University of Twente

Faculty of Electrical Engineering, Mathematics and Computer Science

P.O. Box 217, 7500 AE Enschede
The Netherlands

Nested Quantification in Graph Transformation Rules

Master's Thesis for

Computer Science

by

J.H. Kuperus

submitted on June 25th, 2007

Exam Committee

dr. ir. A. Rensink | ir. H. Kastenberg | dr. ir. J. Kuper

"Life is really simple, but we insist on making it complicated"
–Confucius

Preface

The document before you contains the resulting documentation of my Master's Thesis. In the period from September 1st, 2006 to June 28th, 2007 I worked on this project to finalize my study in Computer Science.

I would like to thank the members of my graduation committee for their support during this project. A few words for each, in alphabetic order:

- Harmen Kastenberg, for pointing me in the right direction when I got stuck during implementation
- Jan Kuper, for warning me about Category Theory and helping me understand it and wield its power
- Arend Rensink, for supplying the project idea and asking the questions I tried to avoid

This version of the document was built on Monday, June 25, 2007 at 10:09, and has version number 10.

Abstract

By tradition, researchers working on model checking tools continuously try to make their tools faster and allow them to handle larger models. GROOVE is a model checking tool which uses the mathematical formalism of graphs and graph transformations to specify models and system behavior. Graph transformation systems allow complex models to be visualized and are a natural way of modeling object oriented systems.

Graph transformation systems only allow their rules to be matched existentially, which poses a serious limitation. Complex constructions with so-called helper edges are often created to perform a task more than once. This causes the model to be more complicated than it should be, for both the user and the tool.

This thesis defines an extension of the use of the single pushout approach which allows nesting of alternated quantifiers to an arbitrary depth. This means entire subgraphs may now be matched without first knowing exactly how many nodes it will contain. The formalism has also been implemented in the GROOVE graph transformation tool and shows drastic decreases of required statespace and computing time for several well-known models.

Contents

List of Figures

Notations

(The number after the item indicates the page where the notation is defined.)

Symbols

$G, H, \ldots$	Labeled directed graphs in the category Graph, 5
$\mathcal{L}$	Set of labels, 5
$f, g, f_0, f_1, \ldots$	Graph morphisms in the category Graph, 6
$m, m_0, m_1, \ldots$	Matchings of a rule into a host graph, 10, 12
$p, q, p_0, p_1, \ldots$	Graph predicates, 18
$r, s, r_0, r_1, \ldots$	Transformation rules, 10, 12
$t, t_0, t_1, \ldots$	Rule trees, 22
$T_p^{\exists}$	Set of existential subtrees of the predicate p, 20
L_i	Left-hand side graph of a rule r_i, 10, 12
R_i	Right-hand side graph of a rule r_i, 10, 12
R_i^n	Right hand side intermediate graph, obtained from sub-rule r_i with matching m_n, 29
E_n	Intermediate graph result, obtained from the rule instance with matching m_n, 26
ρ	A morphism in the category Rule, 22

Definitions

DPO	Double Pushout rewriting, 11
NAC	Negative Application Condition, 10
SPO	Single Pushout rewriting, 10
Graph	Category of simple graphs and partial morphisms, 10
Rule	Category of rules and rule morphisms, 17
Instance	Category or rule instances and rule morphisms, 24
$root_d$	Initial object of the diagram d, 9
$sub_d(K)$	Reachable sub-diagram starting at object K in the diagram d, 9
$out_d(K)$	Arrows in the diagram d originating in the object K, 9
$init_d$	Arrows in the diagram d originating in the object $root_d$, i.e. $init_d = out_d(root_d)$, 9
$exsub_d(K)$	Existential subtree of the diagram d starting at the object K, 20

1 Introduction

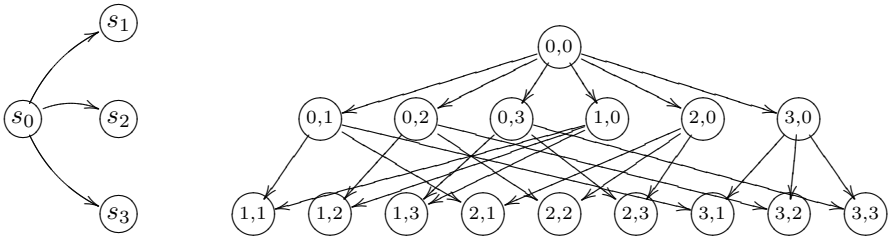
1.1 Context

"Errare humanum est"[1]. Seneca the Younger already stated in the first century after Christ that humans make mistakes. Sadly, people still make mistakes today, computer programmers are no exception to this rule. In an ongoing effort to reduce the number of mistakes that will end up in any released software product, engineers employ tools to attempt to detect errors as early as possible.

One of these methods is verification and in this thesis, the focus lies on verification by model checking. In model checking, the engineer makes a model of the application, either concrete or on a certain level of abstraction. These models rely on two important notions: the state a program is in and what kind of actions may be taken in any particular state. In a formal specification of an application, a designer usually records several properties the system must have. Such properties may include invariants, liveness properties, timing constraints, etc.

Given a model and one or more properties, a model checker can be used to evaluate the model and check whether the chosen properties hold in all those states. In case a property does not hold in the model, a model checker generally identifies how the error was produced. This allows the designer to see the flaws in the model and adjust them even before implementation has begun.

When a model checker is exploring all the possible states or executions of a model, it keeps track of the states it has already visited. Eventually, when all states have been visited, the entire statespace of the model will have been computed. A common problem among model checkers is the fact that such a space tends to become very large. For example, a single process such as shown in figure 1.1a has a statespace of exactly four states. However, when two of these processes are run in parallel, the combined statespace contains contains sixteen $(4 * 4)$ states. Figure 1.1b shows the statespace of the combined processes.



(a) Simple process (b) Statespace of two combined processes

Figure 1.1: Sample process and statespace

[1] To err is human

Adding another instance of the same process to the model causes the space to grow to $64 = 4^3$ states. This phenomenon is called *statespace explosion* and is a major problem among model checkers. A lot of research focuses on reducing either the statespace, or the number of states that require checking. Examples of such techniques are *partial order reduction*, *abstraction*, *bit-state hashing*, etc.

In this thesis, a method is developed to help reduce the statespace by means of grouping many small steps into one single step. This is not done generally for any type of model checker, but specifically for the GROOVE(Rensink, Kastenberg, & Staijen, 2004) model checker. GROOVE (GRaphs for Object Oriented VErification) is being developed by the Formal Methods and Tools chair at the University of Twente.

The GROOVE project focuses on the use of model checking techniques for object-oriented systems (Kastenberg & Rensink, 2006). Since object-oriented systems consist more of reference structures than a collection of primitive values, graphs are used as the means to represent states. Graphs are a more natural way to model these dynamic structures than bit vectors used by classical model checkers. This approach is expected to open up new opportunities to specify and verify highly dynamic structures.

In order to specify behavior of a model, GROOVE uses graph transformation rules. These rules form a so called *graph grammar*, which can be executed by the GROOVE Simulator to form a *graph transition system (GTS)*. This GTS can then be used to check the model against CTL formulas.

The project aims at creating a solid tool to promote model checking within the industry, but also allows for excellent use in education due to the inherent visual orientation of the model. Graphs show what happens and where it happens.

1.2 Goal

Reducing the statespace of models without losing the ability to decide whether or not a property holds is a continuous effort of the model checking community. Currently, GROOVE only supports one existential level of quantification. This means that a single rule will be applied to exactly one instance of it in the model. There is currently no way to express that all such instances should be transformed at once.

The goal for this research is to provide theory and an implementation to support such universally quantified rules. The central question on which the research focuses is *"How can universal quantification help to reduce the statespace?"*. The answer shall be provided by the answers to the following questions:

1. *What methods of quantification are being considered in the field?*
2. *What would be a useful notation for nested predicates / production rules?*
3. *How can universal quantification be defined in a formal way?*
4. *How should GROOVE be extended to support nested Application Conditions / predicates?*
5. *What is the impact on the statespace and computing time?*

Each question will be analyzed and answered in turn. The next section shows the general outline of this thesis and gives an indication of which question is answered in which chapter. For readers who are only interested in the answers, there is a brief discussion in chapter 5 where the answer to each question is summarized.

1.3 Thesis Outline

Chapter 2 of this document is the result of a literature study on Graph Transformations. It describes in short what they are and introduces a few theoretic definitions as a basis for chapter 3. This part is particularly useful for people unfamiliar with this concept of graph transformations. This chapter provides a partial answer to the first research question.

Chapter 3 introduces the central problem for this Master's Thesis, the nested quantification. This part defines the operation in terms of existing graph theory and showcases several examples to illustrate the theoretical process. The third research question is answered in this chapter. It will interest people who are working with graph transformations or who want to know exactly what it is the author did during his Master's Thesis project. The chapter requires quite some background knowledge on category theory, of which chapter 2 only gives a small introduction.

Chapter 4 focuses on the fourth and fifth research questions, but also briefly discusses several other graph transformation tools in the field, answering question one. It describes some of the key choices in the implementation of the universal quantification in GROOVE. Secondly, this chapter takes a few well known examples and shows how the use of universal quantification reduces the statespace and computing time.

Last, but not least, chapter 5 provides a discussion on the work performed in this thesis, recapping on the research questions and identifying future work.

2 Graphs and Transformations

This chapter provides a walkthrough of the required theory for this thesis. It lays out the basic definitions on which the research relies and discusses related work in the field. The material presented in this chapter is quite technical and mostly theoretical. However, some examples are included to make it more accessible to people unfamiliar with this kind of graph theory.

2.1 Basics and Definitions

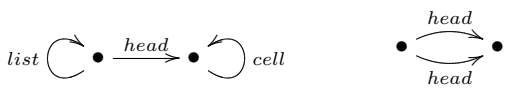
Graphs are used to represent and reason about a wide variety of problems, from traffic models to social networks. In order to be able to work on any form of graph, a formal definition of what a graph is is required. Although there are many ways to define a graph and there are many types of graphs, the focus in this thesis will be on *labeled directed graphs*, since GROOVE operates on these graphs.

The exact specification of the representation of a label is irrelevant, it is simply required that a set of labels, or alphabet, $\mathcal{L}$ exists. The examples in this thesis will use labels consisting of numbers and words, e.g. *list*, *append*, *cell*, *next*, $1, 2, 3, 4$. With such a set of labels, definition 2.1 specifies what a graph is.

Definition 2.1 (simple graphs)
A labeled graph G with labels from the alphabet $\mathcal{L}$ is a tuple (N_G, E_G), where N_G is the set of nodes and E_G is the set of edges, defined as $E_G \subseteq (N_G \times \mathcal{L} \times N_G)$. One tuple in E_G consists of a source node, a label and a target node, respectively.

In addition three functions are given: $src_G : E_G \to N_G$ maps an edge to its source node, $tgt_G : E_G \to N_G$ maps an edge to its target node and $lbl : E_G \to \mathcal{L}$ maps an edge to its label.

The graphs that adhere to this defintion are often called *simple graphs*. A graph is called simple if its edges have exactly one source and one target node. Other types of graphs loosen this restriction, like *hypergraphs* or *multigraphs*. GROOVE currently only supports simple graphs and this thesis is also restricted to the use of simple graphs. If there is no confusion to which graph the sets or functions from definition 2.1 belong, the subscript G will be omitted.



(a) Correct (b) Incorrect

Figure 2.1: Two different graphs

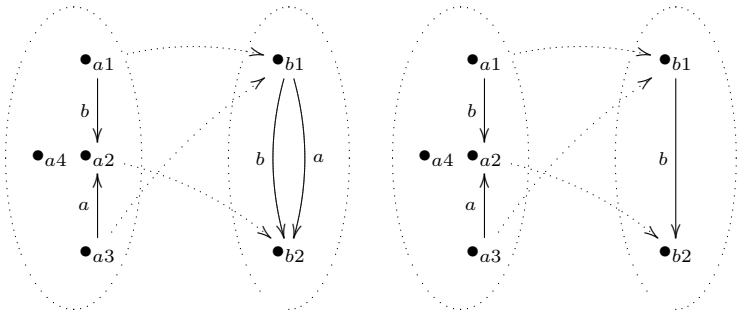
Figure 2.1 shows two sample graphs, one is correct with respect to definition 2.1 (figure 2.1a). The other is not (figure 2.1b), because it has two edges with the same source, destination and label.

Graphs can be related to each other, one may be embedded in another, or two graphs may be structurally equal. Such relations are encoded in *graph morphisms*. These morphisms form the foundation of graph transformations and are heavily used in this thesis.

Definition 2.2 (graph morphism)
A graph morphism $f : G \to H$ is a pair of partial functions $f = (f_N, f_E)$ with $f_N : N_G \to N_H$ and $f_E : E_G \to E_H$, such that $f_N \circ src_G = src_H \circ f_E$ and $f_N \circ tgt_G = tgt_H \circ f_E$ holds for all mapped edges, i.e. if an edge is mapped, its source and target nodes must also be mapped. Labels on mapped edges are explicitly preserved, so $lbl_G = lbl_H \circ f_E$.

Definition 2.2 is a very generic way to define a morphism. It defines an important property for use with graph transformations: morphisms *preserve labels*. In the figures in this document, a morphism will be depicted by dotted arrows between nodes of different graphs. There will not be any arrows between edges, because they are implicitly defined by the mapping of their source and target nodes. If the mapping of an edge is not implicit, this will be noted in the describing text. An example of a non-implicit mapping for an edge is when it has to be removed and a new one with the same label is created.

If a figure depicts more than one graph, each one is encircled by a dotted ellipse. Figure 2.2 shows two different morphisms, figure 2.2a shows a correct graph morphism $f : G \to H$, whereas figure 2.2b shows an incorrect morphism $g : G \to H$ with respect to definition 2.2. Below is a brief examination of the two morphisms f, g which reveals why they are correct or incorrect.



(a) Correct morphism (f) (b) Incorrect morphism (g)

Figure 2.2: Graph morphisms

The graphs G and H are defined as follows:

$G = (N_G, E_G)$
$N_G = \{a1, a2, a3, a4\}$
$E_G = \{e1 = (a1, b, a2), e2 = (a3, a, a2)\}$

$H = (N_H, E_H)$
$N_H = \{b1, b2\}$
$E_H = \{e10 = (b1, a, b2), e20 = (b1, b, b2)\}$

According to the definition, the morphism $f : G \to H$, shown in figure 2.2a, defines the mappings f_N and f_E. The functions f_N and f_E map the elements of G in the following way:

$$f_N(a1) = b1 \qquad f_N(a2) = b2 \qquad f_N(a3) = b1$$

$$f_E(e1) = e20 = (b1, b, b2) \qquad f_E(e2) = e10 = (b1, a, b2)$$

It should now be obvious that the second morphism (g) is not label preserving and therefore not a morphism with respect to the definition. The label of edge $e2 = (a3, a, a2)$ is not preserved, it should map to an edge $g_E(e2) = (g_N(a3), a, g_N(a2)) = (b1, a, b2) \neq (b1, b, b2)$. However, since the morphisms are partial, it is possible not to map $e2$ onto anything in H, which would make the morphism g valid again.

