The People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University Mohamed khider – BISKRA

Faculty of Exact Sciences and Sciences of Nature and Life

Computer Science department

Ordre :.....
Serie :.....

Project
Presented for the diploma of

Master in Computer Science

Option: **Software Engineering and Distributed Systems**

Title of Project :

Design and Implementation of a Graph Transformation Engine

Presented in 06/07/2019
By **Guerfi Maroua**

Board of Examiners :

Mr. Bennaoui Hammadi	**Prof**	**President**
Mr. Guerrouf Fayçal	**MAA**	**Supervisor**
Mr. Kerdoudi Mohamed Lamine	**MCB**	**Examiner**

Abstract

Graphs are used in a wide range of computer science domain to express complex statuses of systems. Graph transformation is used to model statuses changes of these systems. There are many tools to transform graphs. These tools do not separate their graph transformation engine from its graphical interface. This inhibits reuses their engine in other software. The aim of our work is to develop a graph transformation engine easily integrated into other software. Our engine follows the double pushout approach which is an algebraic approach of graph transformation. To demonstrate the use of our work we developed an application that uses the APIs and packages defined by our engine.

ACKNOWLEDGMENTS

I would first to thank ALLAH for giving me the strength, the audacity and, the endurance to realize this work.

My special thanks and appreciation to my supervisor **Mr.Guerrouf Fayçal** for his continuous encouragement, guidance and for his endless patience and precious advice. Thank you very much.

I would like to express my deepest thanks to the members of the jury: **Mr.Bennaoui Hammadi** and **Mr.Kerdoudi Mohamed Lamine** for reading and evaluating my dissertation.

I never forget to thank all my teachers at the Computer Science Department at Biskra.

DEDICATION

First of all, I would like to thank Allah for having given me the strength to finish this work.

I do offer my modest work to my parents, my brother and my sisters for their love, moral support, and outstanding patience during this long journey. And also to all my friends and classmates.

Lastly, I offer my regards and blessing to all those who supported me in one way or another during the completion of this work.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

General Introduction

In model-driven development (MDD), model transformation is used to save effort and reduce errors of modifications of models by applying these modifications automatically. The transformation is performed via a transformation engine, Usually, this later takes as input one model and provides one model as output. The model transformation is called either endogenous if the input and output models expressed in the same language, or Exogenous if the input and output models expressed using different languages. It specifies the meta-model to which a model must conform for specifying which models are acceptable as input and if appropriate what models it may produce as an output.

Graph transformation is an approach of model transformation. Its main idea is the rule-based modification of graphs. Graph transformation rules have an LHS and an RHS graph. The LHS is matched in the graph which will be transformed and replaced by the RHS. Each application of a graph rule leads to a graph transformation step. Moreover, graphs can be used to model the states of all kinds of systems, and the state changes in these systems are modeled by graph transformation.

There are several approaches to graph transformation, The algebraic approach [16] is one of them. It is based on the notions of category theory. This last is used to represent the graph transformation rules and rule application, and it helps to represent complex states at a high level of abstraction. An advantage of this approach is that a great part of the algebraic graph transformation theory is applicable to graphs, typed graphs, and attributed graphs.

There are many tools implement the concepts of graph transformation such as AGG [17], PROGRES [15], Fujaba [8], and VIATRA [2]. Every one of them offers various facilities such as editing graphs and rules, and different analysis techniques, etc. Their primary functionality is graph transformation, Therefore they came embedded with a graph transformation engine.

Unfortunately, this integration with the graphical interface makes it impossible or very hard to reuse these engines in other system.

To overcome this problem we propose in this project a new graph transformation engine. Our engine is designed to be reused independently from any other components. It is a library that defines and expose various APIs structured in different packages. These APIs allow us to manipulate and command the various operation needed to perform graph

transformation. Furthermore, our engine is implemented using Go programming language (a fast and easy language to learn designed by google).

Our engine implements Graph transformation following the double pushout approach [4]. It can transform (typed) graphs using rules. The application of rules is restricted to injective graph morphism. It supports two types of graph transformation:

- Non-deterministic: chooses randomly the rule to be applied.
- By rule priority: the rule with the elevated priority is applied first.

The remainder of this dissertation is organized as follows:

In the first chapter, we introduce all the basics and fundamentals concepts used in our project and illustrate some of them through simple examples such as graph transformation approaches, graphs, typed graphs, morphisms, graph match.

In the second chapter, we present the design of our engine. It is mainly two modules:

1. Graph Grammar: represent the structural part or a graph grammar and the operation related to it.
2. Graph Transformation: represent the operation that could be conducted on the graphs.

In the third chapter, we detail the implementation of our engine. We start by introducing the development tools and languages used. Then, we present its main data structures and functions defined in our system. lastly, we explain the usage of the APIs provided by our engine.

In the fourth chapter, we demonstrate how our engine is used as a library. Finally, we conclude this graduation note with a general conclusion.

Chapter 1

Fundamentals of Graph Transformation

1.1 Introduction

A wide range of fields in computer science and other areas of science and engineering use graphs to explain complex states of their systems and use graph transformation to model state changes of these systems [16]. The double pushout approach to graph transformation is an algebraic approach. It is one of the well-known approaches to graph transformation. It uses notions of category theory to describe graph transformation rules and rule application. The advantage of category theory is the high-level abstraction description of complex situations [1].

In this first chapter, we present definitions related to graph transformation and illustrate them with some examples.

1.2 Graph Transformation

Graph transformation is a formal approach for structural modifications of graphs via the application of transformation rules. A graph rule, also called production p = (L, R), consists of a left-hand side graph L, a right-hand side graph R, and a mechanism specifying how to replace L by R as shown schematically in figure 1.1 [6].

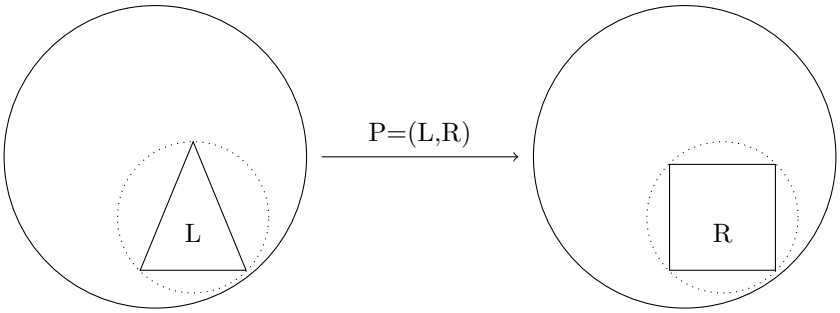


Figure 1.1: Rule-based modification of graphs[16, 6]

There are several approaches to describe Graph Transformation (GT), differing on the kinds of graphs that are used and how rules and their application are defined, which are :

1. Node label replacement approach.
2. Hyperedge replacement approach.
3. Algebraic approach.
4. Logical approach.
5. Theory of 2-structures.
6. Programmed graph replacement approach.

   In our project, we focus on the algebraic approach more precisely the double pushout approach. The interested reader is referred to [13] for more detail about other approaches.

1.3 Graph, Typed Graph, and Their Morphisms

Because graph transformations operate on graphs, we need to clarify what graphs. We consider directed graphs, Parallel edges and loops are allowed.

1.3.1 Graph

Conceptually, a graph is formed by nodes (or vertices) and edges connecting the nodes. Formally,a graph H = (V, E, s, t) consists of a set V of nodes, a set E of edges, and two functions $s, t : E \to V$ mapping to each edge its source and target node.[16, 6]

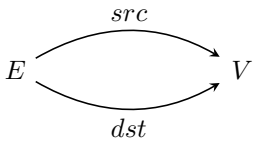


Figure 1.2: Graph [16, 6]

Example

The graph $H = (V, E, s, t)$, with node set $V = \{a, b, c, d\}$, edge set $E = \{u, v\}$, source function $s : E \to V : u, v \to a$ and target function $t : E \to V : u, v \to b$, is visualized in the following (figure 1.3):

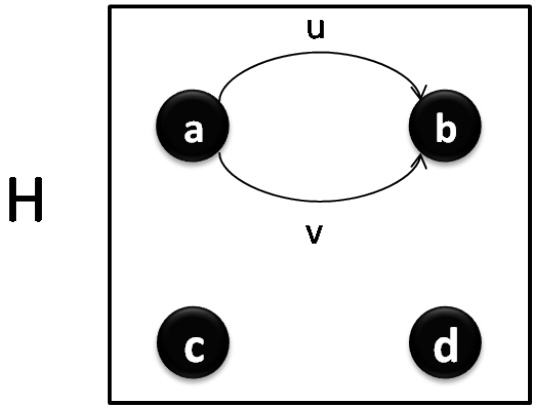


Figure 1.3: Example of Graph

1.3.2 Graph Morphism

Given graphs H and G, a graph morphism $f : H \to G, f = (f_V, f_E)$, consists of two functions $f_V : V_H \to V_G, f_E : E_H \to E_G$ that preserve the source and target functions, i.e. $s_G \circ f_E = f_V \circ s_H$ and $t_G \circ f_E = f_V \circ t_H$.[16, 6]

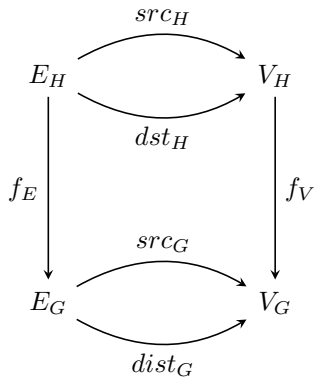


Figure 1.4: Graph Morphism[16, 6]

A graph morphism f is injective (or surjective) if both functions f_V, f_E are injective (or surjective, respectively); f is called isomorphic if it is bijective, which means both injective and surjective [16, 6].

Given two graph morphisms $f = (f_V, f_E) : G_1 \to G_2$ and $g = (g_V, g_E) : G_2 \to G_3$ the composition $g \circ f = (g_V \circ f_V, g_E \circ f_E) : G_1 \to G_3$ is again a graph morphism [16, 6].

Example

Given graphs $H = (V_H, E_H, s_H, t_H)$ and $G = (V_G, E_G, s_G, t_G)$, a graph morphism $f : H \to G$, as we show in figure 1.5 the morphism f map nodes as follow :
$f_V : a \to i; b \to j; c \to k$.

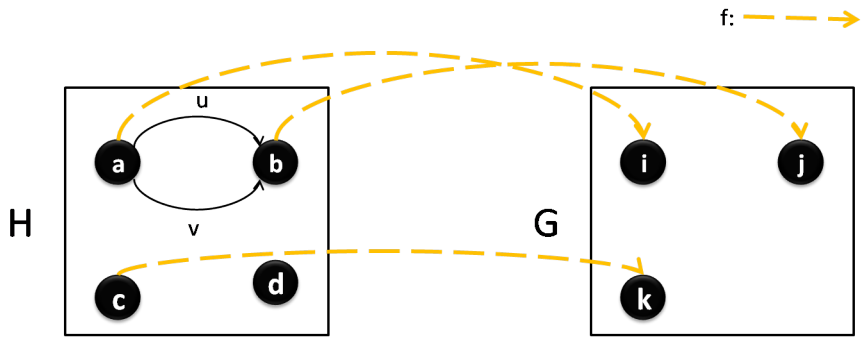


Figure 1.5: Example of Graph Morphism

1.3.3 Typed Graph

A type graph is a distinguished graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$. V_{TG} and E_{TG} are called the vertex and the edge type alphabets, respectively [16].
A tuple $(G, type)$ of a graph G together with a graph morphism $type : G \to TG$ is called a typed graph [16, 6].

Example

Consider the following type graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$ with $V_{TG} = \{x, y\}$, $E_{TG} = \{e\}$, $s_{TG} : E_{TG} \to v : e \to x$, and $E_{TG} = \{e\}$, $t_{TG} : E_{TG} \to v : e \to y$, and consider also the graph $H = (V_H, E_H, s_H, t_H)$, with node set $V_H = \{a, b, c, d\}$, edge set $E_H = \{u, v\}$, source function $s_H : E_H \to V_H : u, v \to a$ and target function $t_H : E_H \to V_H : u, v \to b$.

The graph H and the morphism $type = (type_V, type_E) : H \to TG$ with $type_V : V_S \to V_T : a, c \to x; b, d \to y$ and $type_E : E_H \to E_{TG} : u, v \to e$, is then a typed graph (typed over TG).

