partial_0$ and $\partial'_1 \circ f_M = f_O \circ \partial_1$*
2. *identities are preserved: $i' \circ f_O = f_M \circ i$*
3. *morphism composition is preserved: $f_M(g \bullet f) = f_M(g) \bullet' f_M(f)$*

Definition 177 (Natural transformation)**.** *Let $F, G : \mathscr{C} \to \mathscr{D}$ be functors. A natural transformation $\eta : F \to G$ is a collection of D-arrows $\eta_X : F(X) \to G(X)$ (one arrow for each $\mathscr{C}$-object X) such that the following diagram commutes for each $\mathscr{C}$-morphism $f : X \to Y$.*

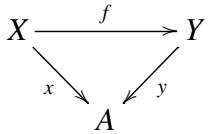
$$\begin{array}{ccc} F(X) & \xrightarrow{\eta_X} & G(X) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(Y) & \xrightarrow[\eta_Y]{} & G(Y) \end{array}$$

If all η_X are isomorphisms in $\mathscr{C}$, we call η a natural isomorphism*.*

Definition 178 (Functor category)**.** *Let $\mathscr{C}$ and $\mathscr{D}$ be categories. The functor category $[\mathscr{C} \to \mathscr{D}]$ is defined by taking as objects all functors between $\mathscr{C}$ and $\mathscr{D}$, and as morphisms all natural transformations between functors.*

Definition 179 (Slice Category)**.** *Let A be an object of category $\mathscr{C}$. The* slice category *$\mathscr{C} \downarrow A$ is formed as follows:*

- *objects of $\mathscr{C} \downarrow A$ are arrows $x : X \to A$ from some object $X \in \mathscr{C}$ towards object A;*
- *an arrow between two objects $x : X \to A$ and $y : Y \to A$ is an arrow $f : X \to Y \in \mathscr{C}$ such that the following diagram commutes:*



- *identity of $x : X \to A$ is id_X;*
- *composition in $\mathscr{C} \downarrow A$ follows from composition in $\mathscr{C}$.*

A.2 Adjunctions

Adjunctions are said by many authors to be the most profound contribution of category theory to mathematics in general. This is due to the fact that adjunctions generalize a great extent of other constructions, and thus can be seen in practically every situation involving limits. For being so abstract, it is also difficult to explain the intuition behind adjunctions, since there is not one single interpretation of the concept. A possible interpretation lies in the idea of "naturally inverse" transformations between categories. Let us try to clarify this last statement by a more detailed explanation, whose main ideas were taken from (ELLERMAN, 2005).

Initially, consider two categories, $\mathscr{C}$ and $\mathscr{D}$. Each category consists of a "world" of objects that have a specific structure or property, and the morphisms can be seen as transformations of some kind that preserve such structure. For instance, in categories in which objects are sets with some structure, such as **Poset** and **Mon**, morphisms are transformations between the carrier sets of the objects that preserve their structure (also known as homomorphisms). According to this view, category theory can be seen as a theory of structure-preserving maps. A functor $F : \mathscr{C} \to \mathscr{D}$ is a transformation between elements of $\mathscr{C}$ (objects and morphisms) that preserve the categorical structure of $\mathscr{C}$, i.e. identities, composition, sources and targets of morphisms. By means of functors, one can define transformations between categories that do not hurt the property of being a category. However, there is no categorical construction that allows us to directly define a map $f : c \to d$ if c and d are in different categories, ($c \in \mathscr{C}$ and $d \in \mathscr{D}$, for example). On the other hand there may be a way to project f over a category, allowing us to describe f as a homomorphism, and, by doing so, to employ the categorical machinery on it. This idea can be implemented in more than one way:

- transform both $\mathscr{C}$ and $\mathscr{D}$ by means of functors to a more general category (usually **Set**), and describe f there.
- transform d to $\mathscr{C}$ by means of a functor G, and then represent f by a function from c to $G(d)$;
- transform c to $\mathscr{D}$ by means of a functor F, and then represent f by a function from $F(c)$ to d.