The fact that node $a4$ is not mapped by the f_N function does not invalidate the morphism. Neither does the fact that two nodes are projected onto the same node. However, in some cases it is desirable to have such restrictions on a morphism. This leads to the following definitions. Since a graph morphism consists of two functions, f_N and f_G, definition 2.3 will first define the different types of functions.

Definition 2.3 (function types)

i) A function $f : G \to H$ is said to be injective *if $\forall a, b \in G$, $f(a) = f(b)$ implies $a = b$ (or $a \neq b$ implies $f(a) \neq f(b)$)*

ii) A function $f : G \to H$ is said to be surjective *if $\forall y \in H$, there exists an $x \in G$ such that $f(x) = y$*

iii) A function $f : G \to H$ is said to be bijective *if it is both injective and surjective, i.e. $\forall y \in H$ there exists exactly one $x \in G$ such that $f(x) = y$*

iv) A partial *function $f : G \to H$ assigns values only to some subset $G_0 \subseteq G$*

v) A total *function $f : G \to H$ is a partial function that assigns values to all elements in G*

The same properties hold for morphisms, defined as follows:

Definition 2.4 (morphism types)

i) A morphism $f : (f_N, f_E)$ is injective *if both f_N and f_E are injective functions*

ii) A morphism $f : (f_N, f_E)$ is surjective *if both f_N and f_E are surjective functions*

iii) A morphism $f : (f_N, f_E)$ is bijective *if both f_N and f_E are bijective functions*

iv) A morphism $f : (f_N, f_E)$ is partial *if either f_N or f_E is a partial function*

v) A morphism $f : (f_N, f_E)$ is total *if both f_N and f_E are total functions*

With these new definitions in mind, the example graph morphisms f, g from figure 2.2 can be called *partial surjective morphisms*. In section 2.2 these different types of morphisms will be used in the definition of graph transformation rules.

2.1.1 Category Theory

Before moving on to graph transformations, it is imperative to include some basic Category Theory, since the approaches are defined in terms of category operations. The definition of a category looks fairly simple, but this is due to a high level of abstraction which makes it a very powerful construction which can create extremely complicated scenarios. These definitions are derived from those in Barr and Wells (1990).

Definition 2.5 (category)

A category $\mathbf{C}$ is a tuple $(obj_{\mathbf{C}}, arr_{\mathbf{C}})$, where $obj_{\mathbf{C}}$ is a set of objects and $arr_{\mathbf{C}}$ is a set of arrows. Furthermore, the following properties must hold:

i) Each arrow or morphism $f \in arr_{\mathbf{C}}$ has a domain *and* codomain *in $obj_{\mathbf{C}}$. If the domain of f is A and the codomain is B, we write $f : A \to B$*

ii) For each pair of morphisms $f : A \to B$ *and* $g : B \to C$*, there is a* composite *morphism* $g \circ f : A \to C$

iii) For every object $A \in obj_{\mathbf{C}}$ *there is an* identity *morphism* $1_A : A \to A$

iv) For every morphism $f : A \to B$ *the following should hold:* $1_B \circ f = f$ *and* $f \circ 1_A = f$ *(identity property)*

v) For morphisms $f : A \to B$*,* $g : B \to C$ *and* $h : C \to D$ *the following should hold:* $(h \circ (g \circ f)) = ((h \circ g) \circ f)$ *(associativity)*

As with graphs, if it does not lead to confusion, the subscript of $obj_{\mathbf{C}}$ and $arr_{\mathbf{C}}$ will be omitted. A few examples will follow to give the reader a feeling for categories.

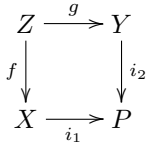
Example 2.6
The category $\mathbf{Pre_n}$ *contains the integer numbers* $0..n(\subseteq \mathbb{N})$ *as objects and the relation* $\leq$ *provides the arrows. Figure 2.3 shows the category* $\mathbf{Pre_n}$*. The looping arrows are the identities, e.g.* $1 \leq 1$*. Composition of arrows is the transitivity of the* $\leq$ *relation:* $a \leq b \wedge b \leq c \Rightarrow a \leq c$

Figure 2.3: A representation of the category $\mathbf{Pre_5}$

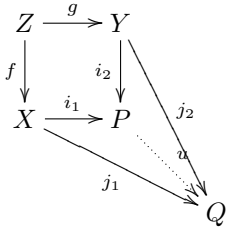
An attentive reader will notice that the above figure is not a complete representation of the category $\mathbf{Pre_5}$. The composite arrows have been omitted. Composite arrows often clutter figures and since the definition of a category requires them to exist they are usually omitted in figures and diagrams. The same goes for identity arrows, which figure 2.3 does show, but the diagrams in the rest of this thesis will also not show identity arrows.

Category theory also allows for operations on objects within a category. This thesis relies heavily on one operation, being the pushout. A pushout is an operation on two arrows (or morphisms) which share the same domain, say $f : Z \to X$ and $g : Z \to Y$. The pushout of these morphisms consists of an object P and two morphisms $i_1 : X \to P$ and $i_2 : Y \to P$ such that the diagram in figure 2.4a commutes.

Definition 2.7 (Pushout)
Let $f : Z \to X$ *and* $f : Z \to Y$ *be morphisms, the pushout of* f *and* g *is an object* P *and morphisms* $i_1 : X \to P$ *and* $i_2 : Y \to P$ *such that the diagram in figure 2.4a commutes. Additionally, if there is an object* Q *and morphisms* $j_1 : X \to Q$ *and* $j_2 : Y \to Q$*, due to the universal property of pushouts there must be a unique morphism* $u : P \to Q$ *so that figure 2.4b commutes.*



(a) Pushout diagram



(b) Universal property

The universal property of pushouts is a property that is very important in the next chapter of this thesis. It is a property that makes it posible to prove the propositions later in this thesis.

As an example, consider the category of sets, with X and Y sets. If we write the object Z as the intersection of these sets, with inclusion morphisms f and g, then the pushout of f and g is the union of X and Y and the morphisms i_1 and i_2 are the inclusion morphisms from X and Y into P respectively.

This thesis uses several categories in its definitions, some of which will be introduced later on in chapter 3. One thing all these categories have in common is that they are based on a common category. The category of simple graphs and partial morphisms is the category in which the graph transformations operate and on which all theory in chapter 3 is based.

Definition 2.8 (Graph **category)**
The category Graph *has simple graphs as objects and partial graph morphisms as arrows.*

The category Graph has all pushouts as proven in Ehrig et al. (1997), which enables the use of pushouts for graph transformations. The specifics of this approach in graph transformations will be explained in section 2.2, but a small preview may make it easier to understand.

A pushout on a graph requires at least two morphisms, with a shared domain. Take a look back at figure 2.4a and say Z, X, Y and P are graphs. In graph transformations, the graph X is usually the graph on which a transformation is executed, Z and Y are part of the transformation rule and P is the result. The morpshism $g : Z \to Y$ specifies what has to be changed in the graph, i.e. what must be deleted is not mapped by the morphism and the new elements only exist in Y. The morphism $f : Z \to X$ is usually called the *matching* of the rule into the host graph.

Chapter 3 relies heavily on the use of diagrams, since they are the primary formalism for reasoning with categorical structures. Diagrams show objects and arrows and are required to commute. Meaning that if two there are two paths from one object to another, the two paths have to yield the same result. Figures 2.4a and 2.4b are examples of diagrams.

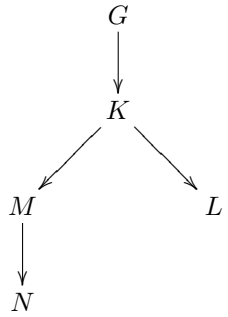


Figure 2.4: Sample tree-shaped diagram

A specific type of diagram is a tree-shaped diagram (see figure 2.4. Such a diagram displays a tree of objects, meaning that there are no cycles and from the root to each element in the tree, there is exactly one path. In this thesis, the arrows originating from an object X in a diagram d will be called $out_d(X)$. The following definition will introduce a few notations used in conjunction with tree-shaped diagrams.

Definition 2.9 (properties of tree-shaped diagrams)
A tree-shaped diagram *d in a category* $\mathbf{C}$ *is denoted as a tuple* $(G, out_d(G))$ *and has the following properties:*

- *the root a of tree-shaped diagram d is denoted* $root_d(= G)$

- *for each arrow* $f : G \to K \in out_d(G)$, $sub_d(K)$ *is the sub-diagram starting at root* K *and is also a tree-shaped diagram*

- *the set of arrows originating from the root of a diagram is* $init_d (= out_d(root_d))$

2.2 Graph Transformations

The use of graph transformations is one of many applications of the generic field of rewriting logic. Rewriting logic acknowledges the relation between logical formulae and computations. This chapter will introduce several approaches to rewriting graphs, but for a more complete overview of different rewrite logic formalisms, the reader is recommended to consider Mesegue (1998). In graph transformations, the rewriting occurs on the structure of a graph. A part of the graph is essentially replaced by another graph.

Algebraic graph rewriting relies on the notion of rules and categorical structures to represent the operation. Rules are built up from morphisms, operating from a left hand side, which is to be matched to the host graph, or a part of the host graph. Once such a match is found, the rule morphism specifies how the matched part in the graph will change.

There are several approaches to algebraic graph rewriting. Each of them specifies a rule and its application differently. The Single (SPO) and Double Pushout (DPO) methods have been used the most and they have extensive theory on their different uses and different tools have been developed using these methods. A good overview of these two approaches and their uses can be found in Ehrig et al. (1997); Corradini et al. (1997). This thesis gives a small introduction to these two approaches.

A more recent approach is the so-called Sesqui approach, which combines the advantages of the SPO and DPO approaches. Although the Sesqui approach is very interesting, a complete explanation lies beyond the scope of this thesis, since it requires a lot of extra base theory to be explained that is of no further use in this thesis. For more information on the Sesqui approach the reader is directed towards Corradini, Gadducci, and Montanari (1995); Stell (1994); Corradini, Heindel, Hermann, and König (2006).

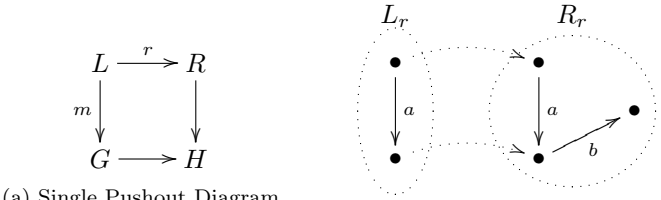
There has also been some research on account of a pullback approach, but it seems this method has lost interest since there are hardly any recent publications on the topic.

2.2.1 Single Pushout approach

The Single Pushout (SPO) approach is at first sight a fairly intuitive method. The SPO approach operates on the category of simple graphs and partial morphisms Graph as defined in definition 2.8. Rules for the SPO approach look for one particular subgraph in the hostgraph, which must match the left side L, and replace this part with the graph R on the right side as specified by the morphism. A single partial morphism $r : L \to R$ (sometimes written as $L \xrightarrow{r} R$) between the left side graph L and the right side graph R specifies exactly what happens in a graph transformation. Applying such a rule means that a *matching*, or *embedding* of L in the host graph G has to be found. This matching must be a total morphism $m : L \to G$, i.e. all elements of the graph L have to be mapped onto an element in G, but it does not have to be an injective morphism, i.e. two elements from L may be mapped onto the same element in G.

Definition 2.10 (SPO rule)
An SPO rule r is a morphism $r : L \to R$ in the category Graph*. The left and right side graphs are written as L_r and R_r. A matching of a rule r is a total morphism $m : L \to G$. Applying an SPO rule is done by computing the pushout of r and m to yield a target graph H (see figure 2.5a)*

Once a matching is found, the application leaves the part in G that is not matched untouched with the exception of *dangling edges* (see below). The matched elements are either preserved or deleted, depending on whether they have an image in the rule morphism or not. Finally, elements that exist in R_r, but have no preimage in the morphism are newly created in the target graph H. The application of the rule then boils down to $H = (G \backslash L_r) \cup R_r$, which essentially says remove the matched part of L_r, replace it with a copy of the right side R_r. Figure 2.5a displays a diagram in Graph in which an application is depicted, an application is valid if this diagram commutes. It also shows how the SPO approach got its name: the application is exactly a single pushout operation.



(a) Single Pushout Diagram

(b) Sample Rule

Figure 2.5: Single Pushout diagram and a sample rule

The SPO has a loose application condition. If a rule application would delete nodes in such a manner that one or more edges no longer have a source or target node, so called *dangling edges*, these edges will be deleted as well to obtain a new valid graph according to defintion 2.1. If two nodes from the left hand side are mapped onto the same node in the host graph and one is deleted in the rule and the other is not, deletion wins. Deletion of nodes always has priority in the SPO approach. This priority of deletion is where some of the less intuitive situations may occur, i.e. one might create a rule that searches for three nodes, places an edge between two of them and deletes the third. It would then seem logical to expect that there would be an extra edge in the target graph, but it is possible to find that only a deletion occurred. This will happen if the node that should be deleted is matched to the same node that gets an extra edge, since deletion takes priority.

Figure 2.5b displays a sample SPO rule. This particular rule will look for two nodes with an edge between them which has the label a, this is shown by the graph L. If such a pair of nodes is found, the rule's morphism specifies that these elements should be preserved in the graph and that a new node and a new edge should be created. This is because node at the end of the b-edge and the b-edge itself do not have a preimage in the morphism, but only exist in the right side R. So the node in the target graph that was matched to the node at the end of the a-edge receives a new edge with a new node at the end.

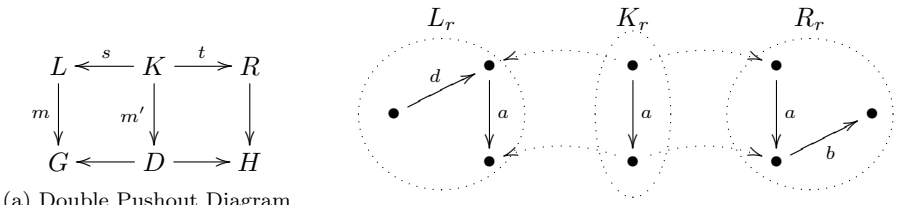
2.2.2 Double Pushout approach

The Double Pushout approach (DPO) is also based on the categorical pushout construction. As the name suggest, this approach uses two pushouts instead of one. A rule for this approach is built from three graphs, related through two *total* morphisms. The need for two pushouts comes from the fact that all morphisms have to be total.

This means deletion can not be specified as with the SPO rule, by simply not mapping an element from L_r to R_r, because then the morphism would be partial. To deal with this, the DPO approach uses an interface graph K_r which is a common subgraph of both L_r and R_r, which contains elements that remain unchanged during application. The rule itself now consists of two total morphisms $s : K_r \to L_r$ and $t : K_r \to R_r$, where deletion is specified by elements that exist in L_r, but do not exist in K_r. New elements are similarly introduced, because they exist in R_r and not in K_r. In short, the application of a DPO rule can be written as $H = (G \backslash (L_r \backslash K_r)) \cup (R_r \backslash K_r)$, i.e. the matched part of L_r that is not also part of K_r is deleted, and the part of R_r that is not part of K_r is added to the graph. A complete DPO rule is usually written as $r = (L \overset{s}{\leftarrow} K \overset{t}{\rightarrow} R)$.