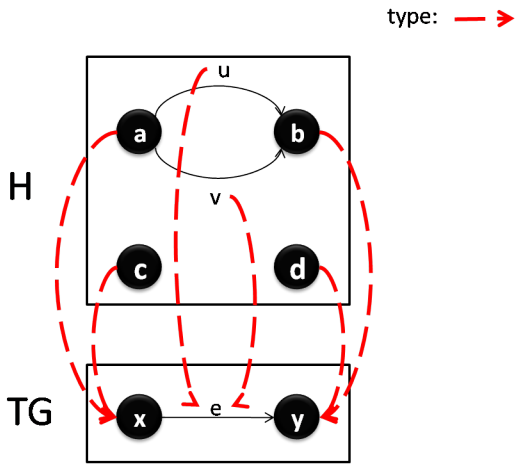


Figure 1.6: Example of Typed Graph

1.3.4 Typed Graph Morphism

Given typed graphs H and G, a typed graph morphism $f : H \to G$ is a graph morphism $f : H \to G$ such that $type_1 = type_2 \circ f$.[16, 6]

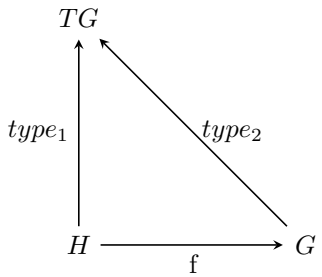


Figure 1.7: Typed Graph Morphism

Example

Given typed graphs $(H, type_1)$, $(G, type_2)$, And type graph TG. The typed graph morphism $f : H \to G$ is depicted in figure 1.8 by dashed yellow arrows.

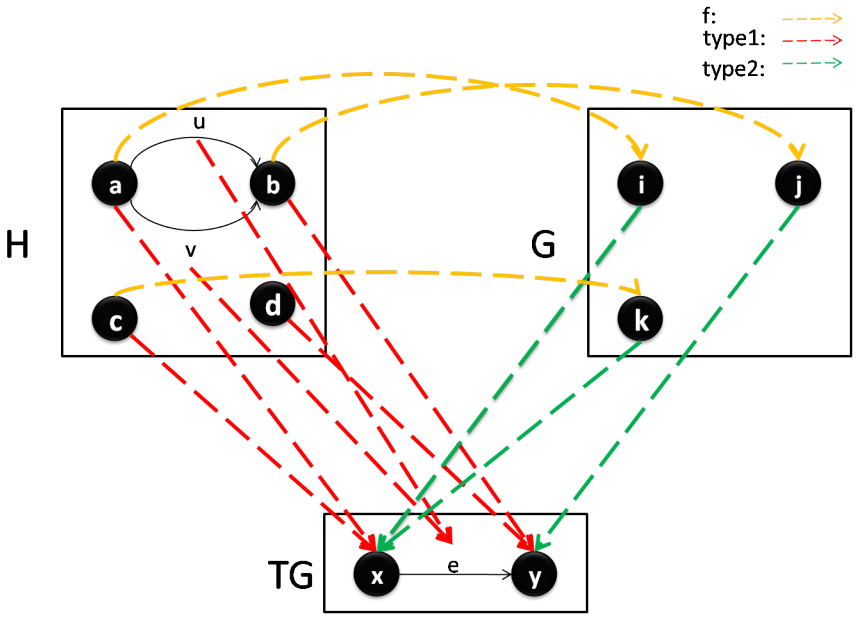


Figure 1.8: Example of Typed Graph Morphism

1.4 Category

A category is a mathematical structure that has objects and morphisms, with a composition operation on the morphisms and an identity morphism for each object.

Formally, a category $C = (Ob_C, Mor_C, \circ, id)$ is defined by [16, 6]:

- a class Ob_C of objects;
- for each pair of objects $A, B \in Ob_C$, a set $Mor_C(A, B)$ of morphisms;
- for all objects $A, B, C \in Ob_C$ a composition operation $\circ_{(A,B,C)} : Mor_C(B, C) \times Mor_C(A, B) \to Mor_C(A, C)$; and
- for each object $A \in Ob_C$, an identity morphism $id_A \in Mor_C(A, A)$;

such that the following conditions hold:

1. Associativity:For all objects $A, B, C, D \in Ob_C$ and morphisms $f : A \to B$, $g : B \to C$ and $h : C \to D$, it holds that $(h \circ g) \circ f = h \circ (g \circ f)$.
2. Identity: For all objects $A, B \in Ob_C$ and morphisms $f : A \to B$,it holds that $f \circ id_A = f$ and $id_B \circ f = f = f$

It is important to note that graphs and graph morphisms define the category *Graphs*. Also, typed graphs and typed graph morphisms define the category $Graph_{TG}$.

1.5 Gluing Relation

A gluing relation $\equiv$ in a Graph G in pair of equivalence relations $\equiv= (\equiv_E \subseteq G_E \times G_E, \equiv_V \subseteq G_V \times G_V)$ such that te following compatibility condition are satisfied [10]:

1. They identify only objects of the same lable :

$$v_1 \equiv_V v_2 \Rightarrow n^G(v_1) = n^G(v_2)$$

$$e_1 \equiv_E e_2 \Rightarrow a^G(e_1) = a^G(e_2)$$

2. Tow edge are only idetified if thier source and target are identified :

$$e_1 \equiv_E e_1 \Rightarrow s^G(e_1) \equiv s^G(e_2) \wedge t^G(e_1) \equiv t^G(e_2)$$

1.6 Pushout

In the algebraic approach to graph transformation, the concept of pushout is used to describe the glue graphs together along to a common sub-graph.

Given a category $C = (Ob_C, Mor_C, \circ, id) morphisms$, objects $A, B, C \in Ob_C$, and morphisms $f : A \to B$ and $g : A \to C \in Mor_C$, a pushout (D, f', g') over f and g is defined by [16, 6]:

- a pushout object D and
- morphisms $f' : C \to D$ and $g' : B \to D$ with $f' \circ g = g' \circ f$

such that the following universal property is fulfilled: for all objects X with morphisms $h : B \to X$ and $k : C \to X$ with $k \circ f$, there is a unique morphism $x : D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$:

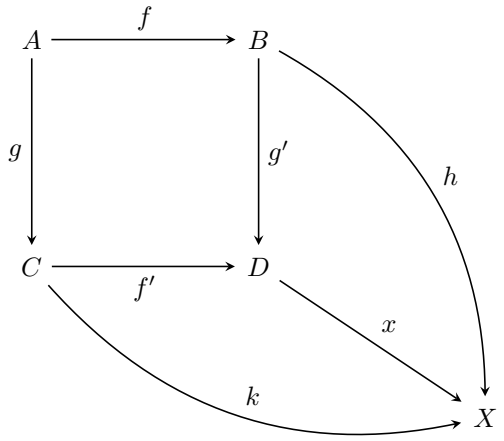


Figure 1.9: Pushout Diagram [16, 6]

It is important to note that the pushout object D is unique up to isomorphism ,and the composition and decomposition of pushouts result again in a pushout.

Example

Given graphs $A = (V_A, E_A, s_A, t_A)$, $B = (V_B, E_B, s_B, t_B)$, $C = (V_C, E_C, s_C, t_C)$. Let the category $Graph = (Ob_{Graph}, Mor_{Graph}, \circ, id)$, objects $A, B, C \in Ob_{Graph}$, and morphisms $f : A \to B$ and $g : A \to C \in Mor_{Graph}$, a pushout (D, f', g') over f and g is defined by the pushout object $D \in Ob_{Graph}$ and morphisms $f' : C \to D$ and $g' : B \to D$ with $f' \circ g = g' \circ f$

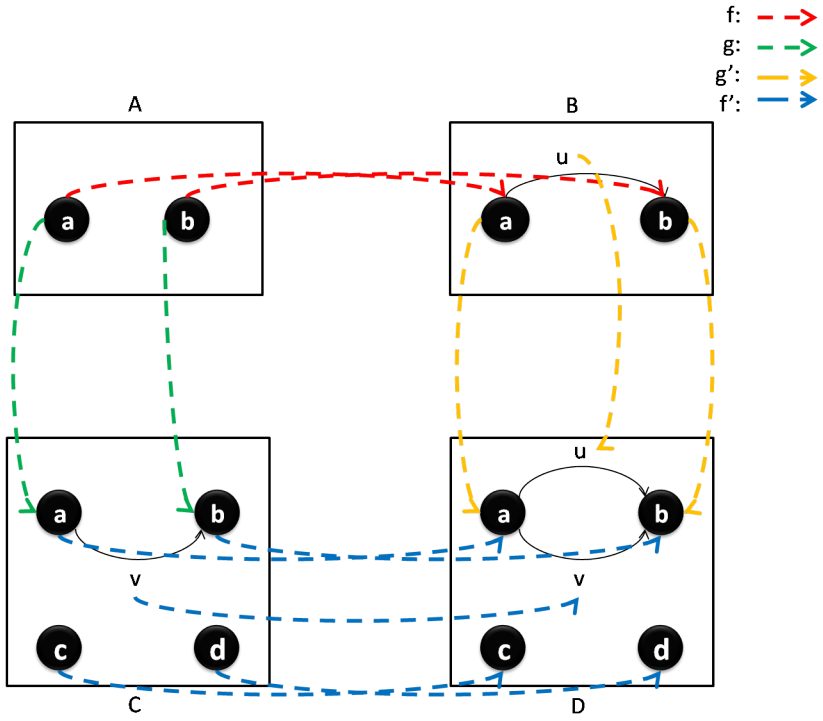


Figure 1.10: Example of pushout

1.7 Graph Production

A (typed) graph production also called rewriting rule (rule, for short) $p = L \xleftarrow{l} K \xrightarrow{r} R$ consists of (typed) graphs L, K, and R, called the left hand side, gluing graph(or interface graph), and the right-hand side respectively, and two (typed) total graph morphisms $l : K \to L$ and $r : K \to R$ [10, 16].

All nodes and edges in L that are not in the image of l are called obsolete. Similarly, all nodes and edges in R that are not in the image of r are called fresh [9].

Example

The rule $r = L \xleftarrow{l} K \xrightarrow{r} R$ (see figure 1.11) consists of:

- The graph $L = (V_L, E_L, s_L, t_L)$, with node set $V_L = \{a, b\}$, edge set $E_L = \{u\}$, source function $s_L : E_L \to V_L : u \to a$ and target function $t_L : E_L \to V_L : u \to b$.
- The graph $R = (V_R, E_R, s_R, t_R)$, with node set $V_R = \{a, b\}$, edge set $E_R = \{v\}$, source function $s_R : E_R \to V_R : v \to a$ and target function $t_R : E_R \to V_R : v \to b$.
- The graph $K = (V_K, E_K, s_K, t_K)$, with node set $V_K = \{a, b\}$, edge set $E_K = \emptyset$.
- The graph morphism $l : K \to L$ is represented by dashed red arrows.
- The graph morphism $r : K \to R$ is represented by dashed blue arrows.

The obsolete element in graph r is the edge u, and the fresh element is the edge v.

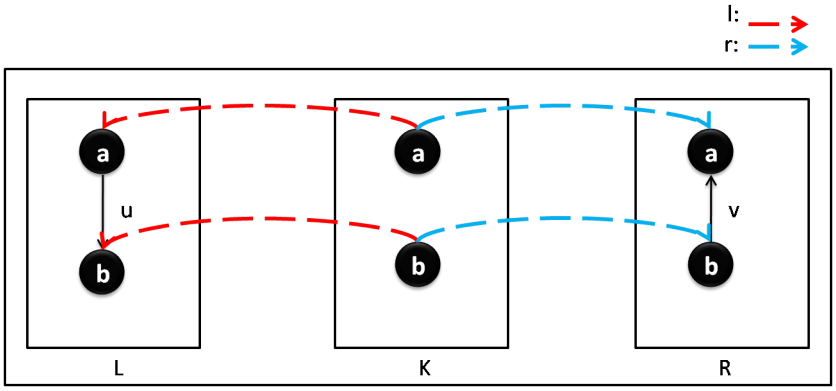


Figure 1.11: Example of Graph Production

1.8 Graph Match

In order to apply a rule $r = L \to R$ on a graph H, we need to identify the match of rule's left side graph L into the graph H. To do so, the match is identified by the (total) graph morphism $m = L \to H$. The match m is used to define nodes and edges of graph H which shall be delete and another one which should be kept after the application of the rule p.

Example We consider the graph H and the graph rule p presented in figure 1.3 and figure 1.11 respectively.

the match of rule's left side graph L into the graph H is identified by the (total) graph morphism $m : L \to H$ which depicted by dashed green arrows in figure 1.12

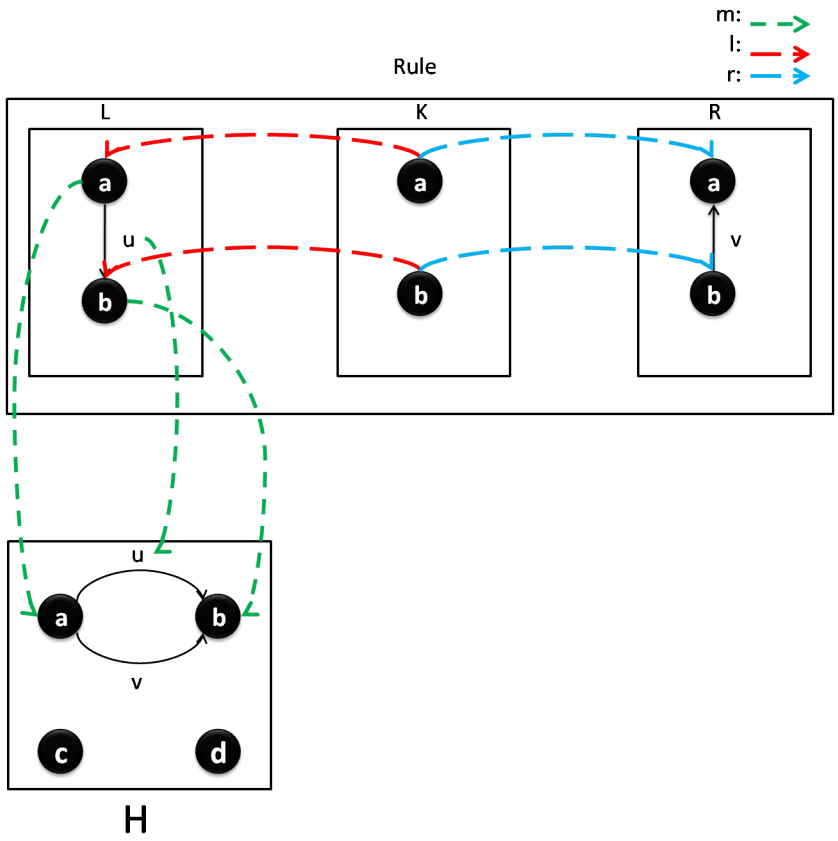


Figure 1.12: Example of Graph Match

There are several techniques to find a graph into another such as constraint satisfaction problem (CSP) [14], Nauty algorithm [11], Backtracking algorithm [18], VF2 algorithm [3]. In our project we use constraint satisfaction problem (CSP) to find the graph maching.