Let c be a $\mathscr{C}$-object, d a $\mathscr{D}$-object, $F : \mathscr{C} \to \mathscr{D}$ and $G : \mathscr{D} \to \mathscr{C}$ functors. The functors F and G are said to be adjoints from $\mathscr{C}$ to $\mathscr{D}$ iff every morphism $f : c \to d$ has a unique representation in both $\mathscr{C}$ and $\mathscr{D}$. The representation of f in $\mathscr{C}$ is an arrow $f_{\mathscr{D}} : c \to G(d)$ and, in $\mathscr{D}$, an arrow $f_{\mathscr{C}} : F(f) \to d$. One can see the two transformations F and G as "inverses" from the perspective of such morphisms, i.e. one can uniquely define f by taking its source to $\mathscr{D}$ and using $f_{\mathscr{D}}$, or, alternatively, by taking its target to $\mathscr{C}$ and using $f_{\mathscr{C}}$. The diagram shown in Figure 9.1 presents this correspondence.

Since arrows like f do not "exist directly" as morphisms in the sense that they do not belong to $\mathscr{C}$ neither to $\mathscr{D}$, they can be represented by both functors F and G and by the bijection that associates the projection $f_{\mathscr{C}}$ to the projection $f_{\mathscr{D}}$. These three elements define and adjunction (or adjoint situation) between the categories $\mathscr{C}$ and $\mathscr{D}$, or, alternatively, the F functor is said to be the left-adjoint of G, and G is the right-adjoint of F. It is also important to note that this characterization occurs in a specific direction, from $\mathscr{C}$ to $\mathscr{D}$. It is possible to exist adjunctions from $\mathscr{D}$ to $\mathscr{C}$, but this would be another situation and may involve other functors.

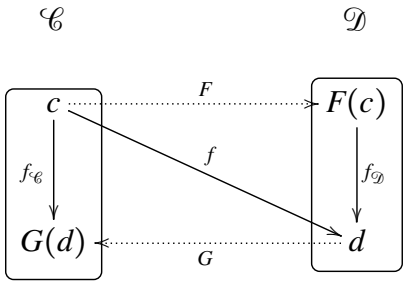


Figure 9.1: Transformation f represented in $\mathscr{C}$ and in $\mathscr{D}$.

The bijection between $f_{\mathscr{C}}$ and $f_{\mathscr{D}}$ can be described in more than one way: over a common base-category (case 1); over the category $\mathscr{C}$ (case 2); or over the category $\mathscr{D}$ (case 3). Consequently, there are three different (but equivalent) definitions of adjunction.

Definition 180 (Adjunction – version 1)**.** *An adjunction is a triple* (F, G, ψ) *,where*

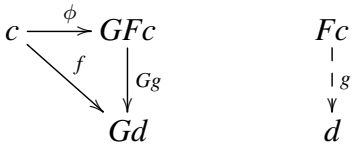
- $F : \mathscr{C} \to \mathscr{D}$ *is a functor from category* $\mathscr{C}$ *to category* $\mathscr{D}$
- $G : \mathscr{D} \to \mathscr{C}$ *is a functor from category* $\mathscr{D}$ *to category* $\mathscr{C}$
- $\psi : Hom(F\_, \_) \simeq Hom(\_, G\_)$ *is a natural isomorphism in* **Set**

This first definition of adjunction can be seen as closer to the presented intuition since the bijection is characterized by an isomorphism in **Set**. However, it relies on the existence of *Hom* functors[1] $Hom(F\_, \_) : \mathscr{D}^{op} \times \mathscr{D} \to$ **Set** and $Hom(\_, G\_) : \mathscr{C}^{op} \times \mathscr{C} \to$ **Set**, that only exists for locally-small categories. The following two characterizations of adjunction are less-intuitive, but also more general since they hold for categories whose collections of arrows between two objects may not constitute sets.