Figure 2.6a shows the diagram that has to commute when applying the DPO approach. The D object shown here is called a pushout complement object, which is actually the intermediate result, constructed from the first pushout. For a DPO rule to work in a certain category, this pushout complement object must be uniquely defined within the category. The requirement of the uniqueness of the pushout complement object is the only requirement the DPO approach imposes on a category.

Definition 2.11 (DPO rule)
A DPO rule r consists of two total morphisms $s : K \to L$ and $t : K \to R$. The left and right side graphs are again written L_r and R_r. Additionally, the interface graph is written as K_r. A matching of a rule r is a total morphism $m : L_r \to G$. Applying a DPO rule is done by first computing the pushout complement object D and then computing the pushout of t and m' to yield the target graph H (see figure 2.6a)



(a) Double Pushout Diagram

(b) Sample Rule

Figure 2.6: Double Pushout diagram and a sample rule

Just like the application of an SPO rule, a matching has to be found in the host graph G. The rest of the rule application now comes down to $H = (G \backslash (L_r \backslash K_r)) \cup R_r \backslash K_r)$. The computation of a matching for the DPO approach is the same as for the SPO approach, both take any application conditions into account and try to find a matching of L_r in G.

As opposed to the SPO approach, the DPO approach has a strict application condition. The so called *gluing condition* consists of a *dangling condition* and an *identification condition*. The dangling condition says that no dangling edges may be left by the rule, as opposed to the SPO approach, which simply deletes dangling edges. The identification condition states that any element that is to be deleted may not be identified with another element that remains constant, i.e. an element that exists only in L_r may not be mapped to an element that also exists in K_r. Situations such as unexpected deletion can not occur due to this restriction.

Figure 2.6b displays a sample DPO rule. This example will search for three nodes, with edges between them as depicted in the L_r graph. When this rule is applied to a hostgraph, the node at the start of the d-edge and the d-edge are deleted, because they exist in L_r, but not in K_r. The nodes connected to the a-edge remain unaltered and finally, a new node is added and connected to the graph by the new b-edge.

2.2.3 Application Conditions

Both the SPO and DPO method support the use of *application conditions*. An application condition K is a graph of which the left side of the rule is a subgraph, i.e. $L \subset K$. When a matching for L in G is being computed, such a matching must not be expandable to one of the application conditions. Intuitively, application conditions specify what is not to be matched, i.e. what kind of structures the matched part may not contain. For example, a rule may state *"Find a seat"* and an application condition may say *"The seat may not be occupied"*, if a matching of this rule is computed, every seat will be examined and if it is not occupied, it is a valid matching, if it is occupied, the matching is not valid.

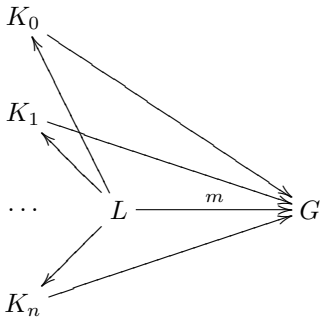


Figure 2.7: Application conditions

Formally, a rule may have any number of application conditions K_i with $L \subset K_i$ for each i. A matching $m : L \to G$ is valid if there are no *total* morphisms from any K_i to G such that diagram 2.7 commutes. Due to the negative nature of these conditions, they are generally referred to as negative application conditions (NAC).

3 Nesting Transformation Rules

3.1 Research Motivation

Behind any research project lies a desire, it is wat sets a researcher in motion. Within a tool-building project it should either fix problems that keep users from effectively using the tool, or they should improve the functionality and usability of it. The work done within this thesis project does a little of both.

Allowing nested quantification within graph transformation rules allows the user to specify actions for groups of elements. It offers the possibility of specifying that several steps should be done at once. Not only does this strengthen the specification language of GROOVE, it makes it possible to solve certain problems in a more intuitive way. This is because it makes more sense to say *"Throw away all newspapers in the bin"*, than to say something like *"Put up a sign to indicate that I am throwing my newspapers away. As long as there is a newspaper in the bin, throw it away. If there are no more newspapers in the bin, remove the sign"*. The sign used in this example is to indicate to other rules that a series of actions of a certain rule is taking place. In order to keep other rules from interfering, each rule that may interfere with this rule needs to make sure the *sign* is not there. If there are multiple rules that use such signs to indicate work-in-progress, just imagine how many other rules may need to look out for all of these signs. It makes a set of rules unnecessarily complicated.

Nesting levels of quantification as described in this thesis allows for the creation of more expressive rules, i.e. things that had to be specified in multiple rules may now be captured in a single rule. GROOVE was until recently only able to handle rules that were matched existentially, but it is now also possible to alternate universal and existential matching to an arbitrary depth. Not only does this allow users to implement their rules more intuitively by specifying actions for groups of elements, it dramatically speeds up model evaluations by reducing the number of states created. With the sign-method described above, a large amount of states would be created while the *sign* was present. There was no way of telling GROOVE that the order in which transformations occurred in such a part did not matter, i.e. GROOVE would unnecessarily explore all possible orders.

A limitation of the current implementation of GROOVE was that NACs were implicitly created for every connected part that contained a NAC edge or node. This means that rule developers had to be careful with specifying NACs correctly. The use of nested quantification would require a mechanism to specify which elements are part of which NAC. This thesis project offered a good opportunity to fix this limitation.

This chapter defines new theory to support the notion of nested quantifaction in graph transformation rules. In order to keep the material readable and make the explanations of the new theory as intuitive as possible, a series of examples will appear throughout this chapter. The examples will aplly the theory that was introduced

before it and eventually define a quantified rule for Petri net execution. The next section introduces the concept of Petri net theory and shows how Petri nets may be modelled by a graph.

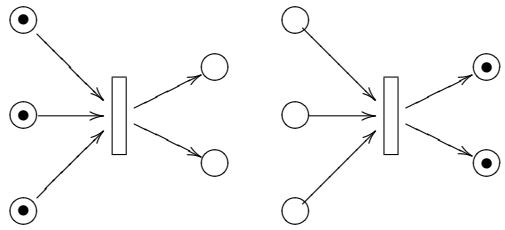
An example: Petri nets

Graph transformation systems can model virtually any problem, but require a translation into the graph formalism at hand, i.e. simple graphs in the case of GROOVE. Since Petri nets are already graph-like structures, there is no need for a complicated translation into simple graphs. It allows the reader to focus on the creation and application of nested rules. Instead of formally defining Petri nets and forcing the reader to read about two formalisms, a definition in natural language will be used.

In computer science Petri nets are used for things like workflow management, data analysis, concurrent programming, etc. When a set of processes have to be modeled that can execute tasks on their own, but have to wait for each other at some point, Petri nets can be used.

In short a Petri net consists of *places*, *transitions* and *arrows*. Transitions may be seen as synchronization points. They have *input places*, which are connected to it by arrows pointing from the place to the transition. They also have *output places*, which are connected to the transition by arrows pointing to the place. Transitions never have arrows to other transitions and places never have arrows to other places. A transition may have any number of input and output places.

Transitions can be either *enabled* or *disabled*. This depends on whether its input places have *tokens* or not. A token on a place indicates that the transition that has the place as an output place was *fired* some time ago. The transition that has this place as an input place must wait until all its input places have tokens, i.e. all other processes that synchronize here have completed their tasks. In Petri nets, a place may contain one or more tokens. Tokens have no value and no extra meaning, they are only needed for transitions to fire. A transition which has at least one token on every input place is called enabled. A transition which has at least one input place without any tokens is called disabled.



(a) Before transition (b) After transition

Figure 3.1: Sample Petri Net

Whenever a transition becomes enabled, due to the *firing* of other transitions, it may *fire* itself. A transition that fires consumes one token from each input place and produces one new token on each output place. Figure 3.1a illustrates a sample Petri net with one transition. The transition in this figure is enabled, since there are tokens on each input place, represented by the dots. When this transition fires, the situation illustrated in figure 3.1b is created. In figure 3.1b, the transition is disabled.

Now it becomes clear why transitions can be seen as synchronization points. Whenever a token is created on an input place of a transition, that token has to *wait* until all other input places also have a token, after which the process may continue past the synchronization point, or transition.

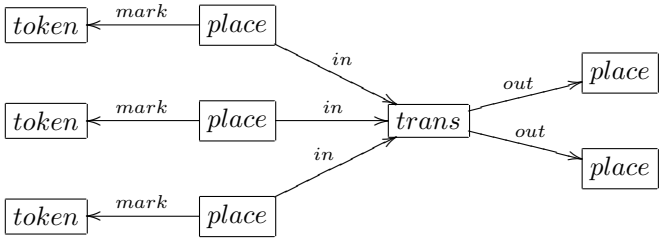


Figure 3.2: Sample Petri net from figure 3.1 as a simple graph in GROOVE

In Petri net theory, a transition shown in figure 3.1 is a single atomic step, i.e. it can not be interrupted by other transitions. When modeling Petri nets in a graph transformation system, it is desirable to capture this behavior in a single transformation rule. A naive solution would be to take the two graphs from the example in figure 3.1 and define a rule which says *"find the (sub)graph of figure 3.1a and replace it with figure 3.1b"* . However, such a rule will only match this particular (sub)graph, or worse, it will match any transition with at least three input places with tokens and at least two output tokens, whether there are any input places without tokens or not. This approach would require a rule for every possible transition in your model and every rule must check that it is not being matched with a transition it should not handle. A better rule would be *"find a transition which has tokens on all its input places, remove those tokens and add tokens to all output places"*.

This chapter will introduce the theory required to create such a general rule for Petri net transitions and will also gradually create that rule in the example sections.

3.2 Nesting Application Conditions

Negative application conditions (NACs) as proposed in (Ehrig, Ehrig, Habel, & Pennemann, 2004; Rensink, 2004) and briefly described in section 2.2.3 have been in use for quite some time now. A more recent proposal in (Rensink, 2006) loosely shows how NACs can be used to extend graph transformations with universal quantification. This chapter redefines several terms from that paper as well as introduce several new definitions to streamline the nesting of NACs.

The entire idea is built around two important notions: a *graph predicate* and a *rule tree*. A graph predicate combines levels of nesting to yield multiple levels of quantification, whereas the rule tree allows for the application of nested rules. In section 3.2.1 the graph predicate and several related definitions will be introduced, after which the notion of a rule tree will be defined in section 3.2.2 and section 3.3 defines how the two should be combined to perform graph transformations.

3.2.1 Graph Predicates

The creation of a nested rule requires two components: a *graph predicate* for computing the matchings of the rule and a *rule-tree* to perform the application. The construction used for matching is called a graph predicate, because a graph that contains a subgraph that has a structure as defind in the predicate is said to *satisfy* this predicate. Whenever a graph satisfies a predicate, a matching can be found. This section defines the theory required to create graph predicates and to compute the resulting matchings accordingly.

A graph predicate p consists of several graphs and connecting morphims and is a tree-shaped diagram in Graph. Together they represent a piece of first order logic

(Rensink, 2004). A graph H has a matching of a predicate p if there is a morphism f from the root of the predicate to the graph H that satisfies the predicate p. Definition 3.1 defines a graph predicate as well as the satisfaction relation.

Definition 3.1 (graph predicate)
A graph predicate p *over* $G \in$ Graph *is a tree-shaped diagram in* Graph *rooted in* $G(= root_p)$. *The diagram* p *is called* ground *if* G *is the empty graph, written as* $G = \emptyset$.

Let $H \in$ Graph *be an arbitrary graph and* $f : root_p \to H$ *a morphism. Predicate* satisfaction *is a binary relation* $\models$ *between predicates and morphisms:* $p \models f$ *expresses that the morphism* f *satisfies* p. *The* $\models$ *relation is defined as the* smallest *relation such that* $p \models f$ *if the following condition holds:*

- *There is at least one* $g : G \to K \in init_p$ *and* $h : K \to H$, *such that* $f = h \circ g$ *and* $sub_p(K) \not\models h$

A host graph H *is said to satisfy* p, *denoted* $p \models H$, *if* $p \models f$ *and* $f : \emptyset \to G$. *Note that* p *has to be ground.*

Just like a first order logic formula can have free variables, graph predicates can have free variables. These should be defined in the root of a predicate, which is why the root of a predicate is also called the *context graph*. This context graph always resides on depth 0 (or level 0) in the predicate. However, this thesis uses only *ground* predicates, i.e. the context graph will always be the empty graph. Future implementations may extend this behavior and allow the use of free variables, but for now free variables in a predicate are not used.

Taking a closer look at the definition reveals that the graphs at depths or *levels* $1, 3, 5, ...$ (i.e. odd levels) in the predicate should be existentially matched, much like a normal SPO rule. A normal rule can also be captured in a graph predicate. It would be a predicate with the left graph on the first level and any potential negative application conditions on the second level.

The graphs at the levels $2, 4, 6, ...$ (i.e. even levels) constitute either a negative application condition or a graph that should be universally matched. For a graph K at a universal level the difference between being a NAC or a universal condition is whether $out_d(K)$ is empty or not. If $out_d(K)$ is empty for a graph at a universal level, the graph K is a negative application condition. If $out_d(K)$ is non-empty, it is a universal condition and it should be matched accordingly.

Given this distinction in levels, the satisfaction relation may be more intuitively described as *existential* ($\models_\exists$) and *universal* ($\models_\forall$) relations.

Definition 3.2 (predicate satisfaction)
- *A graph predicate* p *is existentially satisfied by* $f : G \to H$, $p \models_\exists f$ *if:*
  - *There is at least one* $g : G \to K \in init_p$ *and* $h : K \to H$, *such that* $f = h \circ g$ *and it holds that* $sub_p(K) \models_\forall h$.
- *A graph predicate* p *is universally satisfied by* $f : G \to H$, $p \models_\forall f$ *if:*
  - *For all* $g : G \to K \in init_p$ *and all* $h : K \to H$, *such that* $f = h \circ g$ *and it holds that* $sub_p(K) \models_\exists h$.

In words the above definition states that for existential satisfaction, a (sub)predicate p is satisfied by the morphism f if there is at least one arrow from $root_p$ to an existential condition K and also a morphism from K to the host graph G, such that the morphism f can be decomposed into the arrow from $root_p$ to K ($\in init_p$) and

the morphism from K to G. This morphism from K to G must then be universally satisfied by the subpredicate $sub_p(K)$.

For universal satisfaction, a (sub)predicate p is satisfied by a morphism f if for each arrow from $root_p$ to a universal condition K there are morphisms from K to the host graph G, such that f can be decomposed into the arrow from $root_p$ to K ($\in init_p$) and the morphism from K to G. For each morphism from K to G that decomposes f like this, the subpredicate $sub_p(K)$ must be existentially satisfied by that morphism. If this condition has no arrows in $init_p$, but there is a morphism from $root_p$ to G, then the predicate can not be satisfied and this condition counts as a NAC.

In order to use these more intuitive notations, it is imperative to prove they are equivalent to the earlier definition of predicate satisfaction. Luckily they are equivalent as proposition 3.3 and proof 3.4 will show.