1.8.1 Constraint Satisfaction Problems

A constraint satisfaction problems (CSP) consists of[14]:

- a finite set of variables $X = \{x_1, ..., x_n\}$,
- a finite and discrete domain D_i of possible values for every variable $x_i \in X$, and
- a finite set R of constraints on the variables of X.

In the following we will define some concepts related to CSP.
Constraint [14]:
A constraint C_S on a tuple of variables $S = (x_1, ..., x_r)$ is a relation on these variables' domains: $C_S \subseteq D_1 \times ... \times D_r$.
The number r of variables a constraint is defined upon is called arity of the constraint.
Instantiation of Variables [14]:
Let$X = \{x_1, ..., x_n\}$ be a set of variables with their respective domains $D_i, i \in \{1, ..., n\}$.

Then any n-tuple $\Gamma = (a_1, ..., a_n), a_i \in D_i$ denotes an instantiation of each variable xi with the corresponding value ai. We also write $\Gamma(x_i) = a_i$ for the value of x_i under an instantiation Γ.

Satisfied Constraint [14]:
A constraint CS on a tuple of variables $S = (x_1, ..., x_r)$ is satisfied by an instantiation Γ if $(\Gamma(x_1), ..., \Gamma(x_r)) \in CS$.

Solution of a CSP [14]:
An instantiation Γ is a solution of a CSP if it satisfies all the constraints of the problem.

1.8.2 Graph Matching as a CSP

Given two graphs $L = (L_V, L_E, L_L, s_L, t_L, l_L)$ and $G = (G_V, G_E, L_G, s_G, t_G, l_G)$.

In the following the main steps to obtain an equivalent CSP for a given graph matching problem [14]:

- take the objects of the graph to be matched as the CSP's set of variables:

$$X = L_V \cup L_E = \{x_1, ..., x_n\}, n = |X|$$

- take the objects of the graph to be matched into as the variables' domain:

$$D_i = \begin{cases} G_V, when x_i \in L_V \\ G_E, otherwise \end{cases}, i \in \{1, ..., n\}.$$

- find a proper translation of the restrictions that apply to a graph morphism into a set of constraints:
  The constraint set R is built according to table 1.1 whenever a condition listed in the left column of the table holds for a given pair of variables (x_i, x_k), the corresponding constraint is to be included in R.

$Condition \quad \longleftrightarrow$	$Constraint \in R$
$x_i = x_k$	$C^{type}_{x_i} = \{d \in D_i \mid l_L(x_i) = l_G(d)\}$
$x_i \in L_E, x_k \in L_V, s_L(x_i) = x_k$	$C^{src}_{x_i,x_k} = \{(d_i, d_k) \in D_i \times D_k \mid s_G(d_i) = d_k\}$
$x_i \in L_E, x_k \in L_V, t_L(x_i) = x_k$	$C^{tar}_{x_i,x_k} = \{(d_i, d_k) \in D_i \times D_k \mid s_G(d_i) = d_k\}$

Table 1.1: Conditions for when to establish a constraint between two variables x_i, x_k [14]

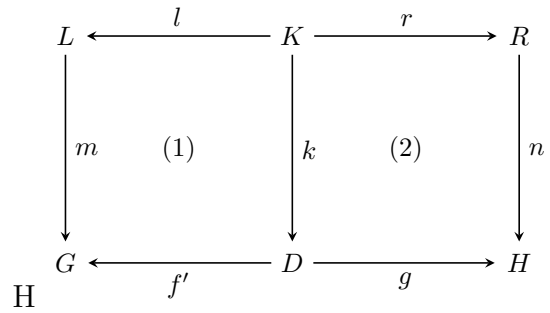


Figure 1.13: double pushout diagram

1.9 Graph Transformation Systems

1.9.1 graph transformation

Given a (typed) graph production $p = L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ and a (typed) graph G with a (typed) graph morphism $m = L \rightarrow G$, called the match, a direct (typed) graph transformation $G \Rightarrow^{p,m} H$ from G to a (typed) graph H is given by the following double-pushout (DPO) diagram, where (1) and (2) are pushouts [16].

A sequence $G_0 \Rightarrow G_1 \Rightarrow ... \Rightarrow G_n$ of direct (typed) graph transformations is called a (typed) graph transformation and is denoted by $G_0 \overset{*}{\Rightarrow} G_n$ For n=0 , we have the identical (typed) graph transformation $G_0 \overset{id}{\Rightarrow} G_0$

1.9.2 graph transformation system

A typed graph transformation system $GST = (TG, P)$ consists of a type graph TG and a set of typed graph productions P.[16]

1.9.3 (Typed) graph grammar

A (typed) graph grammar $GG = (GTS, S)$ consists of GTS and start graph S [16].

The generated language L of GG consists of all graphs G that are derivable from the initial graph S via successive application of the rules in P. The (typed) graph language L of GG is defined by

$$L = \{G \mid \exists (typed) \quad graph \quad transformation \quad system \quad S \Rightarrow^* G\}$$

Example

The figure 1.14 illustrates an example of graph grammar GG=(P,S). This grammar consists of the start graph S and rules P. The start graph consists of a single node x. Rules $p = \{Newnode, Setrelation, Removerelation\}$. The rule $Newnode$ adds new node

x. The rule *Setrelation* generates a new edge y between two existing nodes. The rule *Removerelation* removes an edge y exists between two existing nodes.

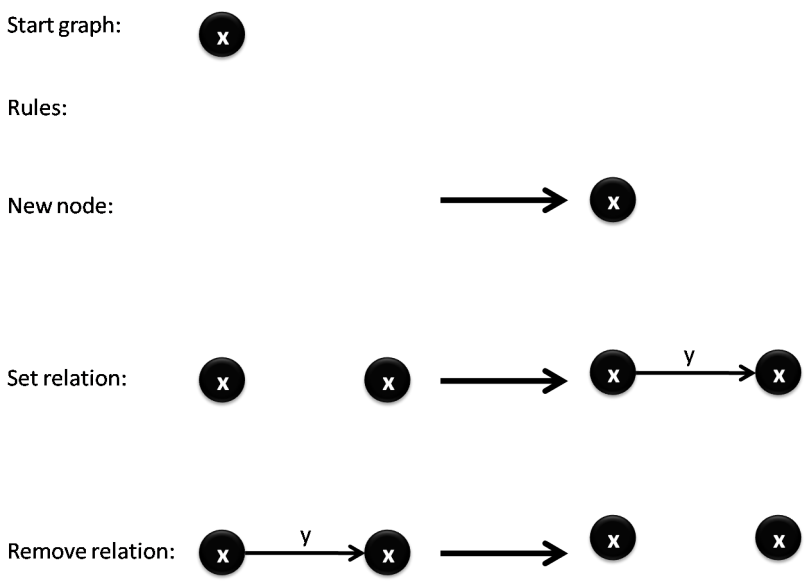


Figure 1.14: Graph grammar example

1.10 Construction of Graph Transformations

1.10.1 Applicability of Productions

A (typed) graph production $p = L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ is applicable to a (typed) graph G via the match m if there exists a context graph D such that (1) is a pushout [16].

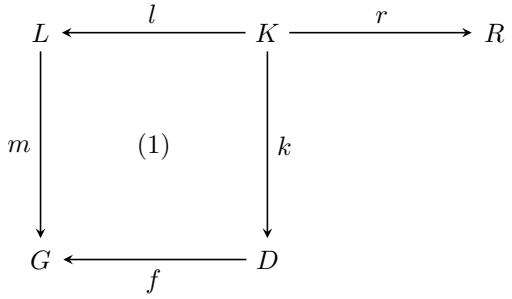


Figure 1.15: Applicability of productions

1.10.2 gluing condition

Given a (typed) graph production $p = L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$,a (typed) graph G, and a match $m : L \rightarrow G$ with X = (V_X, E_X, s_X, t_X) for all $X = \{L, K, R, G\}$,we can state the following definitions[16]:

- The gluing points GP are those nodes and edges in L that are not deleted by p, i.e. $GP = l_V(V_K) \cup l_E(E_K) = l(K)$

- The identification points IP are those nodes and edges in L that are identified by m, i.e. $IP = \{v \in V_L \mid \exists w \in V_L, w \neq v : m_V(v) = m_V(w)\} \cup \{e \in E_L \mid \exists f \in E_L, f \neq e : m_E(e) = m_E(f)\}$
- The dangling points DP are those nodes in L whose images under m are the source or target of an edge in G that does not belong to m(L), i.e. DP = { v $\in V_L$ | ∃ e ∈ $E_G \setminus m_E(E_L)$: s_G $(e) = m_V$ (v) or t_G $(e) = m_V$ (v) } .
  p and m satisfy the gluing condition if all identification points and all dangling points are also gluing points, i.e.$IP \cup DP \subseteq GP$

In is important to note that for a (typed) graph production $p = \left(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R\right)$, a (typed) graph G, and a match $m : L \rightarrow G$, the context graph D with the PO (1) exists if and only if the gluing condition is satisfied.

1.10.3 construction of direct (typed) graph transformations

Given a (typed) graph production $p = L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ and a match $m : L \rightarrow G$ such that p is applicable to a (typed) graph G via m, the direct (typed) graph transformation can be constructed in two steps[6]:

1. Delete those nodes and edges in G that are reached by the match m, but keep the image of K, i.e. $D = (G \setminus m\,(l\,(K)))$. More precisely, construct the context graph D and pushout (1) such that $G = L +_K D$.
2. Add those nodes and edges that are newly created in R, i.e. $H = D \uplus (R \setminus r\,(K))$,where the disjoint union$\uplus$ is used to make sure that we add the elements of $R \setminus r\,(K)$ as new elements. More precisely, construct the pushout (2) of D and R via K such that $H = R +_K D$.

1.11 Application Condition

Application conditions, similarly to the gluing condition, allow us to restrict the application of productions. Now we introduce application conditions for a match $m : L \rightarrow G$, where L is the left-hand side of a (typed) graph production p. The idea is that the production cannot be applied at m if m violates the application condition.

1.11.1 Application Condition

An atomic application condition over a (typed) graph L is of the form $P(x, \wedge_{i \in I} x_i)$ where $x : L \rightarrow G$ and $x_i : X \rightarrow C_i$ with $i \in I$ for some index set I are (typed) graph morphisms.

An application condition over L is a Boolean formula over atomic application conditions over L. This means that true and every atomic application condition are application

conditions, and, for application conditions acc and acc_i with $i \in I$, $\neg acc$, $\wedge_{i\in I} acc_i$, and $\vee_{i\in I} acc_i$ are application conditions.

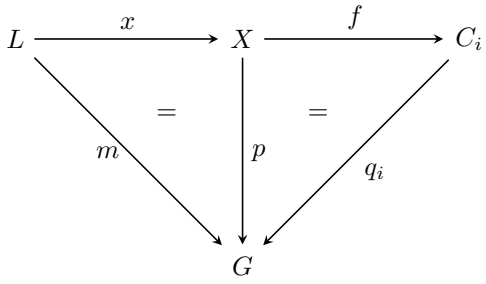


Figure 1.16: Application condition

A (typed) graph morphism $m : L \to G$ satisfies an application condition acc, written $m \vDash acc$, if

- acc = true;
- $acc = P(x, \vee_{i\in I} x_i)$and, for all injective (typed) graph morphisms $p : X \to G$ with $p \circ x = m$ there exists an $i \in I$ and an injective (typed) graph morphism $q_i : C_i \to G$ with $q_i \circ x_i = p$;
- $acc = \neg acc'$ and m does not satisfy acc';
- $acc = \wedge_{i\in I} acc_i$ and m satisfies all acc_i with $i \in I$;
- $acc = \vee_{i\in I} acc_i$ and m satisfies some acc_i with $i \in I$.