Definition 181 (Adjunction – version 2)**.** *An adjunction is a triple* (F, G, ϕ)*, where*

- $F : \mathscr{C} \to \mathscr{D}$ *is a functor from category* $\mathscr{C}$ *to category* $\mathscr{D}$
- $G : \mathscr{D} \to \mathscr{C}$ *is a functor from category* $\mathscr{D}$ *to category* $\mathscr{C}$
- $\phi : \mathscr{I}_{\mathscr{C}} \to G \circ F$ *is a natural transformation from the identity functor of category* $\mathscr{C}$ *to the composition of F and G. The natural transformation* ϕ *is called the* unit *of the adjunction.*

such that, for every $\mathscr{C}$*-morphism* $f : c \to G(d)$*, there exists only one* $\mathscr{D}$*-morphism* $g : F(c) \to d$ *that makes the following diagram commute*

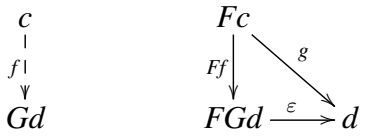


Definition 182 (Adjunction – version 3)**.** *An adjunction is a triple* (F, G, ε)*, where*

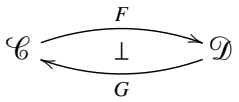
[1] In a locally-small category $\mathscr{C}$, the functor $Hom(a, b) : \mathscr{C} \to$ **Set** takes the set of arrows from a to b into their representation in **Set**. For the precise definition of *Hom*-functors, natural transformations, natural isomorphisms and universality, we recommended (MACLANE, 1998).

- $F : \mathscr{C} \to \mathscr{D}$ *is a functor from category* $\mathscr{C}$ *to category* $\mathscr{D}$
- $G : \mathscr{D} \to \mathscr{C}$ *is a functor from category* $\mathscr{D}$ *to category* $\mathscr{C}$
- $\varepsilon : F \circ G \to \mathscr{I}_{\mathscr{D}}$ *is a natural transformation from the composition of G and F to the identity functor of category* $\mathscr{D}$*. The natural transformation* ε *is called the* co-unit *of the adjunction.*

such that, for every $\mathscr{D}$*-morphism* $g : F(c) \to d$*, there exists only one* $\mathscr{C}$*-morphism* $f : c \to G(d)$ *that makes the following diagram commute*



These two alternative definitions are based in two natural transformations, ϕ and ε, which are called unit and co-unit, respectively. Moreover, for all $c \in \mathscr{C}$ and $d \in \mathscr{D}$, the transformation $\phi : c \to G \circ F(c)$ is *universal* from c to G and, dually, the co-unit $\varepsilon : F \circ G(d) \to d$ is co-universal from F to d. This universality is what allows the unit and co-unit to be interpreted as the projections of the bijection between arrows $f_{\mathscr{C}}$ and $f_{\mathscr{D}}$ in, respectively, $\mathscr{C}$ and $\mathscr{D}$. Moreover, they define each other up to isomorphism, i.e. one can calculate the co-unit from the unit, and vice-versa. The expression $F \dashv G$ is normally used to represent adjunctions, where the side of the dash always points to the left adjoint. This notation can also be used in diagrams, as shown below.



Adjunctions can be classified according to properties of the unit and co-unit. When comparing models, two kinds of adjunction are of special interest: reflections and co-reflections.

Definition 183 (Reflection)**.** *A reflection is an adjunction* $(F \dashv G, \phi, \varepsilon)$ *from* $\mathscr{C}$ *to* $\mathscr{D}$ *where the co-unit* ε *is a natural isomorphism.*

Definition 184 (Co-Reflection)**.** *A co-reflection is an adjunction* $(F \dashv G, \phi, \varepsilon)$ *where the unit* ϕ *is a natural isomorphism.*

If an adjunction $F \dashv G$ from $\mathscr{C}$ to $\mathscr{D}$ is a reflection, it means that the functor G is a full embedding, i.e. it is full and faithfull. In other words, the category $\mathscr{D}$ is equivalent to a full subcategory of $\mathscr{C}$. If it is a co-reflection, the inverse holds: F is a full embedding and $\mathscr{C}$ is equivalent to a full subcategory of $\mathscr{D}$. If the adjunction is both a reflection and a co-reflection, it is fact an equivalence of categories, i.e. both $\mathscr{C}$ and $\mathscr{D}$ are two representations for essentially the same universe. One important characteristic of adjoint functors is the preservation of universals across the involved categories. Left adjoints preserve colimits, while right adjoints preserve limits. This is a very useful property, since it allows us to reuse limits from known models directly in more obscure and complicated ones by simply finding a suitable adjunction between their categories. Other good characteristic is that adjunctions are composable, meaning that we have transitivity in such preservation of universals.