Proposition 3.3
Let p be a graph predicate and $f : root_p \to G$ a morphism from $root_p$ to some arbitrary host graph $G \in \mathsf{Graph}$. Then, $p \models_\exists f$ iff $p \models f$ and $p \models_\forall f$ iff $p \not\models f$.

This proposition relies on the size of the predicate tree. The following proof inductively proves proposition 3.3.

Proof 3.4

- *Size 0: $sub_p(K)$ is a leaf, $out_p(K)$ is empty*
  - $p \models_\exists f \quad = \quad p \models f$, *since $sub_p(K) \not\models f$ vacuously ($out_p(K)$ is empty)*
  - $p \models_\forall f \quad = \quad p \not\models f$, *since $sub_p(K) \not\models f$ vacuously ($out_p(K)$ is empty)*
- *Size n: the hypothesis holds for $sub_p(K)$*
  - $p \models_\exists f \Rightarrow p \models f$: *there is a $g : G \to K \in init_p$ and a $h : K \to H$, $f = h \circ g$ and $sub_p(K) \models_\forall h$. Rewriting the universal satisfaction to an existential yields that $sub_p(K) \not\models_\exists h$ which holds due to the induction hypothesis.*
  - $p \models_\exists f \Leftarrow p \models f$: *there is a $g : G \to K \in init_p$ and a $h : K \to H$, such that $f = h \circ g$ and $sub_p(K) \not\models h$, which is equivalent to $sub_p(K) \models_\forall f$ due to the induction hypothesis.*
  - $p \models_\forall f \Rightarrow p \not\models f$: *for all $g : G \to K \in init_p$ and all $h : K \to H$ it holds that $f = h \circ g$ and $sub_p(K) \models_\exists f$. Rewriting the universal satisfaction as an existential one yields: $p \not\models_\exists f$: there is no $g : G \to K$ and $h : K \to H$ such that $f \neq h \circ g$ and $sub_p(K) \not\models_\exists h$. Removing the negations on both sides yields the requirement from defintion 3.1.*
  - $p \models_\forall f \Leftarrow p \not\models f$: *this proof is the same as the previous, only mirrored.*

□

In logic, predicates are valued by a proof of satisfaction, in this case a graph predicate is valued by a matching. In order to use the satisfaction relation described above, a proof of satisfaction must be formalized to value the graph predicate and create a matching. The result of definition 3.5 is a collection of morphisms from graphs in the predicate to the host graph. These morphisms are related to each other in the same way their domains are related in the predicate. The important thing is that for each matched instance of a universal level, there is a morphism on an underlying existential level that projects the left side of the rule onto the host graph.

Since universally matched parts of the predicate now have morphism both on the universal level and an underlying existential level, it makes sense to only use one of these morphisms. Therefore rules may only be defined on existential levels and all

of the morphisms which originate from an existential level eventually constitute the matching of the nested rule.

Definition 3.5 (proof of satisfaction)
Let p be a graph predicate over G and let $f : G \to H$ be a graph morphism.

- *A* proof of existential satisfaction $\phi : p \models_\exists f$ *is a triple* $(g : G \to K, h : K \to H, \psi)$*, where* $g \in init_p$*,* $f = h \circ g$ *and* $\psi : sub_p(K) \models_\forall h$ *is a proof of universal satisfaction*
- *A* proof of universal satisfaction $\psi : p \models_\forall f$ *is a partial function* ψ *such that for all decompositions* $f = h \circ g$ *with* $g : G \to K \in init_p$ *and* $h : K \to H$*, the image* $\psi(g, h) : sub_p(K) \models_\exists h$ *is a proof of existential satisfaction*

Before this theory is put into practice in the next example section, one last property of a predicate is required. Every graph predicate has *existential subtrees*, which are important when a nested rule is to be applied. Essentially, it takes a predicate and filters out all the universal levels. This is formalized in definition 3.6.

Definition 3.6 (existential subtree)
Let p be a graph predicate, the set of existential subtrees *of p is defined as* $T_p^\exists = \{exsub_p(H) | G \to H \in init_p\}$*, where* $exsub_p(H)$ *is the existential subtree in p originating in H, defined as* $exsub_p(H) = \{(g \circ f, exsub_p(L) | f : H \to K, g : K \to L \in p\}$

Since a graph predicate starts with a context graph and the first level below the context graph contains existential conditions, this definition yields a set of existential subtrees. In a give graph predicate p, there is exactly one existential subtree rooted in K for each arrow $f : root_p \to K \in init_p$.

Petri net Graph Predicate

In the process of creating a nested rule, the first step would be to create a predicate. It will specify in one diagram what should be matched. Building a predicate can start with writing in words what should be matched. For a Petri net rule this would be:

"Find a transition, then find all input places with a token, then find all output places. Ensure there are no input places without a token."

This statement can be easily written in first order logic. The following formula specifies exactly what the above statement says:

$$\exists x \in T : \forall y \in P : (in(x, y) \wedge mark(y)) \vee out(x, y) \wedge \nexists z \in P : in(z, y) \wedge \neg mark(z)$$

In this formula, the set T represents all transition nodes and P all place nodes. The relation $in(x, y)$ says that y is an input place of x and $out(x, z)$ says that z is an output place of x. The last part of this formula is actually superfluous, by requiring all incoming places to have a token, requiring that there should not be an input place without a token is redundant. This reduces the formula to the following:

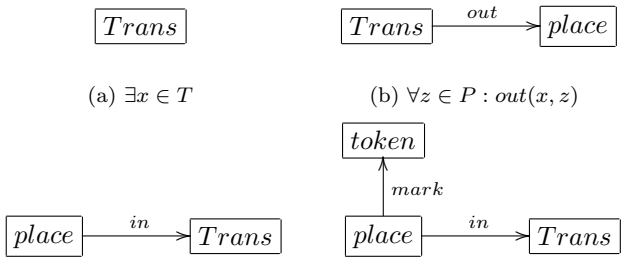
$$\exists x \in T : \forall y \in P : (in(x, y) \wedge mark(y)) \vee out(x, y)$$

Translating this to a graph predicate starts at the left of the formula. The first thing that needs to be translated is $\exists x \in T$, an existentially matched transition (see figure 3.3a).

The next part that will be translated is universally matched and the *or* in this part of the formula causes there to be two seperate levels of universal matching. The

easy part is matching all output places ($\forall z \in P : out(x, z)$), seen in figure 3.3b. The universal condition and the existential condition for this part are the same.

The part that will have to match all input places ($\forall y \in P : in(x, y)$) with a token ($\wedge mark(y)$) is slightly more complicated. To correctly enforce the existence of a token, this part will be broken up into two steps. The first step is to match all incoming places (see figure 3.3c), which will be done at a universal level. The next step is to enforce enforces each matched input place to have a token (see figure 3.3d), which is done at the existential level beneath figure 3.3c. If one of the input places matched by figure 3.3c does not match figure 3.3d, the entire matching will be invalidated due to definition 3.2.



(a) $\exists x \in T$ (b) $\forall z \in P : out(x, z)$ (c) $\forall y \in P : in(x, y) \wedge mark(y)$ (d) $in(x, y) \wedge mark(y)$

Figure 3.3: Petri net predicate graphs

All these graphs have to be combined into a graph predicate to perform the overall task of matching. Figure 3.4 roughly shows the structure required. The morphisms from 3.3a to its 3.3b and 3.3c project the transition, so all matches will use the same transition as the base for matching the places. The morphism between 3.3c and 3.3d is a total morphism, i.e. all elements are projected. The morphism between 3.3b on level 2 and on level 3 is actually the identity. The numbers on the left of the diagram represent the predicate level on which the graphs reside, these are for illustrative purposes only.

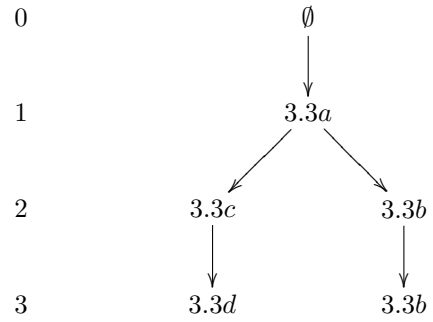


Figure 3.4: Petri net graph predicate

The reason that the subgraph in figure 3.3b appears twice in the diagram is that it has to be matched both universally and existentially. The appearance on level 2 makes sure *all* matching places are matched and the appearance on level 3 allows the transformation of each matched instance. Without an appearance on level 3, the graph on level 2 would be interpreted as a negative application condition. In the next section, rule trees will specify how elements matched by a predicate should be transformed.

3.2.2 Rule trees

With predicates defined as the mechanism to match quantified graph structures, it is time to define transformations on such structures. Since the process of matching has already caused there to be a tree-like structure between the morphisms, it is only logical to define the transformations in a similar fashion. A *rule tree* is therefore a set of normal SPO graph transformation rules which are nested. More formally, a rule tree is a tree-shaped diagram in the category Rule.

Definition 3.7 (Rule category)
The category Rule *has graph morphisms (rules) from* Graph *as objects and rule morphisms as arrows. A rule morphism $\rho : r \to s$ between rules r and s is a tuple of total graph morphisms (ρ^L, ρ^R), where $\rho^L : L_r \to L_s$ and $\rho^R : R_r \to R_s$ such that in* Graph *the diagram shown in figure 3.5 commutes.*

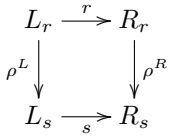


Figure 3.5: Rule morphisms commutativity

Note that the *rule morphisms* mentioned in the definition of Rule are morphisms between *rules*. They are not the morphism of a rule, which are graph morphisms in Graph. This distinction is required to create a nested structure of rules, where rules are connected to each other by a rule morphism. The following definition of a rule tree shows this.

Definition 3.8 (rule tree)
A rule tree *t is a tree-shaped diagram in* Rule *rooted in $r_0 (= root_t)$. Figure 3.6 displays a sample rule tree. Each rule $r \in t$ is called a* subrule *of t.*

The left subtree of a rule tree (t^L) is exactly the tree of L_{r_i} graphs and ρ^L_j morphisms. The right subtree of a rule tree (t^R) is exactly the tree of R_{r_i} graphs and ρ^R_j morphisms.

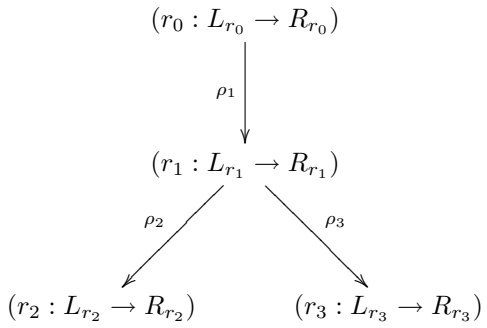


Figure 3.6: A sample rule tree

Definition 3.8 defines a left and right subtree of a rule tree. These subtrees are actually tree-shaped diagrams in Graph, since they have graphs as objects and graph morpshisms as arrows. Since definition 3.6 showed that a graph predicate can have a set of existential subtrees, it is necessary to select a left subtree of a rule tree to compare it with. This is because a rule tree can only work with a predicate if one the predicate's existential subtrees is exactly the same as the rule tree's left subtree.

All the definitions up to this point have been laying the foundation for the definition of a nested rule. A nested rule combines a graph predicate with a set of rule trees, which can then be used to match and apply complicated transformation rules.

Definition 3.9 (nested rule)
A nested rule is a tuple (p, T), where p is a graph predicate and T is a set of rule trees. Additionally the following has to be true: $\{t^L | t \in T\} = T_p^{\exists}$, i.e. the set of left subtrees must be equal to the set of existential subtrees of the predicate p.

Petri Net Rule Tree

The Petri net predicate created in section 3.2.1 will need to have a rule tree associated with it to be useful. This section describes the creation of the rule tree. Figure 3.7 recalls the predicate created in the previous example section. Each graph on an existential level gives rise to a transformation rule. In this example, there will be three rules, since there are three graphs on existential levels, being 3.3a, 3.3b and 3.3d.

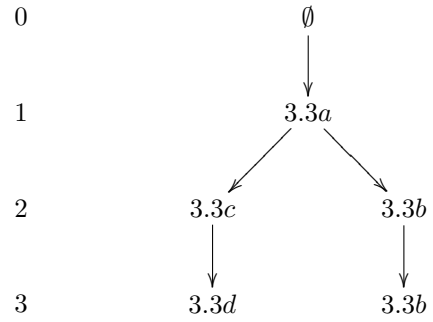


Figure 3.7: Petri net graph predicate

Figure 3.8 shows the individual rules used in the Petri net rule tree. The dotted lines between elements show the morphism projections from left to right. The top rule is simple, the transition does not change, this rule is represented in figure 3.8a, it merely projects the transition upon itself. The rule in figure 3.8b removes exactly one token from each input place, since it only projects the transition and the input place and it does not project the token. Finally, the rule in figure 3.8c places exactly one new token on each output place, since there is a new token element in the right graph.

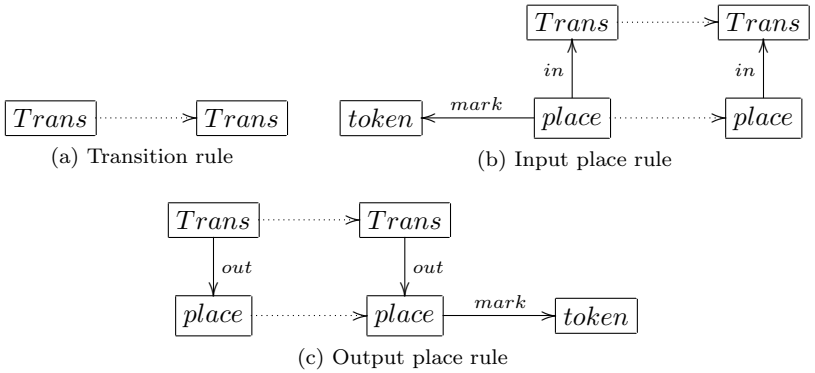


Figure 3.8: Petri net rules for the rule tree

Note how the left side of each rule is exactly the same as the corresponding graph in the graph predicate. The reason that there is no rule with the graph from figure 3.3c on the left side, is because this is a graph on a universal level. As noted earlier, changes on any universally matched element have to be done on an underlying existential level.

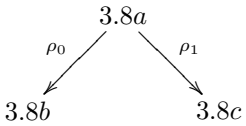


Figure 3.9: Petri net rule tree

The three subrules created so far are yet to be combined into a single rule tree. This requires the creation of two rule morphisms, one from rule 3.8a to 3.8b and one from 3.8a to 3.8c. Doing so results in a rule tree as depicted by figure 3.9. Both rule morphisms ρ_0 and ρ_1 merely project the transition from graph 3.8a onto the transition of graphs 3.8b and 3.8c respectively. The next section will describe how a matching is obtained from a nested rule.

3.2.3 Matching Nested Rules

Where normal rule applications are always associated with a single matching of the left side of the rule, nested rules may increase the number of matchings on each universal level $(2, 4, 6, ...)$ as long as there is another existential level beneath it. The new term *rule instance* will help to keep track of these matchings and their associated subrules.