1.11.2 Negative Application Condition

A simple negative application condition is of the form $NAC(x)$, where $x : L \to X$ is a (typed) graph morphism . A (typed) graph morphism. $m : L \to G$ satisfies$NAC(x)$ if there does not exist an injective (typed) graph morphism $p : X \to G$ with $p \circ x = m$[16].

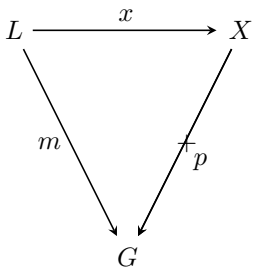


Figure 1.17: Negative Application condition

1.11.3 application condition for a production

Given a (typed) graph production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, and an application condition $A(p)$ for p over L.

A direct (typed) graph transformation $G \xrightarrow{p,m} H$ satisfies the application condition $A(p)$ if $m \vDash A_L$.

Example We consider the graph H and the rule r and the match $m : L \to H$ present in the figure1.12.

Let $NAC(x)$ be a negative application condition for the production r,, the morphism $x : Y \to x$ presented be purple dashed arrows.

The rule r is not applicable into the graph H because exists an injective morphism $p : L \to H$ presented by yellow dashed arrows in the figure 1.18.

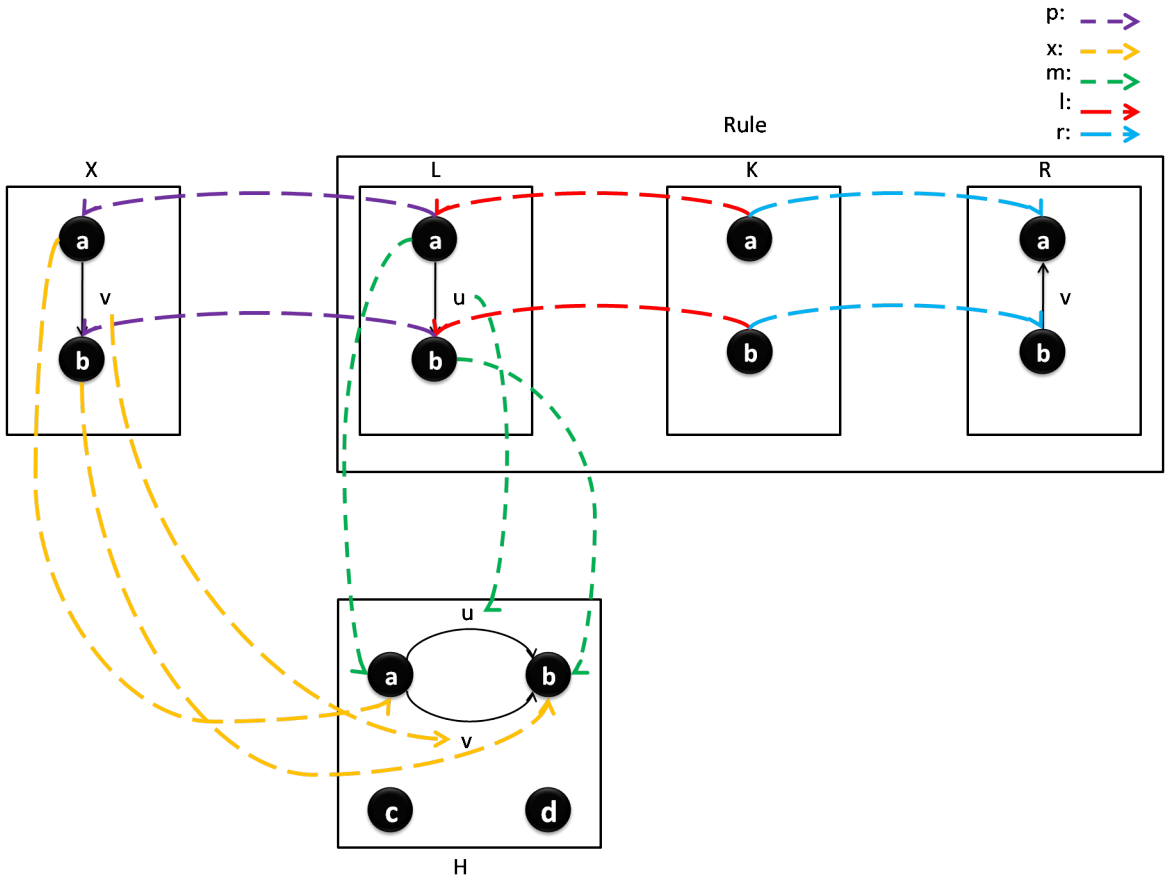


Figure 1.18: Negative application condition for a production example

1.12 Related work

There are many graph transformation tools that are available for many different purposes. In this section, we describe some of them briefly.

1.12.1 Fujaba

The Fujaba [8] [12] environment aims to provide round-trip engineering support for UML and Java. The main distinction to other UML tools is its tight integration of UML class and behavior diagrams to a visual programming language.

1.12.2 AGG

AGG [17]is a general development environment for algebraic graph transformation systems. Its special power comes from a very flexible attribution concept. AGG graphs are allowed to be attributed by any kind of Java objects.

1.12.3 VIATRA

The VIATRA (VIsual Automated model TRAnsformations) [5] framework is a transformation-based verification and validation environment for improving the quality of systems designed within the Unified Modeling Language by automatically checking consistency, completeness, and dependability requirement.

1.13 Conclusion

In this chapter, we defined the main concepts related to graph transformation used in our project such as the concept of category, graphs, typed graph and, graph production.

In the next chapter, we will present the design of our system. We will see how these theoretical concepts are transformed into components and modules.

Chapter 2

System Design

2.1 Introduction

The objective of this project is the creation of a graph transformation engine by implementing the Double-Pushout approach which is one of the most well-known approaches to graph transformation.

In the current chapter, we present the design of our engine. In the first section, we provide an overall view of our project, in which we give the general architecture as a diagram of the different inter-connected element of the system. In the second section, we present the detailed design of our engine. We start by modeling the static part by the class diagram. Then, we describe the dynamic part by using the activity diagram. Finally, we present the most important algorithms we implemented in our system.

2.2 Global Design

As illustrated in figure 2.1, our system can be seen as two main components. These latter allow as to cover the modeling part represented by the module Graph Grammar and the action of transformation part represented by the module Graph Transformation.

In the following, we explain each part of our system, its utility, and its provided functions.

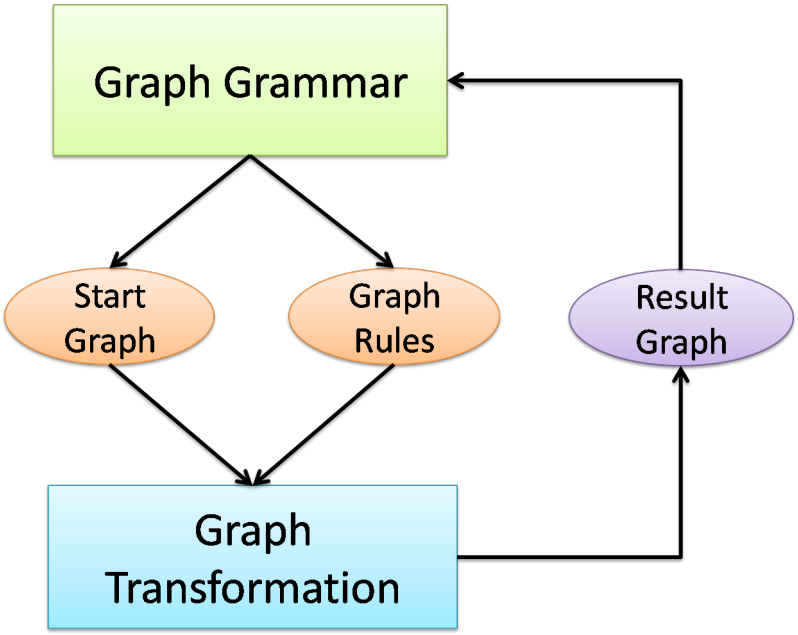


Figure 2.1: Global architecture of the application

2.2.1 Graph Grammar

This module is used to model a graph grammars, this later can be used to generate graph languages similar to Chomsky grammars in formal language theory. a graph grammar consists of a type graph, a start graph and a set of graph rules. A graph is used to model the states of systems. A rule is used to describe the way how to transform graphs. The start graph and Graph rules are taken as input in the next step.

Hereinafter, we will illustrate in figure 2.2 the various functions offered by this module using "UML use case diagram". These functions are used to create and manipulate graph grammars.

The important actions offered by this module are:

- **Create graph grammar**: this action creates new graph grammar;
- **Create type graph**: this action creates new type graph;
- **Create graph**: this action creates new typed graph;
- **Create rule**: this action creates new rule;
- **Add rule**: this action adds to the graph grammar a new rule;
- **Add graph**: this action adds to the graph grammar a new typed graph;
- **Remove graph**: this action removes a specific graph from the listed graphs;
- **Remove rule**: this action removes a specific rule from the listed rules;

- **Select graph**: this action chooses a graph from the listed graphs;
- **Select rule**: this action chooses a rule from the listed rules;
- **Check the type graph level**: this action checks if the level of the type graph is respected or not by a typed graph.

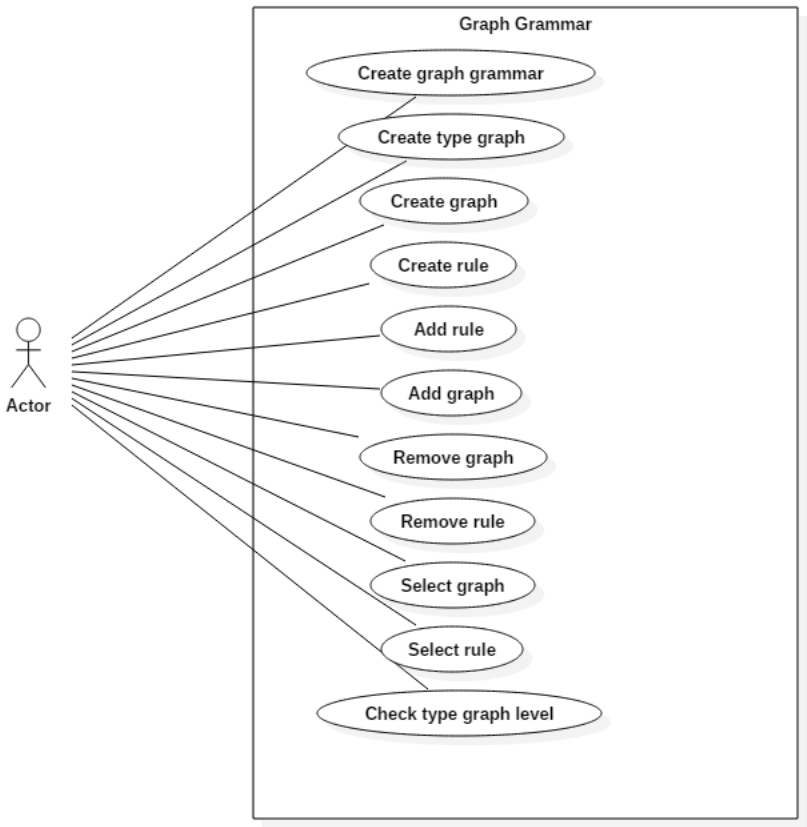


Figure 2.2: Graph grammar use case diagram

2.2.2 Graph Transformation

Graph transformation is used to model state changes in systems. This module presents the action of transformation of graphs. It takes as input a start graph and a set of rules and generates the resulting graph from the start graph. This transformation is based on the application of The set of rules, that describes the transformation of elements from the start graph to elements in the resulting graph. The application of a single rule is called a direct graph transformation and the application of a set of rules is a sequence of direct graph transformation, and it called graph transformation as we see in chapter 1 section 1.9.1. The application of a single rule or a set of rules leads to the resulting graph, These later is taken as input in graph grammar.

Hereinafter, we illustrate in figure 2.3 the various functions offered by this module using “UML use case diagram“.

Graph transformation is the application of a set of rules on a graph. It has two kinds, the first is graph transformation non-deterministic that chosen randomly rules to be applied, and the second is graph transformation by rules priority that chosen by priority the rules to be applied. It includes direct graph transformation that is an application of one rule on a graph by applying the double pushout.

The important actions provided by this module are:

- **Direct graph transformation**: this action applies one rule on a graph by applying the double pushout.
- **Graph transformation by priority**: this action applies a set of rule. The rule with the elevated priority is applied first.
- **Graph transformation non-deterministic**: this action applies a set of rules. It chooses randomly the rule to be applied.