APPENDIX B – RESUMO ESTENDIDO

Este capítulo apresenta um resumo em português dos tópicos abordados ao longo desta tese. Para cada capítulo, apresentamos uma seção descrevendo as principais ideias e contribuições.

B.1 Introdução

O Capítulo 1 introduz o tema geral desta tese, que se refere à modelagem e análise de sistemas sob evolução programada, O ponto central é identificar como modificações estruturais afetam o comportamento de um dado modelo de sistema. Propomos a utilização do paradigma de transformação de grafos, muito usado para especificar linguagens visuais, para representar simultaneamente a execução e a evolução do sistema base. Desta forma, podemos utilizar o mesmo arcabouço formal, no caso a abordagem de reescrita de grafos por pushout duplo, para estudar como a evolução e a execução interagem. A hipótese que consideramos é que modelos baseados em reescrita de grafos com alta ordem são adequados para modelagem e análise de sistemas sob evolução programada.

B.2 Sistemas de transformação de grafos

O Capítulo 2 faz uma revisão dos principais conceitos da área de transformação de grafos. Modelos baseados em regras de transformação de grafos são bastante usados para descrever a sintaxe e a semântica de linguagens visuais, i.e., baseadas em diagramas. A teoria que fundamenta a abordagem algébrica para transformação de grafos é bastante desenvolvida, utilizando construções de *teoria das categorias* para descrever tanto as regras de transformação quanto a aplicação destas. A vantagem desta abordagem é que os principais resultados teóricos se tornam gerais, sendo válidos para uma vasta gama de modelos relacionados que manipulam grafos simples, grafos tipados, grafos com atributos, entre outros. Dentro da abordagem algébrica, existem variações que diferem essencialmente pela forma de representar as regras e o processo de reescrita. Por exemplo, na variação de pushout duplo, regras são spans de grafos formados por homomorfismos injetores (Definição 11), o *match* de uma regra a um grafo é dado por um homomorfismo e a aplicação de regras é representado por um diagrama categorial contendo dois pushouts (Definição 15). Gramáticas de grafos são modelos formais compostos por um grafo inicial e um conjunto de regras de reescrita de grafos (Definição 18). O comportamento de uma gramática de grafos é definido pela aplicação de regras de reescrita, determinada pela presença ou não de *matches* para as regras (Definição 20). Gramáticas de grafos são modelos bastante úteis na modelagem de sistemas com paralelismo e não determinismo, já que noções intuitivas como conflitos entre aplicações de regras e aplicação paralela pos-

suem definições precisas neste arcabouço formal (Seção 2.3). Os modelos semânticos para gramáticas de grafos variam essencialmente na forma com a qual tratam questões como concorrência e representação concreta de transformações de grafos (Seção 2.4). Dentre os possíves métodos para análise de gramáticas de grafos, a análise de par crítico permite que se obtenha uma previsão de todos os potenciais conflitos e dependências entre duas regras da especificação (Seção 2.8). A abordagem de pushout duplo, que é a variação mais antiga da abordagem algébrica, está atualmente generalizada através do arcabouço conhecido por Sistemas Adesivos de Substituição de Alto Nível *(Adhesive High-Level Replacement Systems)* (Definição 2.9). Portanto, resultados teóricos desta abordagem como a caracterização de conflitos e definição de paralelismo estão representados em um alto nível de abstração. Apesar disto, não há atualmente na área de transformação de grafos uma representação padrão de transformações de alta ordem, isto é, regras de mais alto nível que modificam regras de reescrita de grafos. Esta ausência dificulta a utilização de gramáticas de grafos como modelo-base a ser evoluído. O primeiro passo então para definir transformações programadas sobre tais regras é desenvolver a ideia de transformação de alta ordem para tais modelos. Para tal, buscamos inspiração em outros formalismos onde alta ordem é comum, em especial cálculo-lambda.