Definition 3.10 (simple rule instance)
A simple rule instance *is a tuple (r, m), where $r : L \to R$ is a simple SPO rule and $m : L \to G$ is a matching of the rule into the host graph G.*

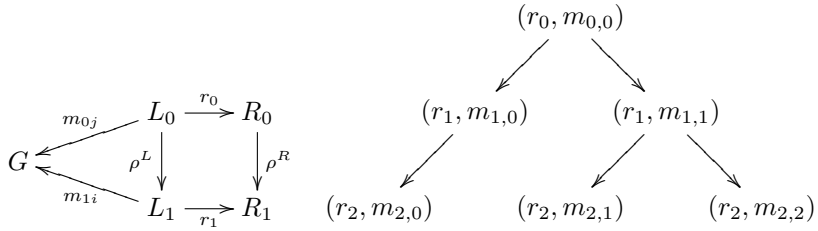
Extrapolating the previous definition to include instances of nested rules is slightly more complicated. Such a definition has to preserve the treelike structure of the nested rule, as well as capture the structure inherent to the matchings. For example, if a universal level gives rise to several matchings and there exists another universal level beneath it, the deepest level needs to be related to the correct parent-matching. This is done by defining another category Instance which has rule instances as objects and rule morphisms as arrows. The rule morphisms will ensure the correct nesting of the instances.

Definition 3.11 (nested rule instance)
A nested rule instance *is a tree-shaped diagram in the category* Instance*, which has tuples of a subrule and a matching (r_i, m_{ij}) as objects and rule morphisms as arrows. Additionally, for any rule instance (r_i, m_{ij}) and it's direct parent (r_{i-1}, m_{i-1j}) the diagram in* Graph *shown in 3.10a has to commute.*

An instance of a nested rule is derived from a proof of existential satisfaction of the associated predicate $p \models f$. Simply said, all the morphisms from graphs on an existential level to the host graph are placed in tuples, along with the subrule that is associated with that level, i.e. the domain of the matching must be the same as the domain of the subrule. Definition 3.12 formalizes this description.

Definition 3.12 (nested rule instance derivation)
A nested rule instance (R, M) is associated with rule R and M is a collection of rule instances. M is constructed from a proof of satisfaction $\phi : p \models f$ from the corresponding nested rule R as follows:

For each proof of existential satisfaction $(g : H \to L_i, h_i : L_i \to G, \psi)$ *in* ϕ*, there is a subrule* $r_i : L_i \to R_i$ *of* R*. The tuple* (r_i, h_i) *is added to the rule instance* M*.*



(a) Instances in Graph (b) Diagram in Instance

Figure 3.10: Nested rule instance

Now that rule instance have also been defined, it is time to show how these are obtained. The following section shows the derivation of instances for the Petri net example.

Petri Net Rule Matching

In order to demonstrate the matching of the Petri net predicate, recall the sample petri net from section 3.1 shown in figure 3.11. This figure shows indexes at the place-nodes to keep the construction of rule instances clear.

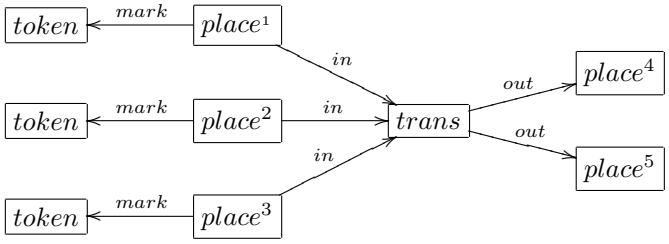


Figure 3.11: Petri Net host graph G

Now recall the rules created in section 3.2.2 and the predicate created in section 3.2.1. The predicate will match one transition, three incoming places with a token and two outgoing places in the host graph G. This means the example yields six rule instances. For illustrative purposes, the instances will first be created per rule.

Rule 3.8a gives rise to a single instance, it matches only a single transition in this graph and therefore also only a single rule instance. If the matching of 3.8a into G is called m_{Trans}, the rule instance becomes $(3.8a, m_{Trans})$.

Rule 3.8b can match exactly three input places, being $place^1$, $place^2$ and $place^3$. The predicate also matches the associated tokens and eventually yields the rule instances $(3.8b, m_{place^1})$, $(3.8b, m_{place^2})$ and $(3.8b, m_{place^3})$.

Rule 3.8c matches the two output places $place^4$ and $place^5$ and creates the rule instances $(3.8c, m_{place^4})$ and $(3.8c, m_{place^5})$.

The total nested instance is now created by combining the seperate instances into a single tree. Figure 3.12 shows the nested instance of the rule. In the next section, rule instances will be used to perform the actual rule application.

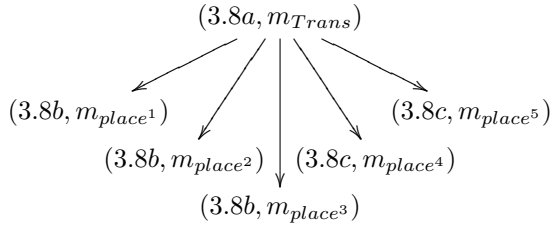


Figure 3.12: Petri net rule instance

3.3 Applying Nested Rules

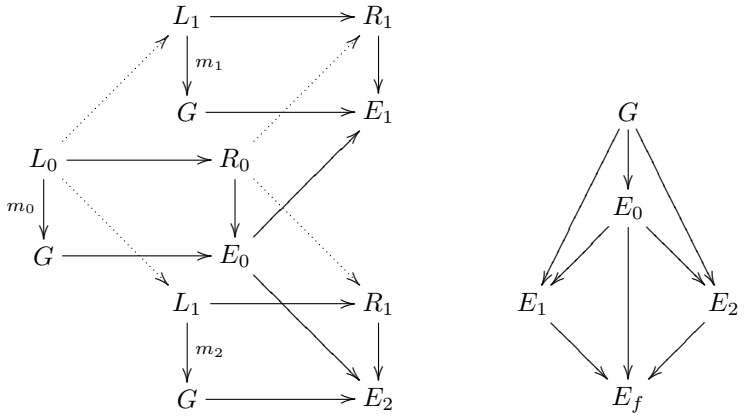
Application of a nested rule may be done in one of two ways. One is to perform a series of independent transformations, after which the resulting graphs have to be merged. The other is to amalgamate the rules into one large rule and execute the new rule. Before even considering to choose one over the other, a proof of equivalence of these two methods is required.

Section 3.3.1 will define and explain the method of applying each rule separately, followed by the amalgamation method in section 3.3.2. Section 3.3.3 will then prove the two methods to be equivalent.

3.3.1 Rule by Rule Application

The first method requires no additional work on the rules themselves before it can be applied. Given an instance of a nested rule (R, M), it simply applies all the rule instances in M separately.

Definition 3.13 (individual application)
Given a rule instance (R, M), I is a set of intermediate graphs created by applying the rules individually. In other words, I contains the pushout results E_j from all the (r_i, m_j) tuples in M.



(a) Three instances applied

(b) Final graph construction

Figure 3.13: Intermediate results and final result construction

In order to compute the final product of the application, all intermediate graphs have to be combined into one final graph H. Due to the universal property of pushouts and the morphisms between rules, there exist morphisms between the intermediate

results which preserve the structure of the rule tree. Figure 3.13a depicts the application of a nested rule with two subrules $r_0 : L_0 \to R_0$, which is matched once, and $r_1 : L_1 \to R_1$, which is matched twice. Now, because E_0 is the result of a pushout and there are morphisms $(G \to E_1)$ and $(R_0 \to R_1 \to E_1)$, there is a unique morphism $(E_0 \to E_1)$ such that the diagram commutes. Similarly, the morphism $(E_0 \to E_2)$ exists uniquely.

The morpshisms $(R_0 \to R_1 \to E_1)$ and $(R_0 \to R_1 \to E_2)$ exist because there are morphisms between the right sides of the rules. This follows from the definition of a rule tree (see definition 3.8). The morphisms $(G \to E_i)$ are the immediate result of definition 3.13.

The construction of the final graph now boils down to placing all the intermediate graphs E_i and the morphisms that connect them together in a diagram and compute the co-limit E_f of this diagram. Figure 3.13b displays this for the current example. Additionally, figure 3.3.1 displays the same process in the case r_1 would have matched three times. This is to show what happens if there are more intermediate results.

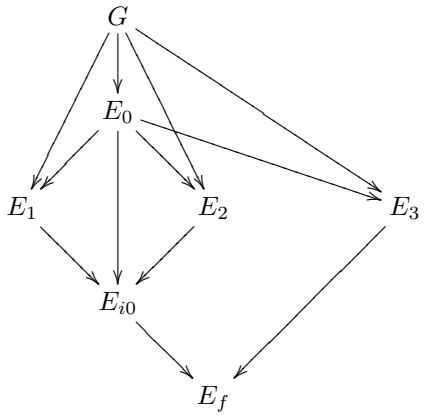


Figure 3.14: Final graph construction for four instance

Here E_{i0} is constructed in exactly the same way as E_f was in figure 3.13b. However, another pushout is required before the result is final. This last pushout is $(E_0 \to E_{i0}, E_0 \to E_3)$ and yields the final graph E_f. This process may be extended for any arbitrary number of intermediate results. It is essentially the creation of the co-limit of the diagram.

3.3.2 Amalgamation

Rule amalgamation is a technique to combine multiple rules into a single rule (Taentzer, n.d., 1996). It is used to execute rules that occur in parallel with each other. Currently, rule amalgamation has only been specified in conjunction with the DPO approach, however this section uses the same idea of combining rules into one large rule to perform quantified rule applications in the SPO approach.

The first problem that needs to be tackled for this approach to work is the fact that there are probably more rule instances than rules when applying a nested rule. This means it is insufficient to simply flatten the tree structure of the rule. The solution to this problem is to use the matchings of each rule instance to essentially create multiple copies of a rule. A property of the Graph category is that any morphism can be split into two morphisms with one new intermediate object. This is called epi-mono decomposition, because the morphism is decomposed into an epimorphism and a monomorphism. The newly created object is exactly the matched part of the host graph G, or the domain of the matching.

Definition 3.14 (matching decomposition)
Each matching $m_i : L \to G$ in the instance of the rule is decomposed into two morphisms $m_{ia} : L \to dom(m_i)$ and $m_{ib} : dom(m_i) \to G$, such that $m_i = m_{ib} \circ m_{ia}$ and $dom(m_i)$ is the domain of m_i in G.

After the matchings have been decomposed in this fashion, a large rule can be constructed by first computing the co-limit of the left subtree of the rule and all decomposed matchings. The right side of the rule will be constructed in a similar fashion later on.

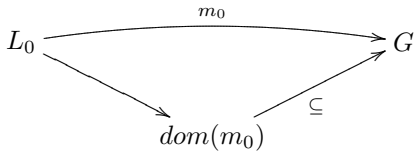


Figure 3.15: Rule matching decomposition

Definition 3.15 (rule amalgamation)
Given a nested rule instance, split all matching morphisms as in definition 3.14. Compute the colimit of the diagram with the left subtree of the rule and all created matching codomains by computing pushouts of all decomposed rule instances.

Figure 3.16 shows a small example of this approach. It consists of one nested rule with two subrules rules, of which only the left hand sides L_0, L_1 are shown, and three matches. The matches have already been decomposed and the amalgamated left hand side L_B has been constructed. The $_B$ subscript is to indicate this is a *big* rule. First the pushout of $(L_0 \to L_1 \to dom(m_1), L_0 \to L_1 \to dom(m_2))$ has been computed, which yielded the intermediate graph I_0. The large left hand side L_B was then constructed by computing the pushout of $(L_0 \to I_0, L_0 \to dom(m_0))$.

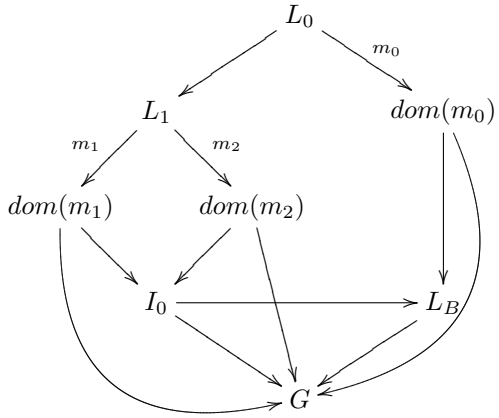


Figure 3.16: Rule amalgamation, part 1

The matchings remained consistent here due to the universal property of the pushout. The intermediate graph I_0 is the result of a pushout and there exist morphisms from $dom(m_1)$ and $dom(m_2)$ to G and therefore there is also a unique morphism from I_0 to G. The same holds for L_B, which has a morphism to G based on the arrows from I_0 to G and from $dom(m_0)$ to G. This morphism from L_B to G is the matching of the amalgamated rule in G.

In order to construct the amalgamated right hand side R_B of the rule, a pushout has to be computed for each decomposed matching, to yield the specific right hand

side. In figure 3.17 this pushout is shown for the subrule r_0 as the pushout $(L_0 \rightarrow dom(m_0), L_0 \rightarrow R_0)$.

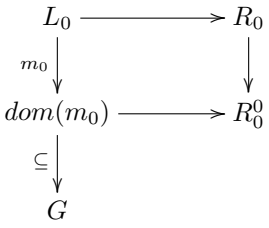


Figure 3.17: Rule amalgamation, part 2

The graph R_0^0 is a small part of the eventual right hand side graph R_B. This is done for all decomposed matchings and yields a series of copies of the right hand sides of all subrules. Together, these graphs and morphisms are put in a diagram similar to the building of the amalgamated left hand side.

Definition 3.16 (rule amalgamation, part 2)
Given the same nested rule instance with splitted matching morphisms, compute new intermediate right hand sides by computing the pushout of $(L_i \rightarrow dom(m_n), L_i \rightarrow R_i) = R_i^n$ *for each rule instance* (r_i, m_n)*. Construct the large right hand side by computing the co-limit of all* R_i^n *graphs.*

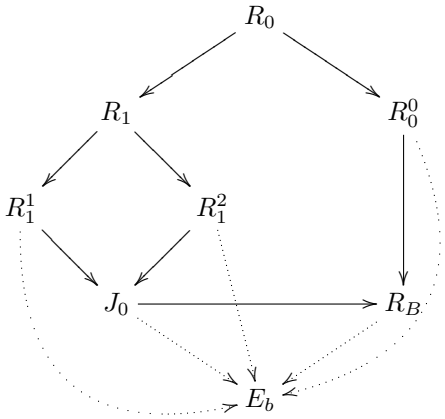


Figure 3.18: Rule amalgamation, conclusion

Figure 3.18 shows the construction of the right hand side R_B for the same example as figure 3.16. The dotted lines in this figure show the morphisms that will exist between these graphs and the final result E_B which will be constructed below.

Due to the universal property of pushouts, there exists a unique arrow between L_B and R_B, because L_B is the result of a pushout and there are arrows from $dom(m_0)$ to R_B (e.g. $dom(m_0) \rightarrow R_0^0 \rightarrow R_B$) and therefore there is a unique arrow between L_B and R_B. This unique arrow will be the new morphism for the amalgamated rule. In fact, there are many more arrows between the objects from figure 3.16 to the objects in 3.18, but showing them all in a single figure would cause the figure to be unreadable. All other arrows that exist can be derived in the same way as the arrow between L_B and R_B was derived.