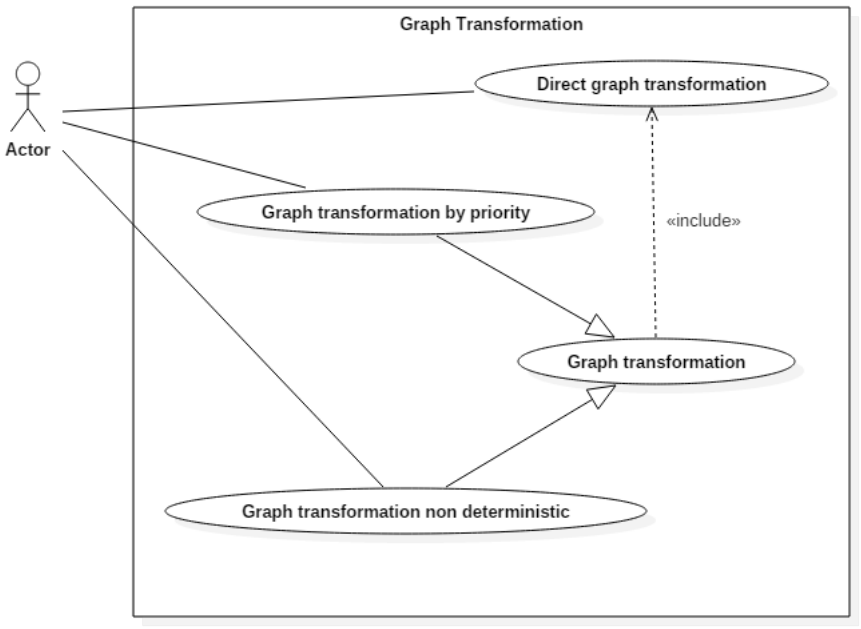


Figure 2.3: Graph transformation use case diagram

2.3 Detailed Design

In this section, we specify the important part of our engine. To do so, we have used UML (Unified Modeling Language) class diagram that shows the structural components of the graph grammar. And we have also used activity diagrams that show the dynamic aspect of graph transformation. We also present the various algorithms that we have to be implemented in order to release our engine.

2.3.1 Graph Grammar Class Diagram

The figure 2.4 shows graph grammar class diagram. In this diagram, most of the classes have a unique identifier and name. In the following we will explain each class:

- **GraphGrammar**: this class is the main class in our diagram. It contains an array of rules, a start graph that is an instance of TypedGraph class, and finally, a type graph.
- **TypedGraph**: this class is an important class, each graph is an instance of this class. It contains tow arrays the first of nodes and the second of arcs, that tow lasts are an instance of classes TypedNode and TypedArc respectively. Any instance of this class must conform to the type graph.
- **TypedNode**: this class has an attribute called type that is an instance of the class TypeNode. The instance of this class represents a node in a typed graph.
- **TypeArc**: this class has the type that is an instance of the class TypeArc. And it has also the attributes source and destination which are instances of the class TypedNode. the instance of this class represents an arc in a typed graph.
- **TypeGraph**: This class contains tow arrays the first of nodes and the second of arcs, that tow lasts are instances of classes TypeGraphNode and TypeGraphArc respectively. The current class has the attribute types that is an instance of the class TypeSet. The last attributes represent a set of types, which can be used to assign a type to the nodes and edges of a graph. This class has also the attribute typeGraphLevel that used to check if the type graph is respected by typed graphs, this attribute can take the following values:
  - *Disabled*: means disable the type graph.
  - *Enabled*: means enable type graph and no multiplicity.
  - *EnabledMin*: means enable type graph and min multiplicity.
  - *EnabledMax*: means enable type graph and max multiplicity.
  - *EnabledMaxMin*: means enable type graph and max and min multiplicity.
- **TypeGraphNode**: this class has an attribute called type that is an instance of the class TypeNode, and It has also the attribute multiplicity that is an instance of the class Multiplicity. An instance of this class represents a node in a type graph.
- **TypeGraphArc**: this class has an attribute called type that is an instance of the class TypeArc. And it has attributes source and destination which are instances of the class TypeGraphNode. And it has also the attributes sourceMult and destination that are instances of the class Multiplicity. The attribute sourceMult represent

the number of nodes, those nodes have a specific type and they are allowed to be the source of an arc has a specific type. And the attributes destinationMult represent the number of nodes, those nodes have a specific type and they are allowed to be the destination of an arc has a specific type. An instance of this class represents an arc in a type graph.

- **Multiplicity**: this class has two integer attributes called min, max. those attributes represent the minimal and the maximal values of the multiplicity

- **TypeSet**: this class contains two arrays TypeNodeSet and TypeArcSet that are instances of classes TypeNode and TypeArc respectively. An instance of this class represents a set of types defined by a type graph, the set of types used to assign a type to graph nodes and edge. The typing itself is done by a graph morphism between the graph and the type graph (see section 1.3.3).

- **TypeArc**: The class has a unique identifier a name, it used to represent the type of arcs.

- **TypeNode**: The class has a unique identifier, a name. it is used to represent the type of nodes.

- **Rule**: this class has an attribute called priority, and it contains LHS, RHS, and morphism. LHS and RHS are instances of the class TypedGraph but morphism is an instance of the class Morphism. LHS and RHS represent the right and the left graph of the rule respectively.

- **Morphism**: this class is used to represents the morphism between two graphs. This class has attributes domain and co-domain that are instances of the class TypedGraph, And it contains two arrays nodeMapSet and ArcMapSet that are instances of the classes ModeMap and ArcMap respectively.

- **NodeMap**: this class is used to represents the mapping between two nodes, this class has attributes called original and image that are instances of the class TypedNode.

- **ArcMap**: this class is used to represents the mapping between two arcs, this class has attributes called original and image that are instances of the class TypedArc.

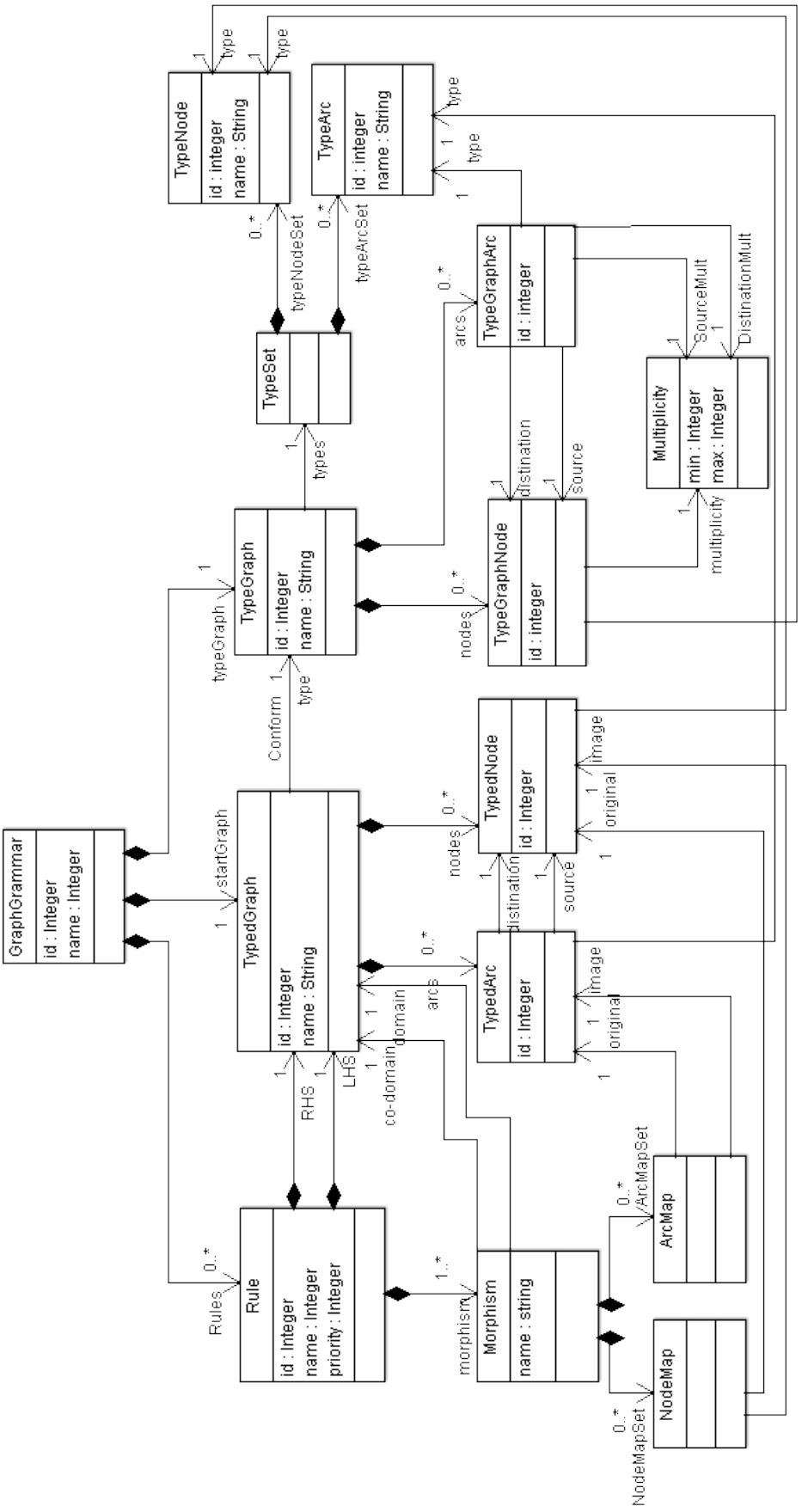


Figure 2.4: Graph grammar class diagram

2.3.2 Direct graph transformation activity diagram

The figure 2.5 shows the direct graph transformation activity diagram.That is the application of a rule r=(L,R) on a graph G.

We start by checking the applicability of rule on the host graph. Than if the rule is applicable, we apply the double pushout, else we stop.

1. find all matches of rule's left graph L in the host graph G.
2. check if rule has matches in the graph G,
3. If no more match go to the step7, else go to the next step.
4. choose randomly one match to apply the rule.
5. Check the application condition of a rule into the chosen match. If It is satisfied go to the next step, else go to the step 2.
6. Apply the rule r into the host graph G. then go to the step 7
7. Finish.

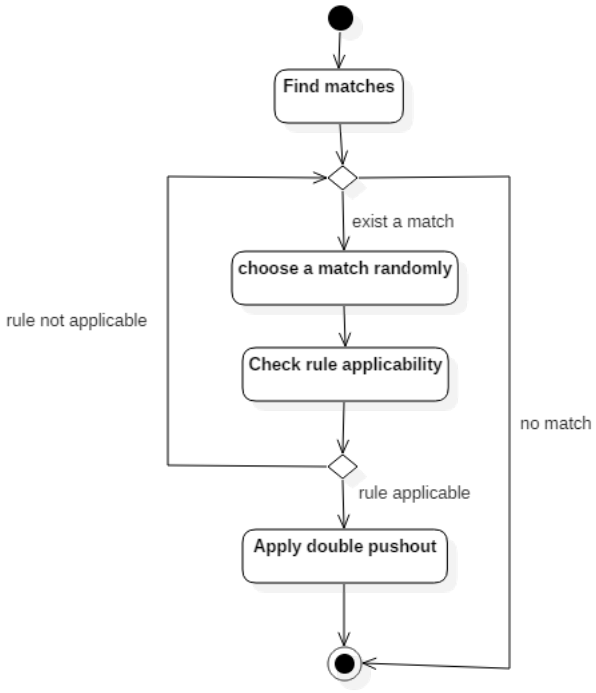


Figure 2.5: Direct graph transformation activity diagram

2.3.3 Non-deterministic Graph transformation activity diagram

he figure 2.6 shows the activity diagram of non-deterministic Graph transformation.

The non deterministic graph transformation is a kind of graph transformation based on choosing randomly the rule which we will be applied. In the following the main steps for applying the non-deterministic graph transformation:

1. check if the rule set is not empty;
2. if the rule set is empty, we stop.
3. choose randomly from the set of rules the rule that will be applied;
4. apply the direct graph transformation by applying the selected rule on a graph;
5. if the the direct graph transformation was not applied than remove the chosen rule from the set of rule the go to the step 1;
6. if the direct graph transformation applied then reset all rules by restoring all the remove rules and go to the step 3.

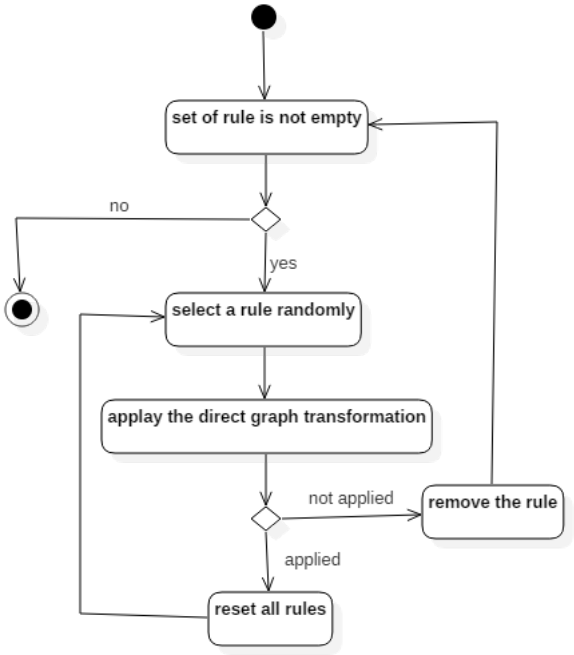


Figure 2.6: Non-deterministic Graph transformation activity diagram

2.3.4 Graph transformation by rule priority activity diagram

The figure 2.7 shows the graph transformation by rule priority activity diagram.

