Internship Report

Introduction to Graph Transformation Proofs

Thomas Ernest CERESA

LIFSTAGE

Supervisor: Rachid Echahed, CNRS, LIG

Academic Supervisor: Alexandre Meyer

12 June 2024

Acknowledgements

I would like to warmly thank Rachid Echahed, who welcomed me duly, explained the theoretical formalisms and the tasks I had to do each week, and was always helpful and human during my internship.

Abstract

Graph-oriented databases are widely used across various domains, including social networks, biological research, recommendation systems, and fraud detection. These databases are represented by different types of graphs, which evolve over time. Consequently, there is a significant interest in graph transformation proofs and developing proof assistants to support these transformations. With the growing usage of graph databases and the emergence of new languages such as Neo4j, a new ISO standard for the Graph Query Language (GQL) was released in April 2024. In this report, we explore the theoretical foundations of graph transformations and discuss the implementation of a proof assistant to facilitate the verification of these transformations

Internship Context

I am currently in my third year of a Bachelor's degree (L3) at Claude Bernard University Lyon 1. Having always been passionate about research, I decided to take the leap and engage in a research internship to deepen my knowledge and gain experience. This internship is part of a course unit (U.E.) called LIFSTAGE.

The internship started on April 29 and will end on July 19. It takes place at the IMAG building, located in Saint-Martin-d'Hères. The host university is Université Grenoble Alpes, an institution that is particularly dear to me since it is where I began my university studies.

From the very beginning of this internship, it was necessary to address significant theoretical formalism to thoroughly understand the fundamentals of graph transformations. This theoretical foundation is essential for later improving the implementation of a proof assistant.

The project aims to create a proof assistant capable of verifying and validating graph transformations, a valuable tool for various application domains.

My internship supervisor, Rachid Echahed, warmly guided me throughout the process, helping me overcome theoretical and practical challenges.

Contents

1. Professionnal Environment

1.1 CNRS Grenoble & CAPP team

The CAPP team, jointly affiliated with CNRS, Grenoble INP, and UGA, focuses on advancing computational models within the realm of Formal Methods, Models, and Languages. Their researchs topics belong to the domains of automated reasoning, quantum computing and the theoretical foundations of programming languages. Their work encompasses algorithmic complexity, programming methodologies, and rigorous proof techniques, aiming to expand the boundaries of what is computationally feasible and applicable across diverse fields.

The exucutive of this team is LIG (Computer Science (**fr.** Informatique) Laboratory of Grenoble). It is 500 members: full-time researchers, PhD students, administrative and technical staff. Also, each year, we can reckon 500 scientific publications and 40 completed PhDs over 5 focus areas: Software and Information System Engineering ; Formal Methods, Models, and Languages ; Intelligent Systems for Bridging Data, Knowledge and Humans ; Interactive and Cognitive Systems; Distributed Systems, Parallel Computing, and Networks (c.f (Annexe) LIG organigram)

1.2 Supervisor of the internship

Rachid Echahed (dr., hdr.) is a research fellow in French National Centre for Scientific Research (CNRS). His domains are Graph and Term Rewriting; Program Verification; Multiparadigm; Programming. He closely works with two doctoral students and Angela Bonifati, Distinguished Professor in CS, at Claude Bernard Lyon I University affiliated with the CNRS Liris. They mainly work on the topic of graph databases.

1.3 Methods and Organization of the Internship

I had an office in the IMAG building. Once a week, I met with my supervisor to discuss my progress. During these meetings, I presented my understanding of the topics through academic articles, especially in the initial weeks. This is why the report includes a chapter titled "Theory", for I reckon that it is an important part of this internship. I also read a lot of literature provided by my supervisor or found on my own. The application I worked on was initially developed by a previous intern. It is a web application built with Flask, incorporating all the theoretical points I studied, which greatly aided my understanding. In the subsequent weeks, I focused on improving this application, addressing incomplete areas, fixing bugs.

Figure 1.1: IMAG Building

2. Theory

2.1 Graphs, Rewriting

2.1.1 Decorated graphs

The beginning of my internship was a mastering of a large formalism, particularly decorated graphs. They are over two alphabet $\mathcal{C}$ (the labels of nodes, "concepts") and $\mathcal{R}$ (the labels of edges, "roles"). A decorated graph is a tuple $(N, E, \Phi_N, \Phi_E, \text{source}, \text{target})$. N (resp. E) is set of nodes (resp. edges). We equip graphs two functions, $\Phi_N : N \to \mathcal{P}(\mathcal{C})$ and $\Phi_E : E \to \mathcal{R}$, which return the label(s) of the requested node or edge. (Notice: $\mathcal{P}(\mathcal{C})$ is necessary, for a node can not to have lables or can have several.). Finally, since these are directed graphs, we need to identify the direction of the edges. We provide the graph with a function source : $E \to N$ that returns the starting point of edges, and target : $E \to N$ that returns the endpoint of edges.