Creating an amalgamated rule is not the end of this approach. The amalgamated rule has yet to be applied to the host graph G. As seen above, the amalgamated rule maintains a consistent matching in G and therefore can be applied. This application

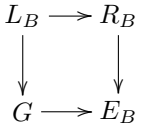


Figure 3.19: Applying the amalgamated rule

is shown in figure 3.19 and is actually little more than a single SPO application. It yields the result graph E_B.

3.3.3 Equivalence of Approaches

Coming up with two different approaches of solving the same problem requires one to wonder if these methods do the same or not. If they do not do the same, then where are the differences? In this section it is proposed that the two methods are equivalent and yield the same result. This proposition will also be proven.

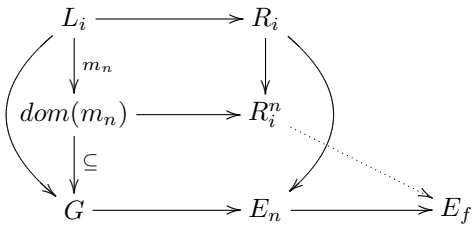
Proposition 3.17
Applying the method defined in section 3.3.1 yields a graph E_F and the method defined in section 3.3.2 yields a graph E_B. The graphs E_F and E_B are isomorphic.

Proving that these two methods are the same is not as easy as it may look. However, both approaches compute a lot of pushouts. Thanks to the universal property of a pushout, this causes many unique arrows between objects to be created as seen in the previous sections. In order to prove the proposition, some clever bookkeeping is required to see if eventually there are unique arrows between the results of both approaches.

Proof 3.18
To prove: $E_f \cong E_B$

Bookkeeping:

- *amalgamated approach: E_B is the pushout of $(L_B \to G, L_B \to R_B)$*
- *individual approach:*
  - *E_n ($n = 0..k$ is the number of rule instances) are the pushouts of $(L_i \overset{m_n}{\to} G, L_i \to R_i)$ (i is the corresponding subrule of the rule instance)*
  - *E_F is the co-limit of the graphs $E_0, \ldots, E_k$*
- *the graphs L_B and R_B are pushouts objects, there exist morphisms:*
  - *$L_i \to R_i$ – trivial*
  - *$L_i \to R_i^n$ – composed from $L_i \to R_i$ and $R_i \to R_i^n$ ($n = 0..k$)*
  - *$L_i \to R_B$ – composed from $L_i \to R_i^n$ and $R_i^n \to R_B$ ($n = 0..k$)*
- *the graphs R_i^n are pushouts of $(L_i \to dom(m_n), L_i \to R_i)$, there exist morphisms from $dom(m_n)$ to E_F and from R_i to E_f:*

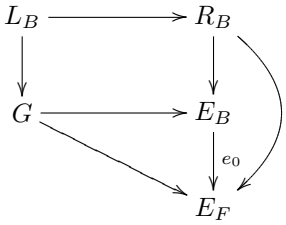


Therefore, there exist unique morphisms from each R_i^n to E_F

Proof:

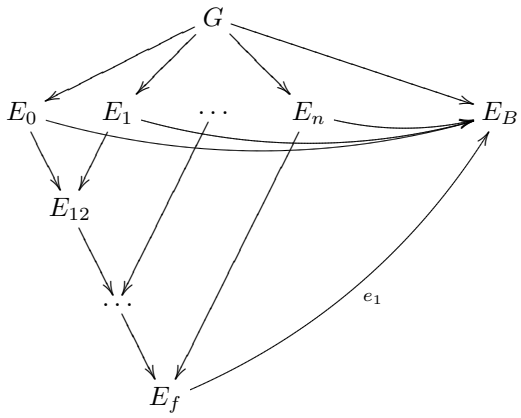
- *existence of morphism* $e_0 : E_B \to E_F$*:*

  The graph E_B *is a pushout of* $(L_B \to G, L_B \to R_B)$ *and there exist morphisms* $f : G \to E_F$ *and* $g : R_B \to E_F$*. The existence of* f *is trivial, since it is a direct result of definition 3.13 and can be obtained through any intermediate result. The morphism* g *exists because the graph* R_B *is also the result of a pushout and there exist morphisms from the graphs that result in this pushout* (R_i^n) *to* E_F *(see above). According to the universal property of pushouts, there also is a unique morphism* $e_0 : E_B \to E_F$*.*



- *existence of morphism* $e_1 : E_F \to E_B$*:*

  the graph E_F *is a pushout of all* $G \to E_n$ *morpshism and there exist morphisms* $h : G \to E_B$ *and* $k_n : E_n \to E_B$*. The existence of* h *is trivial, since it is a direct result of applying the large rule* $L_B \to R_B$*. The morphisms* k_n *exist because the graphs* E_n *are the pushouts of* $(L_i \overset{m_n}{\to} G, L_i \to R_i)$ *and there exist morphisms* $l : G \to E_B$ *and* $m : R_i \to E_B$*, again* l *is trivial and* m *is the composite arrow of* $R_i \to R_B \to E_B$*. According to the universal property of pushouts, there also exists a unique morphism* $e_1 : E_F \to E_B$*.*



- *Finally, since the arrows* $e_1 : E_F \to E_B$ *and* $e_0 : E_B \to E_F$ *are unique and* $E_F \overset{e_1}{\to}\overset{e_0}{\to} E_F$ *and* $E_B \overset{e_0}{\to}\overset{e_1}{\to} E_B$*, the two results are the same.*

□

Figure 3.20 shows both approaches in one abstracted diagram. Many intermediate objects and arrows have been left out, but both the use of the amalgamated rule $L_B \to R_B$ is shown, as well as the individual application intermediate results. Now that both approaches have been proven to yield the same result, the next section will show the last part of the example where the rules will be applied.

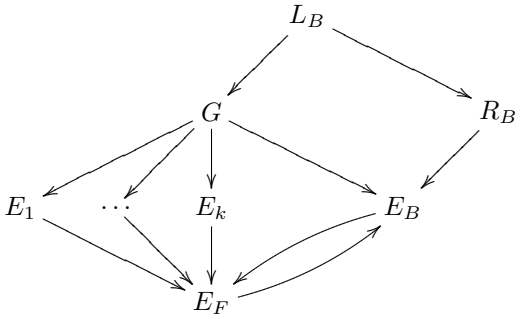
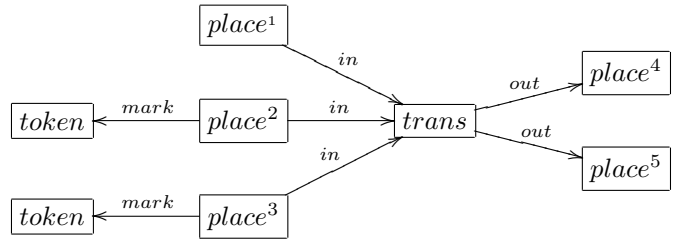


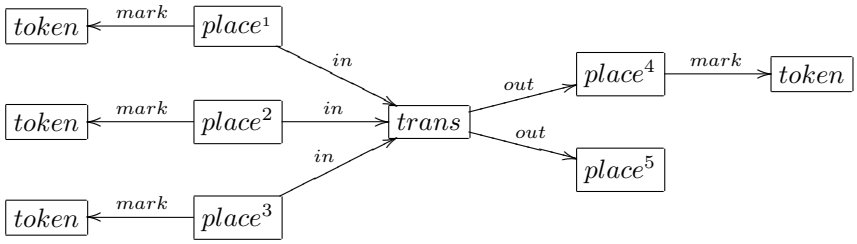
Figure 3.20: Abstracted view of both approaches

Petri net rules applied

The last step in the running example is to apply the rules and fire the transition. The approach shown here will be the individual approach. Each rule instance will be applied separately and then the results will be merged.



(a) Intermediate result from rule 3.8b



(b) Intermediate result from rule 3.8c

Figure 3.21: Petri Net host graph G

Recall that there were six rule instances in section 3.2.3, shown in figure 3.12. When these instances are applied, six intermediate result graphs are created, two of which are shown in figure 3.21. The other four are not shown, because one of them is exactly the same as the host graph G since the top rule does not change anything. Two of them are isomorphic with 3.21a and the other is isomorphic with 3.21b.

These six graphs are then combined again by computing the co-limit of the intermediate graphs. This process is illustrated in figure 3.22, where the six rule instances are shown. Note however that the rule instances are not used to compute a co-limit in Instance, but actually the co-limit of the intermediate graphs $E_0..E_5$ is computed in Graph. For the sake of the example however it is easier to show the rule instances again, since each one of them gives rise to one of the intermediate graphs.

The result of this application is shown in figure 3.23. Note how all the tokens from the input places have been removed and that the output places now each have a new token. This rule can now be used to simulate any Petri net model in GROOVE. Chapter 4 will even show how this rule should be built within GROOVE to yield the

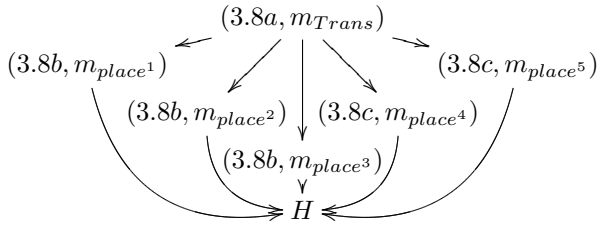


Figure 3.22: Petri net intermediate graphs merging

same result as shown in these example sections.

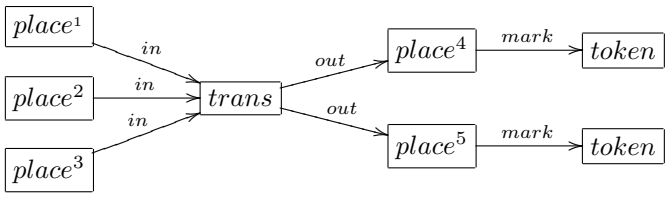


Figure 3.23: Petri net rule final result

4 Implementation

The theory of the previous chapter described precisely how nested quantification can be yielded in graph transformations. The goal of all this however is to extend GROOVE with such functionality. This chapter first describes a few graph transformation tools as related work before diving into the GROOVE tool and the implementation of nested graph transformations.

4.1 Related Applications of Graph Transformations

Unlike traditional string or term rewriting systems, until a few years back there were never a lot of graph rewriting implementations. Mainly because the research into graph rewriting systems was primarily focused on yielding theoretic results. The focus gradually shifted towards more practical use of graph rewriting systems since the appearance of the first graph grammar implementations, such as GraphEd(Himsolt, 1988). Nowadays there are several different implementations, all with their own specific properties, underlying formalisms, aims and features. This section will briefly look at a few of the implementations out there.

4.1.1 PROGRES

PROGRES (PROgrammed Graph REwriting System) (Zündorf, n.d.; Schürr, 2000) is a visual specification language for software engineering. It combines traditional textual specification with graphical data modeling. It is capable of interpreting its own models as well as generate Tcl/Tk and C(++) code for the systems defined. This makes it an ideal tool for fast prototyping.

Like GROOVE, PROGRES also uses the SPO approach, only with a powerful control language for queries and transactions. These transactions support a looping mechanism which effectively allows the user to universally quantify a rule, although it does not arise from a categorical construction.

4.1.2 GREAT

GREAT (Graph Rewriting and Transformation) is a tool for UML modeling. It aims at allowing users to specify platform independent designs. The choice for an implementation platform can then be delayed because GREAT allows for the automatic refactoring of a design to a specific implementation platform.

GREAT is build around the graph rewriting system Optimix, developed as a system for compiler optimizers. Optimix is not bound by a certain graph structure,

instead it can work on any graph, given a proper graph description. Graph transformations are done in a programmatic way, GREAT accepts a set of rewriting modules, which use the Optimix API to perform graph transformations.

4.1.3 AGG

AGG is a rule based visual language (Taentzer & Runge, 2005). It aims at the specification and prototypical implementation of applications with graph-structured data. AGG is also the research project within which the method of rule amalgamation was developed. The AGG environment provides a rule and graph editor and can analyze graph grammars and models.

The tool may also be used within a Java application to perform graph transformations on internal graph structures. This allows the tool to be used for more than model checking. For example, uses in artificial intelligence could be suggested to model and transform knowledge networks.

4.2 Methods of Quantification

Besides the concept of nested rules which this thesis introduces, other researches are trying to do similar things in different ways. This section briefly describes a few of these other methods.

4.2.1 Cloning and Expanding

In Hoffman, Janssens, and Eetvelde (2005), graph transformations are used to do refactoring in UML models. They describe how a generic push-down operation may be created by means of two techniques: cloning and variable expansion.

They show how ordinary rules are limited in their applications because they usually describe a specific instance of a problem. In order to let rules be applicable to more generic scenarios, they introduce *graph variables*. During the matching phase, these graph variables are *expanded* to a concrete subgraph based on the surrounding pattern. Since cloning requires a correct connection with the original hostgraph, they assign cardinalities to certain nodes, which are traversed when the subgraph is being copied.

Although it is a very powerful concept to use wildcards in a graph transformation system, the matching of such a rule becomes quite complex. However, interested minds should certainly keep an eye out for new publications from the University of Antwerp.

4.2.2 Transactions

Transactions can be used to perform a set of complex rules as one single atomic action (Baldan, Corradini, Foss, & Gadducci, 2006). However, transactions are not a native feature of graph transformations. They usually arise as a control language construction, much like the one in the PROGRES tool. While extremely useful, they require an amount of overhead, whereas rules that support quantification natively will be more natural.

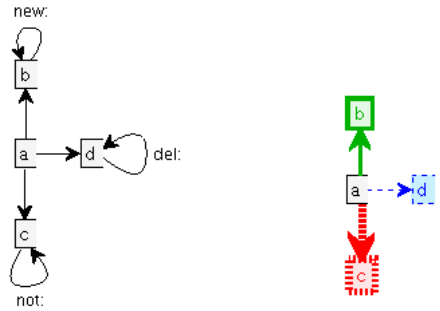
4.3 Implementation in GROOVE

From the very beginning, the design of the GROOVE tool has been focused on flexibility. This is demonstrated by the extensive use of generic interfaces allowing for implementations ranging from simple graphs and very simple rules to hypergraphs and extremely complex rule systems and even control languages and graph abstractions. Although not all of these features are in the currently available version, most of them are being researched while this thesis is written.

4.3.1 GROOVE rules

The formalism on which the current implementation of GROOVE is based is the SPO approach. GROOVE is set up to handle complete graph grammars, which consist of a set of graph transformation rules and one or more start graphs. The amount of control the user has over the execution of the rules is currently limited to a priority system. Rules with a higher priority disable those with lower priorities if they have a matching in the current graph. The use of a control language on top of a rule system is currently being researched and implemented. When the GROOVE release is extended with both nested rules and a control language, GROOVE will be a very powerful and flexible graph transformation system.

Within GROOVE, rules are created with the internal editor. This editor allows the user to set certain properties of an element by means of an aspect. An aspect is created and stored as part of an element's label. For example, figure 4.1a shows four nodes, each of a different type. The node with label *a* is an ordinary node. This node is not deleted, it is not created, it remains constant during transformation. The node with label *b* has a self-edge with label *new:*. The colon behind the word *new* indicates that this was an aspect value. The node (and all incident edges) should be created during transformation. The aspect values *del:* and *not:* specify a node that will be deleted and a node that may not be present respectively.