B.3 Alta ordem em cálculo lambda

No Capítulo 3, revisamos os principais conceitos de cálculo-lambda, formalismo proposto por Alonzo Church com o intuito de formalizar a definição e aplicação de funções. Apesar da sua simplicidade e concisão, cálculo-lambda é um formalismo muito expressivo, sendo equivalente a outros modelos universais como máquinas de Turing. Além da versão original, há variações do cálculo onde classificamos os termos em tipos, sendo que o mecanismo de tipagem garante a compatibilidade dos argumentos passados para uma aplicação de função. No contexto de cálculo lambda tipado, podemos caracterizar termos de alta ordem com base na altura do seu tipo (Definição 37). Em função disto, apresentamos uma analogia entre reescrita de grafos e avaliação de termos em cálculo lambda tipado, a fim de obter intuição sobre como definir termos de alta ordem no contexto de transformação de grafos (Seção 3.3).

B.4 Reescrita de grafos de segunda ordem

O Capítulo 4 aborda a questão de como definir regras de segunda ordem, isto é, regras que transformam regras de transformação de grafos, utilizando o mecanismo de pushout duplo. Para tal, devemos estabelecer uma categoria onde regras sejam objetos e morfismos sejam transformações de regras. Testamos inicialmente a categoria T-**Span**, formada por *spans* arbitrários de grafos e respectivos morfismos. Definimos regras de segunda ordem, a qual nomeamos 2-regras, como spans em T-**Span** onde os objetos são regras e ambos os morfismos são injetores (Definição 42). Por construção, T-**Span** é uma categoria adesiva, permitindo a aplicação direta do mecanismo de pushout duplo e a teoria associada (Proposição 41). Contudo, reescrita de regras pelo mecanismo de pushout duplo em T-**Span** permite que transformemos uma regra em um span onde um dos morphismos é não-injetor. Tal característica é indesejável, pois a injetividade das regras é fundamental para garantir o determinismo da reescrita dentro da teoria de categorias adesivas. Desta forma, consideramos a subcategoria de T-**Span** contendo somente spans que são regras, a qual chamamos T-**Rules** (Definição 51). Provamos uma situação de adjunção entre

T-**Span** e *T*-**Rules**, o que permite caracterizar corretamente limites e colimites de *T*-**Rules** a partir das mesmas construções em *T*-**Span**. Contudo, *T*-**Rules** falha em ser adesiva (Proposição 64). A conclusão é que não é possível obter com ambas as categorias uma caracterização direta de reescrita de regras que permita a aplicação do mecanismo de pushout duplo e, ao mesmo tempo, não gere regras inválidas. São consideradas duas possíveis formas de contornar este problema: a primeira consiste de *corrigir* a regra inválida após a reescrita utilizando a unidade da adjunção entre *T*-**Span** e *T*-**Rules** (Definição 65); a segunda consiste de *evitar* reescritas problemáticas através do mecanismo de *condições negativas de aplicação* para 2-regras (Definição 81). Como a primeira abordagem apresenta algumas características indesejáveis, definimos a segunda abordagem como a definição padrão de *reescrita de segunda ordem* (Seção 4.5). Outra vantagem desta escolha é que a teoria de categorias adesivas, na qual *T*-**Span** se enquadra, foi recentemente generalizada para englobar resultados envolvendo condições negativas de aplicação. Isto faz com que reescrita de segunda ordem se enquadre naturalmente no arcabouço formal atualmente utilizado na área de transformação de grafos. Finalmente, enumeramos as modificações possíveis de aplicar sobre regras usando reescrita de segunda ordem, e instanciamos a caracterização de conflito entre reescritas de segunda ordem.

B.5 Gramáticas de grafos de segunda ordem

O Capítulo 5 introduz gramáticas de grafos de segunda ordem, isto é, especificações contendo tanto regras quanto 2-regras. Antes de definir a especificação em si, tratamos do problema de como representar coleções de regras adequadamente de forma que possamos definir uma noção de reescrita de *coleções*. Verificamos os problemas de considerar conjuntos para representar coleções de regras, e introduzimos a ideia de utilizar para tal propósito coprodutos de regras com infinitas injeções da regra vazia (Definição 96). Esta escolha permite descrever uma forma de reescrita que pode modelar tanto modificação de regras quanto adição e remoção de regras na coleção (Definição 93). Utilizando coprodutos como coleções, redefinimos gramáticas de grafos convencionais (Definição 99), visto que estas são os *estados* da execução de gramáticas de grafos de segunda ordem. Posteriomente definimos gramáticas de grafos de segunda ordem simples, composta de uma gramática de primeira ordem inicial e uma coleção de 2-regras (Definição 103). A execução pode se dar tanto por uma reescrita interna da gramática-estado, quanto de uma reescrita de segunda ordem modificando a gramática-estado (Definição 104). De forma similar, definimos variações de gramáticas de segunda ordem descrevendo como modificar o grafo tipo (Definições 112), o grafo inicial (Definição 116) ou o número de regras da especificação (Definição 124). Após revisarmos os modelos apresentados, definimos as condições para uma derivação representar uma transformação de modelos (Definição 128), e apresentamos como obter o que foi preservado, adicionado e removido por uma transformação de modelo via spans sobre gramáticas de grafos (Definição 137).