2.1.2 Actions

Terminology

To transform these graphs, we use actions. Think of actions not as functions but as instructions to produce graph transformations. Actions are primarily abstract; they have a name and input parameters.

The transformation of a graph by an action is written as $G[a]$. $G[a]$ is a graph of the same nature as G with the modifications specified by a.

A composition of actions α, also called a sequence of actions, is a series of actions performed inductively:

- $G[\epsilon] = G$
- $G[a] = (G[a])[\epsilon]$
- $G[a_0; a_1; \ldots; a_n] = (G[a_0])[a_1; \ldots; a_n]$

Similarly, for a sequence of actions α and an action a, we simplify the syntax as: $G[a; \alpha] = (G[a])[\alpha]$

Elementary actions

Graph transformations often use the same basic instructions, called elementary actions. The most common ones are adding and deleting:

- $add_N(i)$ *with* $i \in N$: Adds node i to the graph without any edges or labels.
- $delete_N(i)$ *with* $i \in N$: Removes node i and all its edges from the graph.
- $add_C(i, c)$ *with* $i \in N,\ c \in \mathcal{C}$: Adds label c to node i.
- $delete_C(i, c)$ *with* $i \in N,\ c \in \mathcal{C}$: Removes label c from node i.
- $add_E(e, i, j, r)$ *with* $e \in E,\ \{i, j\} \in N^2,\ r \in \mathcal{R}$: Adds edge e with label r between nodes i and j, with i as the source.
- $delete_E(e, i, j, r)$ *with* $e \in E,\ \{i, j\} \in N^2,\ r \in \mathcal{R}$: Removes all edges between i and j with label r.

There are also less obvious elementary actions like merging, redirecting, and cloning a node.

$merge(i, j)$ *with* $\{i, j\} \in N^2$: Merges nodes i and j into one. The resulting node has all incoming and outgoing edges of both nodes and their combined labels. (Fig 2.1)

$redirect(i, j)$ or $i >> j$: Redirects all incoming edges of i to j. (Fig 2.2)

Examples (Annexe Fig 7.5 7.6

This formalism is highly inspired from [1].

2.1.3 Rules, Rewriting System

Concrete Example

To understand which is a rule, I have to get out from computer science. A rule for example, in grammar, to make a question from an affarmative sentence, we add an auxiliary before the subject and we remove the conjugation of the verb. Here, it will be similary for graph transformation. It may do thinking a context-free grammar [2] where we recognise a context (a homomorphism [3]) and then we apply actions.

Rule

A rule ρ is defined as a pair (L, α) where L is a labeled graph with alphabets $\mathcal{C}$ and $\mathcal{R}$, serving as the context, and α represents a sequence of actions. This rule is denoted as $L \to \alpha$ in formal terms.

A rewriting system comprises a collection of such rules that dictate transformations on graphs.

Pattern-matching

To determine if a graph G, decorated with labels, matches the left-hand side (lhs) L of a rule, we utilize two pattern-matching functions:

- $h^N : N^L \to N^G$ for nodes.
- $h^E : E^L \to E^G$ for edges.

These functions ensure:

1. For every node $n \in N^L$ in graph L and every concept $c \in \Phi_N^L$, there exists a node $h^N(n) \in N^G$ in graph G that satisfies concept c.
2. For every edge $e \in E^L$ in graph L, the label assigned by Φ_E^G to edge $h^E(e)$ in G matches the label $\Phi_E^L(e)$ in L.
3. The source node in G associated with edge $h^E(e)$ aligns with the node $h^N(source^L(e))$ in L for every edge $e \in E^L$.
4. Similarly, the target node in G associated with edge $h^E(e)$ aligns with the node $h^N(target^L(e))$ in L for every edge $e \in E^L$.

Application of a rule

A graph G, decorated with labels, undergoes transformation to G' using rule $\rho = (L, \alpha)$ if there exist valid pattern-matching functions h^N and h^E between L and G. The resulting transformed graph G' is obtained by applying the sequence of actions α to G, where each node n in L is replaced by $h^N(n)$ in α. This transformation is represented as $G' = G[h(\alpha)]$. The notation $G \to_\rho G'$ signifies the application of rule ρ.