(a) A rule in the editor (b) A rule in the viewer

Figure 4.1: Rules in the GROOVE editor

Once a rule is parsed, these properties are stored and visually translated into a style for the node. The rule in figure 4.1b is exactly the same as the one in figure 4.1a, only now it has been parsed. New nodes and edges are shown in bold green, deleter nodes and edges are blue and dashed and NAC nodes and edges are bold, red and dashed.

4.3.2 Nesting GROOVE rules

One of the goals for this project was to come up with a notation for nested rules. Several attempts were made at graphical representations that captured the nesting of

rules, but none were deemed good enough to pursue. Therefore, the notation with the least impact on the code was chosen and nesting has been implemented as an aspect. Levels of nesting have to be explicitly defined and connected. See for example figure 4.2, which depicts the nested rule for a Petri net transition in the GROOVE editor. This is exactly the rule defined in section 3.2.2.

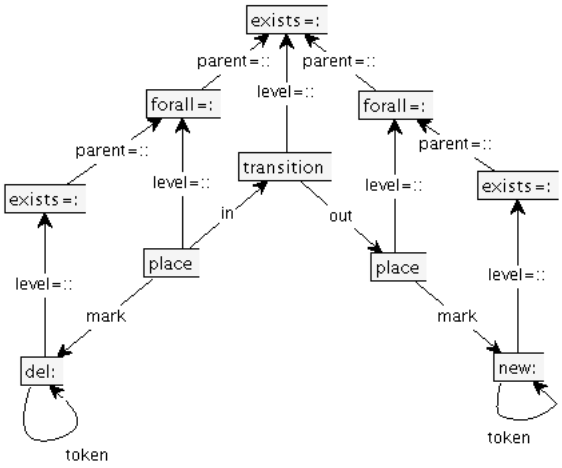


Figure 4.2: Petri net rule in GROOVE

The structure of the rule is represented by the meta nodes which are labeled *exists=:* and *forall=:* and also *nac=:* that is not in this example. These meta nodes are purely for the structure of the rule and will not be in the actual rule. During the parsing of this rule graph, GROOVE also automatically creates the graph predicate which will then be used for matching. The structure of the meta nodes defines how this predicate is formed.

A few simple rules dictate the overall way of defining a nested rule (see figure 4.2):

- At the top there should be exactly one meta node of type *exists=:*. Theoretically, a predicate can have any number of first level existential graphs, but the implementation only takes one. This is because such graphs constitute a logical or, which is already available by creating multiple rules.
- Meta nodes should be connected to their parent by means of a *parent=::* edge. Note the double colon in this label, this is required for all meta edges.
- Normal rule nodes should be connected to the level at which they are introduced in the predicate by means of a *level=::* edge to one of the meta nodes.
- The meta elements *exists=:* and *forall=:* should alternate each other in the structure.
- Edges between nodes are introduced on the level of the deepest node they are connected to (e.g. the *mark* edges are introduced on the level of the *token* node)

Looking at the definitions of predicates and rule trees, it seems changes to elements may only be placed on an existential level. However, when the implementation finds such behavior on a universal level, it is automatically pushed down to an existential level. In fact, when a *forall=:* meta node is the deepest in a structure, an implicit *exists=:* is created with exactly the same graph as the above universal level. This is to prevent the predicate from interpreting it as a negative application condition.

GROOVE supported negative application conditions by adding the aspect *not:* to an element. However, this created a negative application condition for each element

with such an aspect value. The user is now able to influence this behavior by declaring explicit *nac=:* meta nodes. A *nac=:* meta node should be beneath an existential meta node and creates exactly one NAC graph. All elements connected to the *nac=:* node are considered to be in one graph.

Because GROOVE does not allow edges from an edge to a node, it seems impossible to place an edge on an explicit level of nesting. However, the aspects allow the user to name the levels. For example, if an edge is supposed to be part of a certain negative application condition, one can change the label of the *nac=:* node to *nac=myFirstNAC:*. This associates the name *myFirstNAC* with that particular level in the predicate. If an edge now has the same aspect value (and a normal label), e.g. *nac=myFirstNAC:someEdgeLabel*, then this edge is not put into the predicate in the default way, but it is added explicitly to the referenced NAC. This also works for the *exists=:* and *forall=:* meta nodes. The labels on meta edges are not required to have the = after the type of the meta element, but the editor in GROOVE will automatically insert the = when the rule is parsed. It indicates that the meta element may have a name, but does not have one.

One last important note is that in the absence of any meta nodes, GROOVE parses the rule as a normal SPO rule. This also means that rules that were created before nested rules were introduced will still work within GROOVE.

4.3.3 Implementation specifics

This small subsection will describe briefly which elements within GROOVE have been altered to make the use of nested rules possible. This is mainly to help other people working on the project understand what changed within the tool. Readers that have no idea of the internal architecture of GROOVE may skip this section.

As was stated above, the implementation of nested rules should have as little impact on the tool as possible. Therefore the primary design was to try and extend the current implementation of rules. The current implementation uses three basic classes to perform SPO transformations: `SPORule`, `SPOEvent` and `SPOApplication`. A rule is captured in the class `SPORule`, which in combination with a matching can form an `SPOEvent`, which has some idea of what has to be changed, but does not have access to a specific host graph. This is where `SPOApplication` takes over, it uses the information stored in the `SPOEvent` and the given host graph to complete the transformation.

The idea to create classes `NestedRule`, `NestedEvent` and `NestedApplication` as specializations of their SPO counterparts proved to create more problems than it was thought to prevent. In the case of `NestedRule` there were no problems, a `NestedRule` is now a specialization of `SPORule` and maintains the rule tree and the predicate in one single tree structure. The `NestedEvent` and `NestedApplication` could not be used as specializations of their SPO counterparts, mostly because these two classes required extensive rewriting to take the nesting of rules in account. That is why these two classes are fresh implementations of the `RuleEvent` and `RuleApplication` interfaces.

The nested rules are edited without changes to the `Editor`, as the `Editor` used the class `AspectualRuleView` to convert a rule back into a simple graph and vice versa. This method was kept the same for nested rules, but instead there is now a `NestedAspectualRuleView` which can parse a nested graph with meta elements and will create a `NestedRule`.

To store the nested structure of a rule in a basic graph it was necessary to create a new `Aspect`, begin `NestingAspect` with its corresponding class `NestingAspectValue` which also defines several helper functions for dealing with `AspectGraphs` that contain elements with a `NestingAspect`.

In addition to these major changes, several smaller changes were made. For example, an interface `VarNodeEdgeMultiMap` and a corresponding implementation of said interface `VarNodeEdgeMultiHashMap` were created to capture the fact that certain rule elements may now be matched multiple times, so instead of mapping one element to one other element, an element can now be mapped to a `Set` of elements. Another small addition was the class `SPORuleMorphism` to create morphisms between rules.

Last but not least, the occurrences of `AspectualRuleView` within GROOVE were replaced with `NestedAspectualRuleView`. The same goes for the `SPORule`, `SPOEvent`, `SPOApplication` and `VarNodeEdgeMap` references.

When a nested rule is applied in GROOVE, several smaller applications occur under the hood. In fact, GROOVE collects all changes from the rule instances and applies the changes to the host graph. To the outside world, e.g. the transition system, this is just a single transition. Seeing a nested rule applied as a single transition helps again with the intuitiveness of these rules. If something happens in one step, it is shown and done in one step.

4.4 Results of using nested rules

The theory has been defined, the implementation described, but how do the new rules perform? This section shows a few sample models which have been created to test the impact of nested rules on statespace and computing time. The two models have been implemented twice in a GROOVE grammar, once with normal SPO rules and once with nested rules.

4.4.1 Gossiping Girls

In the gossiping girls problem, there are a number of girls who all know one secret each. The girls may randomly call each other and exchange secrets after which both girls know all the secrets either one of them knew. Figure 4.3 shows this model with four girls in the start state.

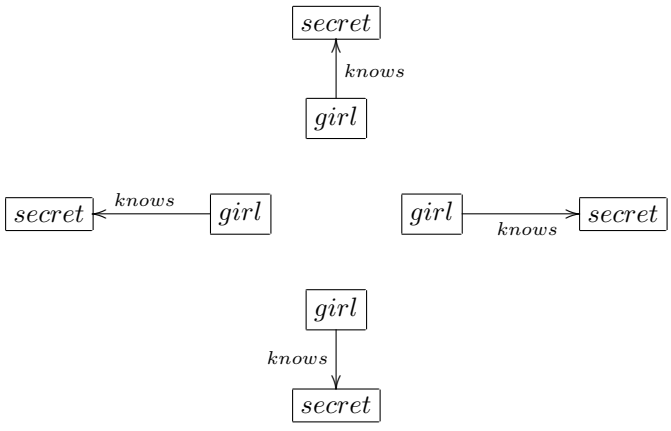


Figure 4.3: Gossiping girls example, start state

The girls will now randomly start calling each other, which is described by a rule that creates an edge with label *calling* between two girls. This rule has the restriction that there may only be one pair of girls calling each other on a given moment. After they have established a connection, the girls exchange secrets. In the old SPO approach, this had to be solved by two rules with a high priority that

transfer the secrets between the girls. In the new newsted approach, this is done in a single rule.

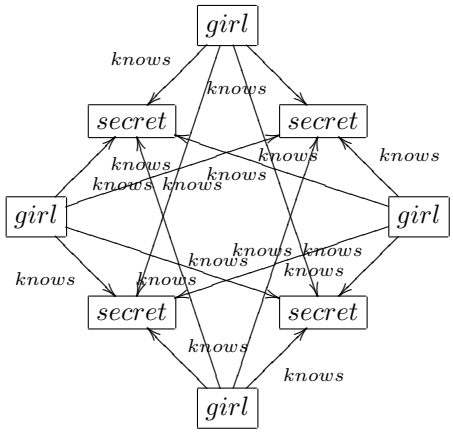
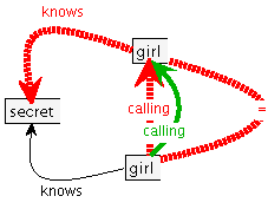


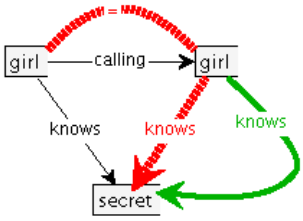
Figure 4.4: Gossiping girls example, final state

Figure 4.5 shows the rules used to simulate this problem. There is a small priority hierarchie between these rules to ensure correct execution. The rule that initiates a call (figure 4.5a) has priority 0, meaning that if no other rule is applicable, a call is initiated between two girls where one of them knows a secret the other does not know. The rule that stops a call has priority 1, meaning that is no secrets are left to copy, the call is ended (figure 4.5b). The two rules that copy secrets (figures 4.5c and 4.5d) have priority 2, which causes the secrets to be exchanged as soon as a call has been established.

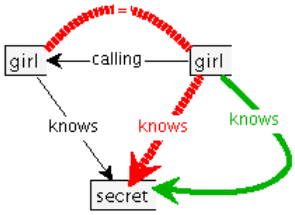


(a) Rule to establish call

(b) Rule to end call



(c) Rule to copy secrets (1)



(d) Rule to copy secrets (2)

Figure 4.5: Normal SPO rule for gossiping girls

Running these rules with the start graph from figure 4.3 yields the final graph shown in figure 4.4 after 334 milliseconds. During this computation, it created 115 states and 300 transitions.

By using nested rules, the two rules that copy secrets can be combined into a single rule. This rule is shown in figure 4.6 and copies all secrets known by either girl to the other girl. Running the model with this rule instead of the two separate copy

rules causes the computation time to be reduced to 221 milliseconds. The number of states is reduced to 36 states and there are only 136 transitions.

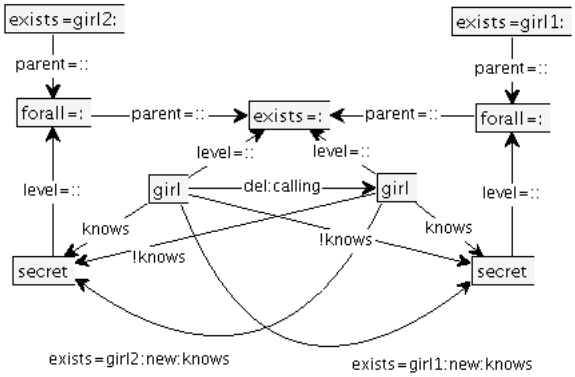


Figure 4.6: Gossiping girls, nested copy rule

The real increase is found when the number of girls and secrets are increased. The model was also run with five and six girls, at seven it could no longer be done within the maximum JVM memory. Table 4.1 shows the computation time, number of states and number of transitions for each model with both rule sets. It is obvious that the use of nested rules severely decreases the computation time and the statespace. The number of states is decreased by 62% for 3 girls, 69% for 4 girls, 74% for 5 girls and 78% for 6 girls. The same holds for the number of transitions: 55% less transitions for 3 and 4 girls, 62% less for 5 girls and 66% less for 6 girls. The increase from 3 to 6 girls is very interesting, since it gives an indication of what happens to a model if the size is doubled. The old rules create more than 550% more states and roughly a 1000% more transitions, whereas the new rules only cause a gain of 330% states and 750% transitions. It is safe to say that nested rules constitute more than a linear gain in efficiency.

Model	*SPO Rules*			*Nested Rules*		
	Time	States	Transitions	Time	States	Transitions
3 girls	108ms	21	40	76ms	8	18
4 girls	334ms	115	300	221ms	36	136
5 girls	1445ms	930	2825	1000ms	243	1087
6 girls	15286ms	11684	40139	5231ms	2657	13478

Table 4.1: Gossiping girls, results on a MacBook with 2G RAM

The results speak for themselves. Using nested rules reduces computing time and statespace significantly. Models with more than six girls failed to compute on the test machine due to a lack of memory, but on a newer JVM with a higher memory limit the nested rules will definitely still perform better than the old rules.

4.4.2 Petri nets

The running example from chapter 3 will demonstrate the enormous gain in power that the new rules yield. While creating the old rules for simulating a petri net, it became painfully obvious why nested rules are much easier for users. To correctly simulate a Petri net without nested rules, the firing of a transition has to be split up in several phases. First, each token connected to a transition must be marked until there is no input place left without a token. Then the transition will enter the *firing*

phase, indicated by a *firing* label on the transition. During firing, tokens are deleted from the input places and tokens are created on the output places. To ensure output places only get one new token each, they again have to be marked when they get a token. When this is done, the marks on the places have to be cleaned up and the extra edges have to be deleted. The rules used for this are shown in figure 4.8. To correctly simulate Petri nets with these rules, the mark and unmark rules need to have a higher priority than the other rules.