Graph transformation by rule priority is a kind of graph transformation we apply the rule with the elevated priority first, such that when rules have the same priority we choose randomly the rule to be applied.

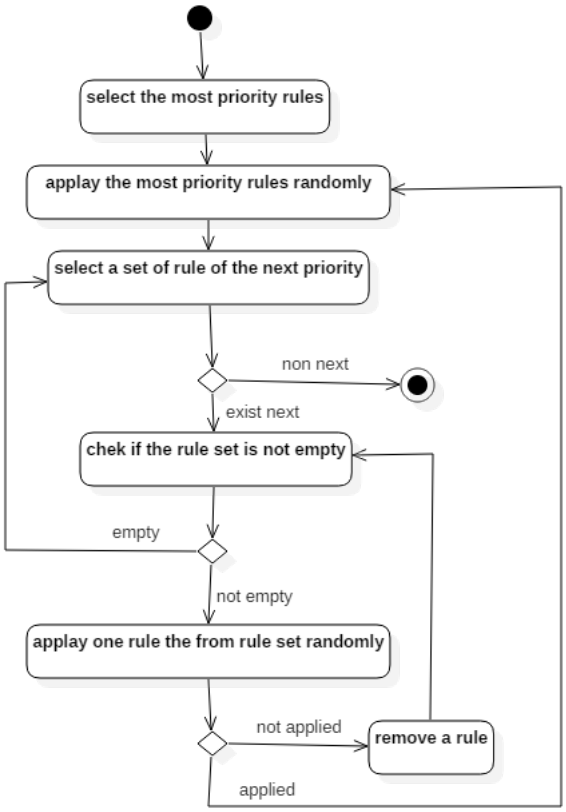


Figure 2.7: Graph transformation by rile priority activity diagram

2.3.5 Find matches algorithm

To find the graph match (see 1.8) of a rule' left graph in a host graph we use a technique called constraint satisfaction problems (see 1.8.1). To do so, we follow the next steps:

1. tack nodes and arcs of the left graph of a rule as to disjoint domains.
2. tack element of the host graph as variables and assign to each variable its suitable domain.
3. tack constrains that used to restrict
4. gets a set of constraint correspond to variable.
5. solve the CSP to find the list of all matches.

Algorithm 1 Find matches algorithm

1: \\ GN is a Domain which contains all nodes of LHS
2: GN ← Domain nodes(rule's left graph)
3: \\ GE is a Domain which contains all arcs of LHS
4: GE ← Domain Arcs(rule's left graph)
5: \\ vars is an array of variables
6: vars ← Variables(host Graph, GN, GE)
7: constraints ← Constraints(vars)
8: \\ is an array which contains the matches of the LHS into the host graph.
9: $solutions = Solve(vars, constraints)$

2.3.6 Check the applicability of rule algorithm

According to figure 2.5, checking the applicability of a rule is necessary to apply the double pushout, to say that a rule is applicable it must satisfy two application condition. the first condition is the dandling condition, and the second is the negative application condition.

The algorithm 2 shows Check rule applicability algorithm. that returns true if the specified rule is satisfied the dangling 1.10.2 and the negative application conditions 1.11.2 at the specified graph by the specified matching, returns false otherwise.

Algorithm 2 Check rule applicability algorithm

1: \\ IsDandlingConditionSatisfied returns true if the dangling condition is satisfied, returns false otherwise
2: isDandlingSatisfied←IsDandlingConditionSatisfied(rule, hostGraph, match)
3: \\ IsNACsSatisfied returns true if all negative application conditions are satisfied, returns false otherwise
4: is NACsSatisfied←IsNACsSatisfied(rule, hostGraph ,match)
5: **if** isDandlingSatisfied ∧ NACsSatisfied **then**
6: **return** true
7: **end if**
8: **return** false

2.3.7 Double Pushout algorithm

Applying a double pushout consists of applying two pushouts which are pushout1 and pushout2. The pushout1 is used to construct the context graph by deleting the absolute elements(the absolute elements are the elements exist in the left graph of a rule that do not exist in the right graph of this rule). The pushout2 is used to construct the resulting graph by adding the fresh elements (the fresh elements are the elements exist in the right graph of a rule that do not exist in the left graph of this rule) to the context graph.

Algorithm 3 Double pushout algorithm

```
1: function ApplyDoublePushout (rule, graph, match)
2:     contextGraph ← ApplyPushout1(rule,graph,match)
3:     if contextGraph ≠ nil then
4:         resultingGraph ← ApplyPushout2(rule,contextGraph)
5:         return resultingGraph
6:     end if
7:     return nil
8: end function
```

2.3.8 Conform Algorithm

In the algorithm 4, we verify if typed graph conforms the type graph according to the type graph level, If the type graph level respected return true, return false otherwise.

Algorithm 4 Conform Algorithm

```
1: if type Graph Level respected then
2:     return true
3: end if
4: return false
```

2.4 Conclusion

In this chapter, we presented the design of our engine. In first section, we have described the global architecture. In second section, we have explained the detailed design. We use class diagram to model structure of graph grammar and activity diagrams to model direct graph transformation, non-deterministic graph transformation, and graph transformation by rule priority. Then we present the main implemented algorithms.

Chapter 3

Implementation

3.1 Introduction

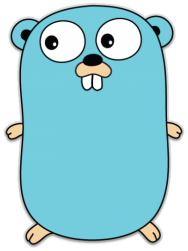
After the design step that is mentioned in the previous chapter, we pass to the next step, which is the step of implementation.

This chapter includes two sections. The first section introduces briefly the development tools and languages that we have used them in the realization of our engine. The second section aims to implement the algorithms that we have studied in chapter 2 and explain the APIs provided by our engine.

3.2 Development Tools and Languages

In this section, we present different tools and languages, that help us during the development of our engine.

3.2.1 Go programming language



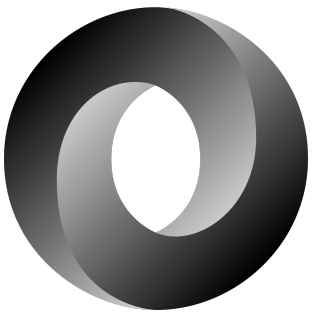
Go (also called GoLang) is programming language developed by Google engineers that we have used it in the implementation of our system. It is easy to learn and easy to use. It is easy to learn, because it is flexible, and its syntax is not hard to learn. Go is an open

source and typed programming language. It is available for all these operating system (Windows, LINUX, Mac OS).

3.2.2 Visual Studio Code

Visual Studio Code is a lightweight but powerful source code editor developed by Microsoft. It has a rich ecosystem of extensions for many languages (such as C++, C, Java, Python, PHP, Go). It is available for all these operating system (Windows, LINUX, Mac OS).

3.2.3 JavaScript Object Notation



JSON (JavaScript Object Notation) is a data-interchange format. It is easy to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language.

3.3 Implementation

In this section, we will present the main data structure that our engine is defined, then we will present the most important APIs that our engine provide, and explain how they are used.

3.3.1 Data Structure

In the following we list the main data structure that our engine is defined.

- **Graph grammar**: According to the listing1 a graph grammar is defined by:
  - **id**: is the identifier of graph grammar.
  - **name**: is the name of graph grammar.
  - **typeGraph**: is the type graph, all graphs exists in the graph grammar must be conform to the type graph.
  - **graphs**: is the full set of graphs exists in graph grammar.
  - **rules**: is the full set of rules exists in graph grammar.

Listing 1 Graph Grammar data structure

```
type grammar struct {
        id        int64
        name      string
    typeGraph graph.TypeGraph
        graphs  map[int64]graph.TypedGraph
        rules     map[int64]rule.Rule
}
```

- **Type graph**: According to the listing2 a type graph is defined by:
  - **id**: is the identifier of a type graph.
  - **name**: is the name of a type graph.
  - **typeGraphLevel**: is the level of a type graph, it can be one of the following constants values:
    - **Disabled**: disable type graph.
    - **Enabled**: enable type graph and no multiplicity.
    - **EnabledMin**: enable type graph and min multiplicity.
    - **EnabledMax**: enable type graph and max multiplicity.
    - **EnabledMaxMin**: enable type graph and max and min multiplicity.
  - **types**: is a types set for nodes and arcs.
  - **nodes**: is a set of nodes, each node has a specific type exists only one time.
  - **from**: associates to each node the set of its incoming arcs.
  - **to**: associates to each node the set of its outgoing arcs.

Listing 2 Type graph data structure

```
        type typeGraph struct {
        id              int64
        name            string
        typeGraphLevel int
        types           TypeSet
        nodes           map[int64]TypeGraphNode
        from            map[int64]map[int64]TypeGraphArc
        to              map[int64]map[int64]TypeGraphArc
}
```

- **Typed graph**: According to the listing3 a type graph is defined by:
  - **id**: is the identifier of a typed graph.
  - **name**: is the name of a typed graph.
  - **kind**: is the kind of a typed graph, it can be Host, LHS, RHS, or NAC.
  - **nodes**: is a set of nodes of a typed graph.
  - **from**: associates to each node the set of its incoming arcs.
  - **to**: associates to each node the set of its outgoing arcs.
  - **itstype**: is the type graph that must be respected during the construction of a typed graph.

Listing 3 Typed graph data structure

```
type typedGraph struct {
        id      int64
        name    string
        kind    int
        nodes   map[int64]Node
        from    map[int64]map[int64]Arc
        to      map[int64]map[int64]Arc
        itstype TypeGraph
}
```

- **Rule**: According to the listing4a rule is defined by:
  - **id**: is the identifier of a rule.
  - **name**: is the name of rule.
  - **lhs**: is the left side graph of a rule.
  - **rhs**: is the right side graph of a rule.

- **ruleMorphism**: is morphism from the left side graph into the right side graph of a rule.
- **nacs**: is the set of negative application conditions of a rule.
- **priority**: is the priority of a rule.

Listing 4 Rule data structure

```
type rule struct {
        id           int64
        name         string
        lhs          graph.TypedGraph
        rhs          graph.TypedGraph
        ruleMorphism morphism.Morphism
        nacs         []morphism.Morphism
        priority     int
}
```

- **Morphism**: According to the listing5 a morphism is defined by:
  - **name**: is the name of a morphism.
  - **domain**: is the domain of a morphism.
  - **codomain**: is the codomain of a morphism.
  - **nodeMapSet**: is the set of node map of a morphism.
  - **arcMapSet**: is the set arc map of a morphism.

Listing 5 Morphism data structure

```
type morphism struct {
        name       string
        domain     graph.TypedGraph
        codomain     graph.TypedGraph
        nodeMapSet map[int64]NodeMap
        arcMapSet  map[int64]ArcMap
}
```

3.3.2 Functions

In this following, we will view how the algorithms presented in chapter 2 have been implemented in our engine according to the Go programming language.

Double Push Out

The listing 6 shows the function DoublePushOut. The visibility of this function is public. It starts by calling the function PushOut1 to construct the context graph, if PushOut1 returns a nil err which means that the context graph has been constructed, then it passes to call the function PushOut2 to construct the target graph, and then it returns nil.

Listing 6 Double Pushout Function

```
func (d *doublePushOut) DoublePushOut() error {
        err := d.PushOut1()
        if err != nil {
            d.PushOut2()
                return nil
        }
        return err
}
```

Conform

The listing 7 present the function IsConform, it used to check if a specific typed graph is conform to its type graph. this function returns nil if the level of type graph is disabled, else it passes to check if the type graph structure is satisfied, if not then it returns an error message, else if the type graph level is enabled then it returns nil, else it check if type graph multiplicity is satisfied, if yes it returns nil, else it returns error message.

Listing 7 Conform Function

```
func (g *typedGraph) IsConform() error {
        if g.Type().TypeGraphLevel() == Disabled {
        return nil
        }
        err := IsTypeGraphStructureStisfied()
        if err == nil{
                if g.Type().TypeGraphLevel() > Enabled {
                        err = IsTypeGraphMultiplicitySatisfied()
                }
        }
        return err
}
```

3.3.3 Usage of the APIs

In the following, we list some APIs that our engine provide and explain how they are used.

- **Func IsValidMatch**

  ```
  func (m *match) IsValidMatch() bool
  ```

  it is Called by a match. It is used to check if this match is valid (checking if the gluing condition and the full set of negative application conditions are satisfied). It has no parameters. It returns true if this match is a valid match, otherwise, it returns false.

  Example

  – Code:

Listing 8 Usage of function IsValidMatch exemple

```
xmlReader := xmlfile.NewReader()
grammar := xmlReader.Read("Lovers_Graph.ggx")
graph := grammar.Graph(7)
rule := grammar.Rule(27)
matches := match.NewMatches(rule, graph)
if matches.FindMatches() != 0 {
    match := matches.ChooseMatch()
    if match != nil {
            if match.IsValidMatch() {
                    println("This match is valid")
                    println(match)
                    println("This is not a valid match")
            }
    }
}
if matches.PossibleMatches() == 0 {
    println("No valid match")
}
```

  – Output: This match is valid

- IsConform:

  ```
  func (g *typedGraph) IsConform() bool
  ```

  This function is called to check if it is conform to the type graph.