B.6 Interação entre as camadas de primeira e segunda ordem

No Capítulo 6 tratamos da interação entre reescritas de primeira e segunda ordem durante a execução de gramáticas de grafos de segunda ordem. Iniciamos definindo as noções de independência paralela e conflito (Definição 140). A partir disto, estendemos estas noções para derivações de gramáticas de segunda ordem (Definição 142). Posteriormente, apresentamos uma extensão do método de cálculo de pares críticos para gramáticas

de segunda ordem (Seção 6.2). Finalmente, definimos como relacionar pares críticos do sistema original com pares críticos do sistema após uma derivação representando transformação de modelos (Seção 6.3).

B.7 Gramáticas de grafos orientadas a aspectos

No Capítulo 7, gramáticas de grafos de segunda ordem são utilizadas para descrever abstrações de orientação a aspectos sobre gramáticas de grafos, definindo o modelo conhecido como *gramática de grafos orientadas a aspectos* (Definição 155). Aspectos sobre uma gramática de grafos são representados através de uma camada de segunda ordem implementando transformações estruturais. Definimos como combinar diversos aspectos sobre a mesma gramatica em um único aspecto composto (Definição 158), o que permite representar o processo de combinação aspectual através da execução de uma única gramática de segunda ordem. Definimos a *gramática combinada* como o resultado de uma transformação de modelo sobre a gramática com aspecto composto (Definição 159). Finalmente, discutimos como os métodos de análise apresentados no Capítulo 7 podem ser utilizados para estudar a o efeito da combinação aspectual sobre a semântica da gramática base.

B.8 Trabalhos relacionados

O Capítulo 8 apresenta alguns trabalhos da literatura relacionados à nossa proposta. Inicialmente, falamos de resultados preliminares sobre a modificação de regras de transformação de grafos. Após, comentamos sobre redes de Petri com suporte a modificação dinâmica de regras. Depois apresentamos gramáticas de grafos triplas, modelos cuja estrutura de regra é similar a 2-regras, porém possuem um propósito diferente destas. Finalmente, mencionamos alguns trabalhos que associam transformações de grafos a construções de orientação a aspectos.

B.9 Conclusões

No Capítulo 9, conclui-se a apresentação desta tese através da revisão das principais contribuições e apontamento de trabalhos futuros. Em linhas gerais, discutiu-se a interação entre transformações de modelo e a execução destes no contexto de gramáticas de grafos. Para tal, tomamos como base teórica a teoria de transformação de grafos através do mecanismo de pushout duplo, e a aplicamos tanto para implementar transformações de modelo quanto para representar a execução do sistema base. Foi necessário definir uma teoria de transformações de segunda ordem para descrever *regras que modificam regras* (ou 2-regras). Para garantir a preservação da estrutura de regra após a reescrita, utilizou-se o mecanismo de condições negativas de aplicação. Após, propusemos diversos tipos de gramáticas de grafos de segunda ordem, compostas de uma especificação inicial (grafo inicial mais regras de reescrita) e uma camada de segunda ordem (2-regras, regras para atualização do grafo inicial, retipagem), promovendo modificações sobre o modelo original. A noção de coleção baseada em coproduto foi introduzida com o objetivo de servir de estrutura para a definição de reescrita sobre coleções de regras. Finalmente, propusemos uma noção de conflito entre derivações de primeira e segunda ordem em gramáticas de grafos de segunda ordem, e discutimos como esta noção pode ser usada no estudo da interação entre a evolução do sistema e a sua execução. A hipótese na qual esta tese se baseou é

que princípios de alta ordem no contexto de gramáticas de grafos podem ser úteis na modelagem e análise de sistemas sofrendo modificações programadas. As seguintes conclusões puderam ser obtidas a partir do processo de desenvolvimento desta tese:

- *Modelagem:* Os modelos propostos de gramáticas de grafos de segunda ordem são bastante flexíveis, permitindo que representemos tanto modificações no grafo tipo, no grafo inicial e na coleção de regras de primeira ordem (via alteração, inclusão e remoção de regras). Ao longo do texto apresentamos diversos exemplos de transformações passíveis de serem implementadas nestes modelos, incluindo adição de registro de log, domínios para envio e recepção de mensagem e até mesmo correções do sistema original, todas representadas através de uma camada de segunda ordem. Além disto, foi natural introduzir os conceitos de *derivação representando transformação de modelos* e *span de evolução* para capturar as transformações induzidas sobre o modelo original através da execução da camada de segunda ordem sobre ele. A forma direta pela qual *aspectos* sobre gramáticas de grafos foram implementados utilizando camadas de segunda ordem são uma evidência da conveniência de termos construções de alta ordem em especificações.

- *Análise:* O fato de termos utilizado o mesmo princípio (pushout duplo) para ambos os níveis de reescrita permitiu que obtivéssemos uma noção de *conflito inter-nível* de forma natural, representando as possíveis interferências de reescritas de segunda ordem sobre as reescritas do sistema de primeira ordem. Com base nesta noção de conflito, foram sugeridos duas técnicas para estudar o comportamento de tais especificações: uma sendo a extensão natural de análise de par crítico para gramáticas de grafos com segunda ordem, e outra que, dada uma derivação representando uma transformação de modelos, identifica como pares críticos no modelo original se relacionam a pares críticos no modelo evoluído. Discutimos também como as informações obtidas por ambas técnicas podem ser úteis no estudo de gramáticas de grafos com aspectos.

Baseado nesta argumentação, defendemos que sim, conceitos de alta ordem são úteis na modelagem e análise de sistemas sob transformações programadas. Adicionalmente, ao criar uma noção de transformação de segunda ordem para a abordagem baseada em pushout duplo, iniciamos uma investigação teórica de *situações inter-nível* em gramáticas de grafos de alta ordem. Por exemplo, a partir deste ponto podemos investigar a construção de modelos de terceira ou quarta ordem, ou então se a ideia de construção de segunda ordem que introduzimos pode ser generalizada para categorias adesivas em geral.

Apesar de construções de alta ordem surgirem naturalmente e diversos campos da Ciência da Computação, não há atualmente uma teoria geralizada de transformação de alta ordem disponível para a abordagem de pushout duplo. Como contraste, podemos citar a literatura de redes de Petri, onde a representação do modelo como grafo é mais direta e há diversos modelos que suportam transformações de alta ordem. Apesar de estar presente em noções como meta-modelagem e esquemas de regras, a noção de evolução para gramáticas de grafos não é tão madura como, por exemplo, a formalização de conflitos entre reescritas e a definição de modelos semânticos. Acreditamos que esta tese, através da definição das noções de reescrita de segunda ordem, introdução do modelo de gramáticas de grafos de segunda ordem, e da noção de conflito inter-nível, fornece um passo inicial importante para compreender alta ordem no contexto de reescrita de grafos. Além disso, diversos aspectos como a questão da utilização de coprodutos para representar de coleções de regras, também

são considerados ao longo do texto. A aplicação prática dos conceitos introduzidos foi exemplificada através da modelagem de conceitos de programação orientada a aspectos sobre gramáticas de grafos, e da discussão sobre como utilizar algumas técnicas de análise para obter informação sobre a evolução do sistema combinado.

Naturalmente, por questão de escopo, algumas questões originadas ao longo do processo de pesquisa não puderam ser exploradas em profundidade nesta tese. Como trabalhos futuros, citamos: extensões ao modelos propostos, tais como considerar que regras de primeira ordem podem conter condições negativas de aplicação; investigações formais, como a generalização do mecanismo de reescrita de segunda ordem para englobar reescritas de ordem arbitrária; ou a implementação do mecanismo de alta ordem em ferramentas de transformação de grafos, o que permitiria a realização de experimentos de modelagem e análise sobre cenários de interesse.