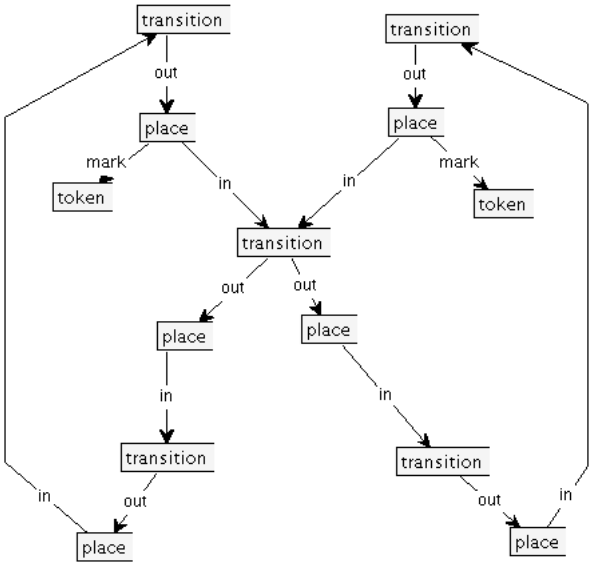


Figure 4.7: Petri net, start graph

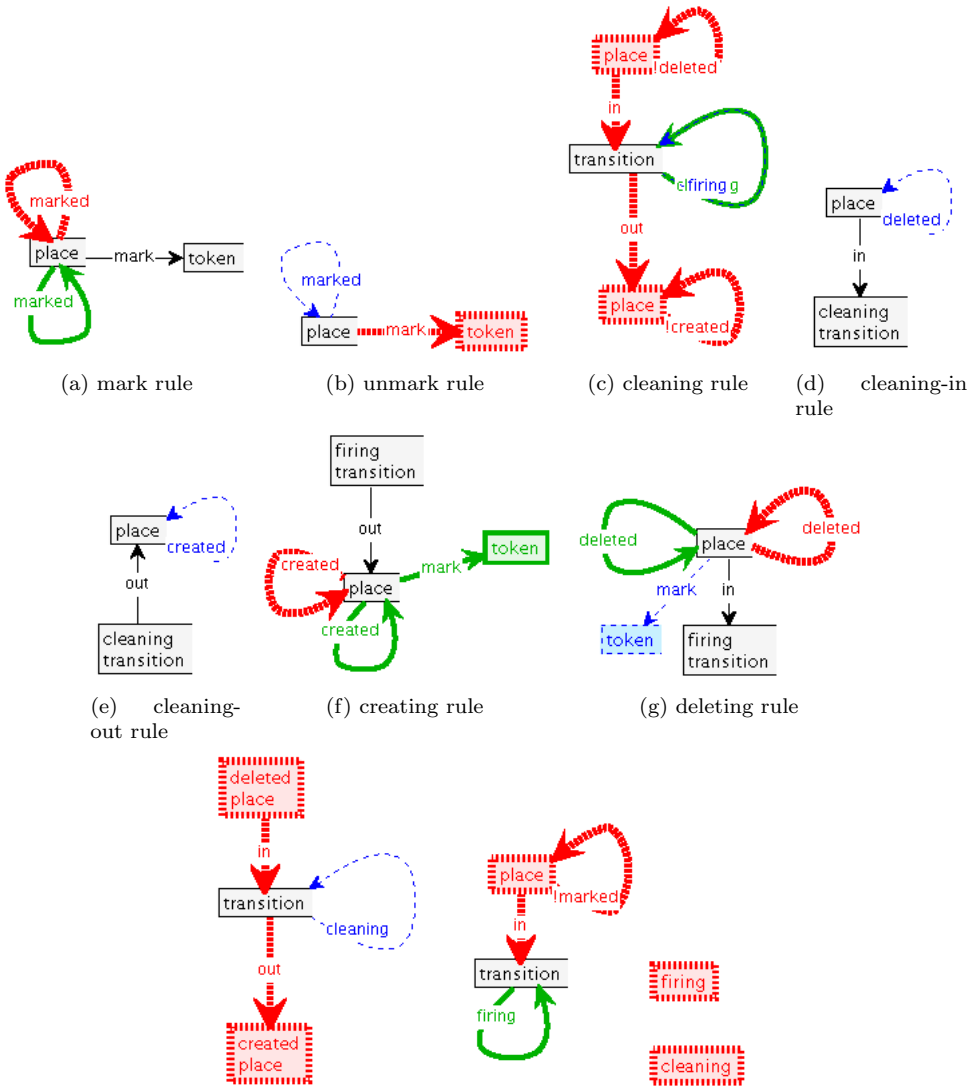
Figure 4.7 shows the host graph on which the rules were tested. It is a circular Petri net which is free of deadlocks, i.e. it will never end. The graph transformation system will execute one full circle of transitions before ending up in the same state again, which will be the final state.

Petri nets can be simulated with only one single nested rule. The rule used for this is the one created in the example sections of chapter 3. This rule is also depicted in figure 4.2. To conclude this chapter, table 4.2 shows the running time and statespace for simulating the Petri net from figure 4.7.

Model	*SPO Rules*			*Nested Rules*		
	Time	States	Transitions	Time	States	Transitions
3 girls	350ms	116	154	105ms	6	9

Table 4.2: Results for simulating Petri nets

Again the number of states and transitions are drastically reduced.



(a) mark rule (b) unmark rule (c) cleaning rule (d) cleaning-in rule

(e) cleaning-out rule (f) creating rule (g) deleting rule

(h) finishing rule (i) firing rule

Figure 4.8: Petri net, SPO rules

5 Conclusions

This chapter turns around and looks back upon the work done in this project. It will briefly discuss the result of each research question, conclude whether the project was a success or not and point out some future work.

5.1 Summary

This thesis started with introducing a graph formalism and some required definitions from category theory. In chapter 3, the notions of *graph predicate*, *rule instance* and *nested rule* were introduced. The examples throughout the chapter showed how the nested structure within graph predicates may be used to match quantified subgraphs within a host graph. It was then shown how a graph predicate gives rise to rule instances, which are a subrule together with its individual matching. These rule instances could then be applied to a host graph to obtain the desired result. Two methods for applying nested rules were introduced and proven to be equal.

The examples and results from chapter 4 showed how nested rules help to reduce the statespace of models. The number of states that were necessary to fully explore a model were drastically reduced when operations could be specified for groups of elements. The results were offset against the same models with older rules.

5.2 Discussion

Each research project is initiated by research questions. These questions sometimes change along the way, new ones may be added and some may even be left out. During this project most of the research questions were answered. One however failed to yield any real results. The question of what would be a useful notation for nested rules gave rise to a number of ideas, but none were even remotely feasible within the allotted time. The answer to this question thus remains elusive for now and GROOVE has been extended with a rather simple representation.

What methods of quantification are being considered in the field? Several methods were described as allowing for more than just existential matching. Cloning and expanding (Hoffman et al., 2005) showed some promise in matching subgraphs of unknown sizes, but currently lack a working implementation. The notion remains interesting however, since graph wildcards create powerful opportunities.

The notion of transactions as used by PROGRES (Zündorf, n.d.; Schürr, 2000) could also apply certain rules more than once in a single transaction. However, this technique is not a pure universal quantification, since any loops in the control language may be terminated prematurely.

How can universal quantification be defined in a formal way? Chapter 3 showed that using graph predicates to match graphs allows for universal and existential matchings to an arbitrary depth. Combining this with the power of a rule tree, many subgraphs can be transformed in just a single step. This chapter also showed two approaches to applying the nested rules, both of which were proven to yield the same result.

How should GROOVE be extended to support nested Application Conditions / predicates? Chosing to implement the nested rules with as little impact on the rest of the tool proved to be an effective approach. All of the features of the GROOVE generator and simulator can now work with both ordinary rules and nested rules. Chapter 4 described in a general outline how the implementation was done and what choices were made.

What is the impact on the statespace and computing time? Using nested rules showed dramatic decreases in statespace and computing time. The decrease of number of states and transitions required by the test model started at about 65% and grew towards 80%. The total gain in efficiency is estimated on an order of magnitude, if not more. However, there is a small tradeoff to be noted here. The computation of a nested rule takes slightly more time than a single SPO rule, but the use of nested rules reduces the number of applications significantly enough to gain dramatic efficiency increases.

During the process of composing the theory and implementing it, one always runs into problems. The first mistake made was coming up with two different approaches to apply the nested rules as it was now obviously necessary to prove if the two methods were the same or not. However, having to come up with an original proof proved to be a fun and educational journey.

The first idea for an implementation was shot down fast when the data generated by a nested rule superseded the capabilities of the original implemenation. This posed a problem as the new implementation would no longer be flawlessly integrated into the system. All in all, the final result is satisfying in its own sense.

5.3 Future Work

As was already described in chapter 4, GROOVE is an active research project and a lot of interesting ideas are currently being researched. Techniques like graph abstraction will allow for faster approximations, whereas a control language will increase the explicit specification power of the user. During the work done in this thesis project, a few ideas for improvement came up but were deemed beyond the scope of the project.

First of all, the current editor is not equipped to handle nested rules very well. The current solution with meta nodes specifying the nesting does what it should do, but it is far from beautiful and it clutters the rule graph. A new editor with support for nesting might be able to hilight certain parts of the graph if they are on the selected level. Alternatively, more colors could be used to distinguish between normal elements and universal elements. This is however not limited to the editor, the rule viewer currently loses all information about nesting and displays a nested rule as if it were a single SPO rule.

One of the mayor problems when using large graph transition systems is memory. GROOVE had a few optimizations for simple SPO rules, most of which were turned off to allow nested rules to work. Some research into these optimizations might yield versions which are compatible with the notion of nested rules and allow GROOVE to maintain a smaller memory footprint.

Besides all that, the work done in this thesis project allows users of the GROOVE tool to more effectively simulate their models. Operational semantics for Petri nets, UML activivty diagrams and similar models are now well within reach. It would even be possible to simulate pacman games or even complete rail networks in GROOVE now. Given the right model, GROOVE might now be able to find optimal train schedules for railway companies. All in all, this new addition of nested rules, together with the upcoming control language make GROOVE a tool that may very well be interesting to even commercial users.

References

Baldan, P., Corradini, A., Dotti, F., Foss, L., Gadducci, F., & Ribeiro, L. (n.d.). Towards a notion of transaction in graph rewriting. *Electronic Notes in Theoretical Computer Science*.

Baldan, P., Corradini, A., Foss, L., & Gadducci, F. (2006). Graph transactions as processes. In *Proc. of the 3rd int. conf. on graph transformations (icgt'06)* (pp. 199–214).

Barr, M., & Wells, C. (1990). *Category theory for computing science*. Prentice Hall.

Bauderon, M., & Jacquet, H. (2001, January). Pullback as a generic graph rewriting mechanism. *Applied Categorical Structures*, *9*, 65-82.

Bauderon, M., & Jacquetélène. (n.d.). *Categorical product as a generic graph rewriting mechanism.*

Christoph, A. (2002). Graph rewrite systems for software design transformations. In *Objects, components, architectures, services, and applications for a networked-world: International conference netobjectdays, node 2002* (Vol. 2591, p. 76-86). Springer-Verlag.

Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., & Rozenberg, G. (Eds.). (2006). *Proc. of the 3rd int. conf. on graph transformation (icgt'06).* Springer-Verlag.

Corradini, A., Gadducci, F., & Montanari, U. (1995). Relating two categorical models of term rewriting. In *Proceedings rewriting techniques and applications* (p. 225-240).

Corradini, A., Heindel, T., Hermann, F., & König, B. (2006). Sesqui-pushout rewriting. In *Proc. of the 3rd int. conf. on graph transformations (icgt'06)* (pp. 30–45).

Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., & Löwe, M. (1997). Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. In G. Rozenberg (Ed.), (chap. 3). World Scientific.

Ehrig, H., Ehrig, K., Habel, A., & Pennemann, K.-H. (2004). Constraints and application conditions: From graphs to high-level structures. In *Graph transformations* (Vol. 3256, p. 287-303). Springer-Verlag.

Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., et al. (1997). Algebraic approaches to graph transformation - Part II: Single pushout approach and comparison with double pushout approach. In G. Rozenberg (Ed.), (chap. 4). World Scientific.

Ehrig, H., Wolter, U., & Corradini, A. (2001, January). Double-pullback transitions and coalgebraic loose semantics for graph transformation systems. *Applied Categorical Structures*, *9*, 83-110.

Gorp, P. van, Schippers, H., & Janssens, D. (2006). Copying subgraphs within model repositories. *Electronic Notes in Theoretical Computer Science*.

Habel, A., Heckel, R., & Taentzer, G. (1996). Graph grammars with negative application conditions. *Fundamenta Informaticae*, *26*(3/4), 287–313.

Habel, A., & Pennemann, K.-H. (2005). Nested constraints and application conditions for high-level structures. In *Formal methods in software and systems modeling* (Vol. 3393, p. 293-308). Springer-Verlag.

Heckel, R., Muller, J., Taentzer, G., & Wagner, A. (n.d.). *Attributed graph transformations with controlled application of rules.*

Heckel, R., & Zündorf, A. (n.d.). *How to specify a graph transformation approach: A meta model for FUJABA.*

Himsolt, M. (1988). Graphed: An interactive graph editor. In *Proceeding stacs 89* (Vol. 349, p. 532-533). Springer-Verlag.

Hoffman, B., Janssens, D., & Eetvelde, N. van. (2005). *Cloning and expanding graph transformation rules for refactoring.*

Kastenberg, H., & Rensink, A. (2006). Model checking dynamic states in GROOVE. In *Model checking software* (Vol. 3925, p. 299-305). Springer-Verlag.

Klempien-Hinrichs, R. (1998). Net refinement by pullback rewriting. In *Foundations*

of software science and computation structures (Vol. 1378, p. 189-202). Springer-Verlag.

Kopka, & Daly. (2004). *Guide to LaTeX fourth edition.* Addison Wesley.

Mesegue, J. (1998). *Research directions in rewriting logic.*

Nickel, U., Niere, J., & Zündorf, A. (n.d.). *The FUJABA environment.*

Pierce, B. C. (1991). *Basic category theory for computer scienties.* The MIT Press.

Rensink, A. (2004). Representing first-order logic using graphs. In *International conference on graph transformations (icgt)* (Vol. 3256, p. 319-335). Springer-Verlag.

Rensink, A. (2006). Nested quantification in graph transformation rules. In *International conference on graph transformations (icgt)* (Vol. 4178, p. 1-13). Springer-Verlag.

Rensink, A., Kastenberg, H., & Staijen, T. (2004). *GROOVE (GRaphs for Object-Oriented VErification).* http://sourceforge.net/projects/groove/ (19-09-2006).

Rozenberg, G. (Ed.). (1997). *Handbook of graph grammars and computing by graph transformations* (Vol. 1 - Foundations). World Scientific.

Schürr, A. (2000). *A guided tour through the PROGRES environment.*

Stell, J. G. (1994). Modelling term rewriting systems by sesqui-categories. *Catégories, Algèbres, Esquisses et Néo-esquisses.*

Taentzer, G. (n.d.). *Parallel high-level replacement systems.*

Taentzer, G. (1996). *Parallel and distributed graph transformation: Formal description and application to communication-based systems.*

Taentzer, G., & Beyer, M. (n.d.). *Amalgamated graph transformations and their use for specifying AGG - an algebraic graph grammar system.*

Taentzer, G., & Runge, O. (2005). *Agg documentation.* http://tfs.cs.tu-berlin.de/agg/docu.html.

Walters, R. (1991). *Categories and computer science.* Carslaw Publications.

Wiemann, H. (2005). *Theory of graph transformations.*

Zündorf, A. (n.d.). *Graph pattern matching in PROGRES.*