  Example

  – Code:

Listing 9 Usage of function Conform exemple

```
xmlReader := xmlfile.NewReader()
grammar := xmlReader.Read("Lovers_Graph.ggx")
graph := grammar.Graph(7)
err:=graph.IsConform()
if err == nil{
        println("This graph is conform to the type graph")
}
println(err)
```

– Output: This graph is conform to the type graph.

3.4 Conclusion

In this chapter, we have presented the development tools and languages that we have used to develop our engine. We have also explained the different data structures and functions defined in our engine, then, we give some illustrative example to explain the usage of the important APIs provided by our engine.

Chapter 4

Use Case: Implementing A Graph Transformation Tool

4.1 Introduction

In this chapter, we demonstrate the usability and the utility of our engine. To do so, we will create an application that uses and exploits the various packages and APIs (Application Programming Interface) that we have defined. Our application is designed following the concept of REST (REpresentational State Transfer) architecture. We start this chapter by an overview of REST architecture, and then we illustrate the design of our application in two levels (global and detailed design).

4.2 REST architecture

REST is an architectural style defined to help create and organize distributed systems. The term representational state transfer was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation [7].

The specific constraints that compose the REST style are [7]:

- **Client-Server:**
  The first constraints is the client-server architectural style. A server component, offering a set of services, listens for requests upon those services. A client component, desiring that a service be performed, sends a request to the server via a connector. The server either rejects or performs the request and sends a response back to the client.

  The main principle behind this constraint is the separation of concerns.

- **Stateless**:
  Communication between client and server must be stateless, meaning that each

request done from the client must have all the information required for the server to understand it, without taking advantage of any stored data.

- **Cache-able**:
  Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

  The advantage of adding cache constraints is that they have the potential to partially or completely eliminate some interactions, improving efficiency, scalability, and userperceived performance by reducing the average latency of a series of interactions. The trade-off, however, is that a cache can decrease reliability if stale data within the cache differs significantly from the data that would have been obtained had the request been sent directly to the server.

- **Uniform Interface**:
  One of REST's main characteristics and winning points when compared to other alternatives is the uniform interface constraint. By keeping a uniform interface between components, you simplify the job of the client when it comes to interacting with your system. Another major winning point here is that the client's implementation is independent of yours, so by defining a standard and uniform interface for all of your services, you effectively simplified the implementation of independent clients by giving them a clear set of rules to follow.

- **Layered System**:
  REST was designed with the Internet in mind, which means that an architecture that follows REST is expected to work properly with the massive amount of traffic that exists in the web of webs.

  In order to achieve this, the concept of layers is introduced. By separating components into layers, and allowing each layer to only use the one below and to communicate its output to the one above, you simplify the system's overall complexity and keep component coupling in check. This is a great benefit in all type of systems, especially when the complexity of such a system is ever-growing (e.g., systems with massive amounts of clients, systems that are currently evolving, etc.).

  The main disadvantage of this constraint is that for small systems, it might add unwanted latency into the overall data flow, due to the different interactions between layers.

- **Code-on-Demand**:
  Code-on-demand is the only optional constraint imposed by REST, which means

that an architect using REST can choose whether or not to use this constraint, and either gains its advantages or suffers its disadvantages. With this constraint, the client can download and execute code provided by the server (such as Java applets, JavaScript scripts, etc.). In the case of REST APIs (which is what this book focuses on), this constraint seems unnecessary, because the normal thing for an API client to do is just get information from an endpoint, and then process it however needed; but for other uses of REST, like web servers, a client (i.e., a browser) will probably benefit from this constraint.

After this overview of REST, in the following sections, we present the different aspects and design of our application. We will start with the global design in which we specify the different parts of our application and discuss the various relations between them. Then we present the detailed design in which we explain each part in detail.

4.3 Global Design

The general architecture of our application is depicted by the Figure 4.1.

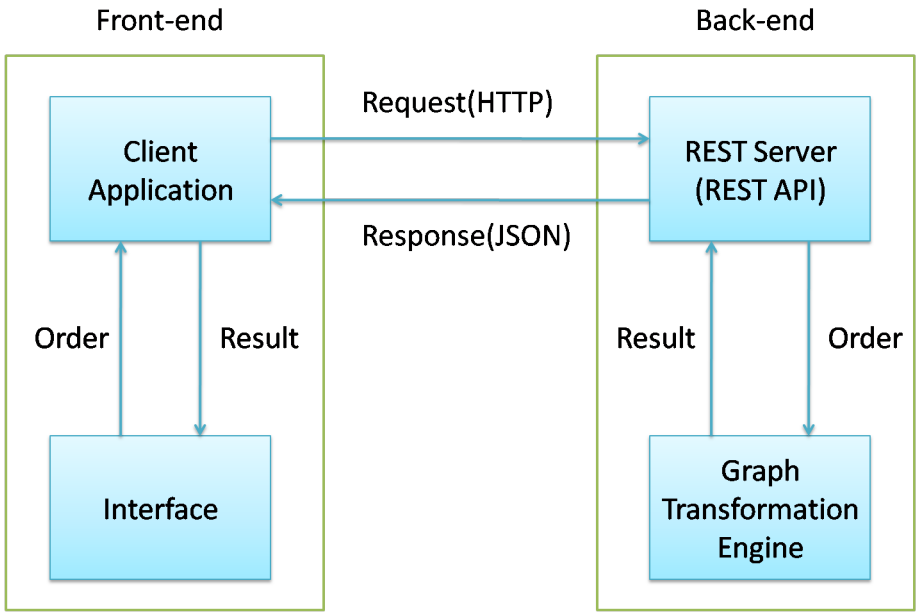


Figure 4.1: Graph transformation tools global architecture

Our application consists of two parts interacting between them through http protocol. These two parts are: the front-end and the back-end.

4.3.1 Front-end

The front-end represents the client-side, it consists of two components: the interface and the client.

1. **Interface:** The interface manage the different interactions between the user and the client application. It displays the graph grammar and its different components, and it tacks the orders from the user and transfer them to the client application, and display the results offered by the client to the user.

2. **Client:** The client plays the role of mediator between the rest server and the interface. In one hand, it tacks orders offered by the interface, then sends it to the server using the various URIs offered by the Rest server, In the other hand, it receives the responses from the server which will be displayed by the interface.

4.3.2 Back-end

The back-end represents the server-side, it consists of two components: the Rest server and the Engine.

1. **Rest server:** The REST server implements the server part of the REST architecture. It exposes a set of function that we defined in our engine as a set of URI to the client part. It receives requests from the client application, and then, it analyses this URI to identify the corresponding function in the engine and extract a set of parameters to invoke the identified API. After that, it returns the result offered by the engine to the client application in the form of JSON data. we can say that the rest server manages the transport part between the client and the graph transformation engine.

2. **Engine:** As we explained in a detailed way in chapters 2 and 3, our engine exposes a set of APIs that can be used to manage the different aspect of graph transformation. these APIs are also exposed themselves by the REST server to the client through the different URIs.

4.4 Detailed Design

Our application has several components and different kind of interaction. In this section, we will detail the most important ones.

As we did in chapter 2, We use the UML (Unified Modeling Language) to describe the design of our application. specifically, we use a sequence diagram that shows the convenient interactions between elements of our application.

4.4.1 The sequence diagram of Single rule application

The interaction scenario, depicted in figure 4.2 is explained as follows:

1. The client invokes the Rest server by sending the identifier of a rule which will be applied, and of a graph which will be transformed by this rule.
2. The server invokes the engine by sending the corresponding rule and graph to the received identifier.
3. The engine check the applicability of the rule.
4. Making a test: if the specific rule is not applicable the engine returns an error to the server, and the server in his turn returns an error to the client.
5. else the engine returns the resulting graph to the server. And the server in his turn returns the resulting graph as JSON file.

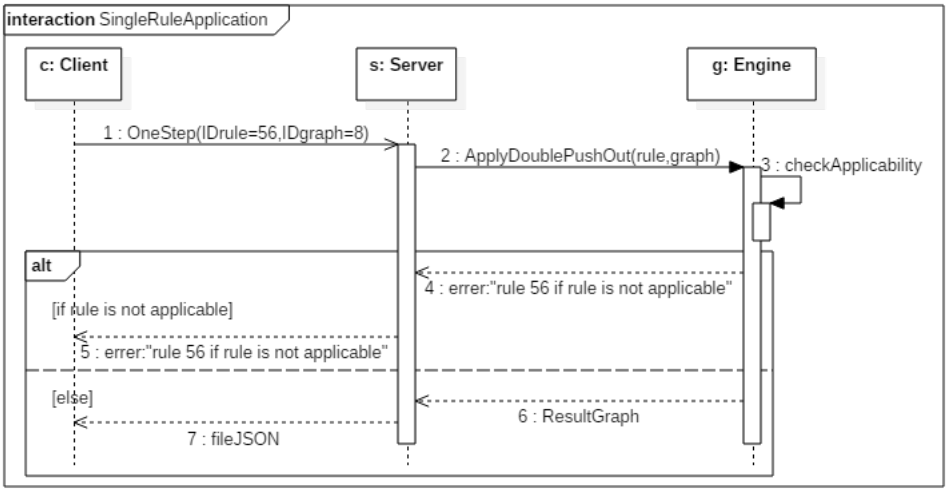


Figure 4.2: Sequence diagram of Single rule application

4.4.2 The sequence diagram of rule set application

The interaction scenario, depicts in figure 4.3 is explained as follows:

- the client invokes the Rest server by sending the identifier of the graph which will be transformed.
- the server selects the corresponding graph and sends it to Engine.
- while the time is not out and there is a rule applicable the Engine select a rule then applies this rule on our graph. it applies this the selected rule as long as it is applicable.
- if there no rule applicable the Engine returns an error message which explains the problem, else it returns the resulting graph to the server,

- then the server returns it as JSON file.

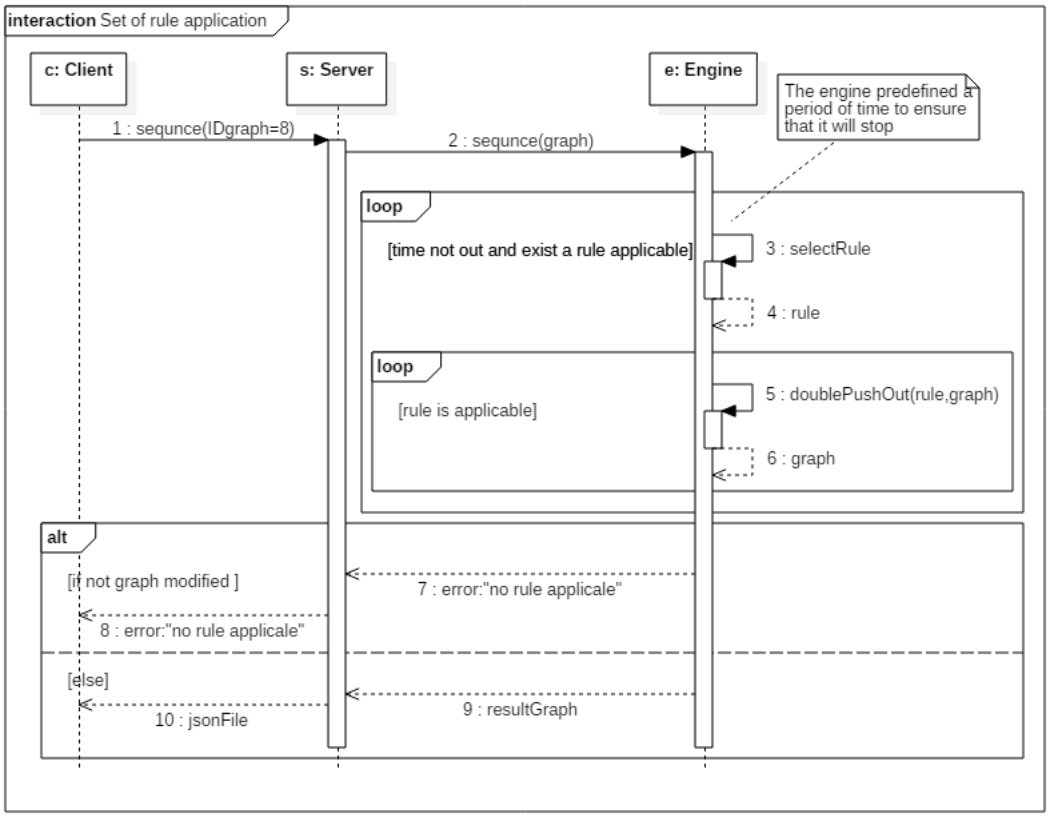


Figure 4.3: Sequence diagram of the graph transformation sequence

The sequence diagram of conform

Figure 4.4 presents the sequence diagram of conform, the client sends the identifier of the graph which we will check if it conforms to the type graph or not, then the server sends the corresponding graph and the type graph if the type graph level is not disabled the Engine checks if the graph conforms to the type graph, if it conforms, it returns true else it returns an error message which explains the problem.

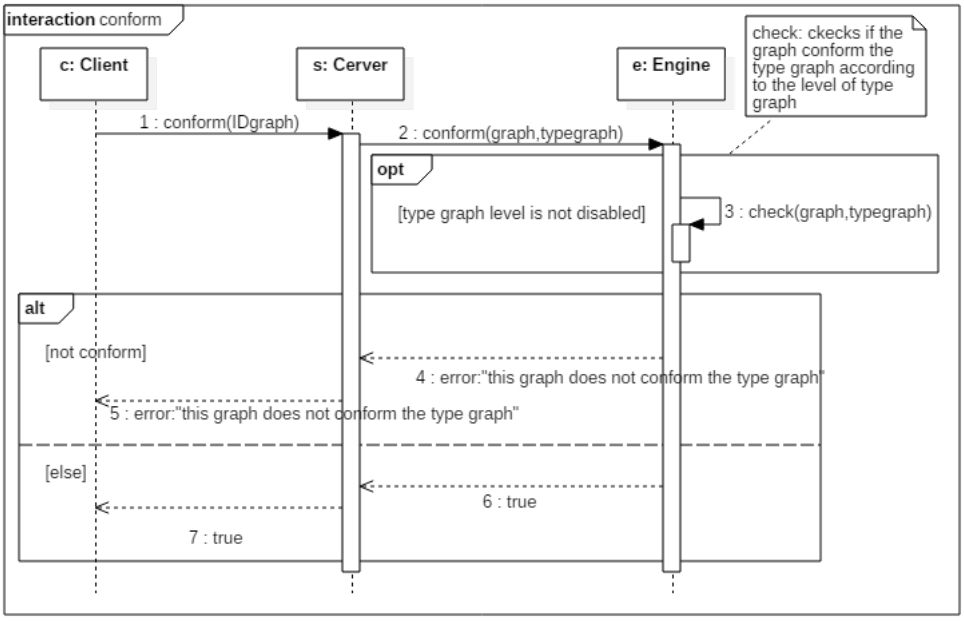


Figure 4.4: The sequence diagram of conform

4.4.3 List of APIs offered by the REST server

Table 4.1 describes APIs offered by the REST server.

Table 4.1: List of APIs offered by the REST server

Number	API	Description
1	/grammars	returns a full list of graph grammar
2	/grammars/:name	returns information of a specific graph grammar
3	/type/nodes	returns a full list of type nodes of a specific graph grammar
4	/type/arcs	returns a full list of type arcs of a specific graph grammar
5	/type/graph	returns the type graph of a specific graph grammar
6	/type/graph/nodes	returns the list of nodes of type graph exists in a specific graph grammar
7	/type/graph/arcs	returns the list of arcs of type graph exists in a specific graph grammar
8	/graphs	returns a full list of graphs exist in a specific graph grammar

Table 4.1 – List of APIs offered by the REST server

Number	API	Description
9	/graphs/:id	returns information of a specific graph
10	/graphs/:id/nodes	returns a full list of nodes of a specific graph
11	/graphs/:id/arcs	returns a full list of arcs of a specific graph
12	/rules	returns a full list of rules exist in a specific graph grammar
13	/rules/:id	returns a specific rule
14	/rules/:id/left	returns the left graph of a specific rule
15	/rules/:id/right	returns the right graph of a specific rule
16	/rules/:id/left/nodes	returns a full list of nodes of the left graph of a specific rule
17	/rules/:id/left/arcs	returns a full list of arcs of the left graph of a specific rule
18	/rules/:id/morphism	returns the morphism that is exists between the left and the right graph of rule
19	/rules/:id/nacs	returns a full list of negative application conditions of a specific rule
20	/rules/:id/nacs/:name	returns information of a specific negative application conditions
21	/transform/:ruleID/graphID	returns the result graph of the application of a specific rule into a specific graph
22	/applicable/:ruleI	:graph cheDID/ck if a specific rule is applicable into a specific graph, if the rule is applicable it returns true, returns false otherwise.
23	/transform/:graphID	returns the result graph of the application of a list of rules into a specific graph
24	/conform/graphs/:id/	check if a specific graph conform the type graph

4.5 Example of Application

To explain our example, we assume that our server is running locally, therefore to access it, we use this address ≪ http://localhost/≫. For every defined APIs to be executed we just concatenate the API with the address of the server.

The objective of this example is to show a scenario of transforming a specific graph by a specific rule.

1. **Requesting list of available graph grammars**:
   To request the list of available graph grammars the client use this endpoint "/grammars". The server response to this request by returning the list of available graph grammars by using JSON format. In our example, there are two available graph grammar the first called lovers_graph and the second called statecharts as shown in the listing. We can access the information of the first and the second by using the URIs "/grammars/Lovers_Graph" and "/grammars/statecharts" respectively.

Listing 10 Example of a JSON Response of requesting list of available graph grammars

```
{
        "GraphGrammar":[{
                "name": "Lovers\_Graph",
                "uri": "/grammars/Lovers\_Graph"
            }
            {
                "name": "statecharts",
                "uri": "/grammars/statecharts"
            }
        ]
}
```

2. **Requesting information about a specific graph grammar**:
   To get information about a specific graph grammar the client use the URI "grammars/Lovers_Graph". The server response to this request by returning the identifier and the name of this graph grammar, and also URIs corresponding to its type nodes, type arcs, type graph, graphs, and rules.

Listing 11 Example of JSON response for requesting information about a specific graph grammar

```
{
        "ID": "I1",
        "name": "Lovers\_Graph",
        "Types": {
                "NodeType": {
                        "uri": "type/nodes"
                },
                "EdgeType": {
                        "uri": "type/arcs"
                },
                "Graph": {
                        "uri": "type/graphs"
                }
        },
        "Graph": {
                "uri": "/graphs"
        },
        "Rule": {
                "uri": "/rules"
        }
}
```

3. **Requesting a list of graphs**:
   To get the list of graphs exists in a specific graph grammar the user use the URI: "/graphs". And the server returns the list of existing graphs. In our case, the specified graph grammar has just one graph named "HostGraph".

Listing 12 Example of JSON response for requesting a list of graphs

```
{
        "Graph": [{
                "ID": "I7",
                "name": "HostGraph"
        }]
}
```

4. **Requesting information about a specific graph**:
   To request information about a specific graph the client use the URI: "/graphs/I17". The server responses by returning the identifier, the kind, and the name of the specified graph, and returns also URIs corresponding to its nodes and arcs.

Listing 13 Example of JSON response for requesting information about a specific graph

```
{
        "Graph": {
                "ID": "I7",
                "kind": "HOST",
                "name": "HostGraph",
                "Node": {
                        "uri": "/graphs/I7/nodes"
                },
                "Edge": {
                        "uri": "/graphs/I7/Edges"
                }
        }
}
```

5. **Requesting list of nodes of a specific graph**:
   to get the list of nodes of a specific graph, the client uses the URI "/Graph/I17/nodes". The server returns the list of nodes of this graph and returns for each node its identifier and its type.

Listing 14 Example of JSON response for equesting list of nodes of a specific graph

```
{
        "Node": [{
                "ID": "I29",
                "type": "I2"
        },
        {
            "ID": "I30",
            "type": "I2"
        }
    ]
}
```

6. **Requesting list of arcs of a specific graph**:
   to get the list of arcs of a specific graph, the client use the URI "/Graph/I17/arcs". The server returns the list of arcs of this graph and returns for each arc its identifier and its type.

Listing 15 Example of JSON response for equesting list of arcs of a specific graph

```
{
        "Edge": [{
                "ID": "I29",
                "type": "I2"
        },
        {
            "ID": "I30",
            "type": "I2"
        }
    ]
}
```

7. **Requesting list of rules**:
   To get the list of rules, the client uses the URI "/rules". The server response to this request by returning the list of rules and returning for each rule its identifier, name, and the URI used to get information about each rule. In this example our graph grammar has three rules named "NewPerson", "SetRelation", "RemoveRelation".

Listing 16 Example of JSON response of requesting list of rules

```
{
        "Rule": [{
                "ID": "I15",
                "name": "NewPerson",
                "uri": "/rules/I15"
        },
        {
                "ID": "I19",
                "name": "SetRelation",
                "uri": "/rules/I19"
        },
        {
                "ID": "I27",
                "name": "RemoveRelation",
                "uri": "/rules/I27"
        }
        ]
}
```

8. **Check the applicability of a specific rule on a specific graph**:
   To check the applicability of a specific rule on a specific graph, The client uses the URI "transformation/I15/I7". In this example the specified rule is applicable on the specified graph, so the server returns true.

Listing 17 Example of JSON response of requesting list of rules

```
{
    "Graph":"I7",
    "Rule":"I15",
    "applicable":true
}
```

9. **Apply a specific rule on a specific graph**:
   To apply a specific rule on a specific graph. The client uses the URI "transformation/I15/I7". If the specified rule is applied. The server returns true, and sets of added and removed elements. In this example, the application of the rule "NewPerson" leads to the addition of one element to the graph "HostGraph" which is the node that has the identifier "I35".

Listing 18 Example of JSON response of applying a specific rule on a specific graph

```
{
    "Graph": "I7",
    "Rule": "I15",
    "applied": true,
    "Graph": {
        "uri": "/graphs/I17"
    },
    "Added": [{
        "ID": "I35"
    }],
    "Deleted": null
}
```

4.6 Conclusion

In this chapter, we have presented the design of the application that we used to demonstrate the use of our library. Because our application is developed according to the REST architecture, we have started this chapter a brief definition of the REST architecture.

General Conclusion

In this project we proposed a solution to the problem (reusing and decoupling the engine from the graphical interface) of existing tools implementing the graph transformation concepts.

Our engine is developed taking into account the reuse. It is a library that defines and expose a set of APIs. They are of two types, the first one is for graph grammar manipulation such as add, remove, and find element. And the second one is for graph transformation such as find matches,check rule applicability, apply transformation, and so more.

These APIs can be used directly or indirectly to perform graph transformation independently from the calling client. Our engine is developed using go programming language (golang) which is fast and easy to use.

In its current state, the graph transformation engine may be used directly using golang. For other languages, a binding is required to be defined. For the indirect use case, we have shown how simple it is to use our engine in chapter 4. We have developed an application following the REST architecture style. The engine decoupled from the graphical interface is executed in a server side, and the interface in the client side. Both the client and the server are communicating through HTTP protocol.

Our engine has the following characteristic:

- implements the graph transformation following the double push out approach (DPO)
- the graphs used in the transformation are typed graphs.
- Rules can have NACs (negative application condition).
- the selection of rules to be executed is either random or with priority.

perspective

What we have developed in our project is a core that can be enriched with many features. therefore we have defined the following perspective:

1. Add attributed graphs, PACs (positive application conditions), graph constraints, and critical pair analysis to the capabilities of the engine.

2. The development of a rich graphical user editor.

3. Define bindings for other languages such as c\c++.

Bibliography

[1] Guilherme Grochau Azzi, Jonas Santos Bezerra, Leila Ribeiro, Andrei Costa, Leonardo Marques Rodrigues, and Rodrigo Machado. The verigraph system for graph transformation. In *Graph Transformation, Specifications, and Nets*, pages 160–178. Springer, 2018.

[2] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F Korth. *Semantics and implementation of schema evolution in object-oriented databases*, volume 16. ACM, 1987.

[3] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[4] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation–part i: Basic concepts and double pushout approach. In *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 1: Foundations*, pages 163–245. World Scientific, 1997.

[5] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. Viatra-visual automated transformations for formal verification and validation of uml models. In *Proceedings 17th IEEE International Conference on Automated Software Engineering,*, pages 267–270. IEEE, 2002.

[6] Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Frank Hermann. *Graph and Model Transformation*. Springer, 2015.

[7] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Doctoral dissertation, 2000.

[8] Steven Kelly and Juha-Pekka Tolvanen. Visual domain-specific modelling: Benefits and experiences of using metacase tools. In *International Workshop on Model Engineering, at ECOOP*, volume 2000. Citeseer, 2000.

[9] Barbara König, Dennis Nolte, Julia Padberg, and Arend Rensink. *A Tutorial on Graph Transformation*, pages 83–104. Springer International Publishing, Cham, 2018.

[10] Alfio Martini, Hartmut Ehrig, and Daltro José Nunes. *Graph Grammars: An Introduction to the Double-pushout Approach*. Technische Universität Berlin, Fachbereich 13, Informatik, 1996.

[11] Brendan D McKay et al. *Practical graph isomorphism*. Department of Computer Science, Vanderbilt University Tennessee, USA, 1981.

[12] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The fujaba environment. In *Proceedings of the 22nd international conference on Software engineering*, pages 742–745. ACM, 2000.

[13] Grzegorz Rozenberg. *Handbook of Graph Grammars and Comp.*, volume 1. World scientific, 1997.

[14] Michael Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *International Workshop on Theory and Application of Graph Transformations*, pages 238–251. Springer, 1998.

[15] Andy Schürr. Specification of graph translators with triple graph grammars. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1994.

[16] G Taentzer, U Prange, K Ehrig, and H Ehrig. Fundamentals of algebraic graph transformation. with 41 figures (monographs in theoretical computer science. an eatcs series), 2006.

[17] Gabriele Taentzer. Agg: A graph transformation environment for modeling and validation of software. In *International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 446–453. Springer, 2003.

[18] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.