Illustrated exemples (Annexe Fig 7.2 7.3 7.4)

2.2 Proofs, Verifications, Correctness

2.2.1 Strategies

It becomes evident that a single rule is often insufficient for representing programs or complex rewritings; multiple rules are needed, and choices between them must be made. Thus, strategies in this context resemble sequences of instructions, akin to imperative programming languages. Let $\mathcal{S}$ denote the set of strategies and R denote the graph rewriting system. $\mathcal{S}$ is inductively defined as follows:

1. $\epsilon \in \mathcal{S}$ (skip instruction)
2. If $p \in R$, then $p \in \mathcal{S}$ (single rule as a strategy)

3. If $s_1 \in \mathcal{S}$ and $s_2 \in \mathcal{S}$, then $s_1; s_2 \in \mathcal{S}$ (sequence: apply s_1, then s_2)
4. If $s_1 \in \mathcal{S}$ and $s_2 \in \mathcal{S}$, then $s_1 \oplus s_2 \in \mathcal{S}$ (choice: apply either s_1 or s_2)
5. If $p \in R$, then $p^* \in \mathcal{S}$ (closure: repeat p as long as possible, akin to a "while" structure)
6. If $p \in R$, then $p! \in \mathcal{S}$ (obligatory: apply rule p)
7. If $p \in R$, then $p? \in \mathcal{S}$ (trial: attempt to apply rule p)

These semantics provide operational meaning to each term within $\mathcal{S}$, aligning them with concepts familiar in imperative programming languages.

Applications

Previously, we saw that $G \rightarrow_\rho G'$ means G' is obtained by applying the action of ρ on G. Simulary, we jot down $G \Rightarrow_s G'$ for a strategy s.

There is an important formula : "$App(s)$" such as for all decorated graph G, which can be applied by **at least one step** of the strategy s. We write $G \models App(s)$.

To be more clear, let ρ be a rule. We say that G satisfies $App(\rho)$ if and only if G is pattern-matched with the left-hand side of ρ. Then let $s = \rho_1; \rho_2; \rho_3$. If $G \models App(s)$, then G matches necessary ρ_1.

Furthermore:

- $G \models \text{App}(s_1 \oplus s_2) \iff G \models \text{App}(s_1) \vee G \models \text{App}(s_2)$
- $G \models \text{App}(s^*) \Rightarrow G \models \text{App}(s)$

2.2.2 Specifications

Specifications on our futur proofs respect a Hoare [4] style. Let R a graph rewriting system, s a strategy. Both, in a specifiation, are conditionned by Pre and $Post$. Pre as well as $Post$ must be both logic propositons – Pre ($resp.$ $Post$) must be true before the rewriting ($resp.$ at the end). Notice this current intership limits itself to using FOL [5]

Therefore we write a specification as a triple: $\{Pre\}R, s\{Post\}$

2.2.3 Weakest precondition

In general

In Hoare calculus [6], the weakest precondition (introduced by Dijkstra [7]) automates deduction. It effectively builds an algorithm to verify Hoare triples.

The core idea is to reduce the problem to proving a FOL (First Order Logic) formula. We seek a formula $\mathcal{W}$ such that $Pre \Rightarrow \mathcal{W}$ is a tautology. This concept adapts the Post condition to the Pre condition, ensuring correctness when applying a statement (or strategy). We denote $\mathcal{W}$ as $wp(s, Q)$, where $s \in \mathcal{S}$ and Q is a FOL formula.

For instance, consider the Hoare triple $\{x = 2\}x := x - 4\{x < 10\}$. We compute $wp(x := x - 4, x < 10)$, yielding $x - 4 < 10$, hence $x < 14$. Therefore, $x = 2 \Rightarrow x < 14$ holds true, validating the triple.

For graph rewriting system

To adapt this to our context, it's crucial to demonstrate the correctness of our specialized triple. Ensuring the correctness involves proving that for all decorated graphs G and G', where $G \Rightarrow_s G'$, $G \models Pre$ and $G' \models Post$.

The calculation of the weakest precondition $wp(s, Q)$ requires substituting Q. Here, these substitutions correspond to elementary actions. Let a denote an elementary action. A substitution of a is denoted as $[a]$, resulting in $wp(s, Q) = Q[a]$.

Consider $\mathcal{F}$ as a First Order Logic (FOL) formula and G as a graph. We seek G to satisfy the substitution of $\mathcal{F}$. It's evident that if G satisfies $\mathcal{F}$ with substitution, then $\mathcal{F}$ must also hold true without substitution when the elementary action is applied to itself:

$$G[a] \models \mathcal{F} \iff G \models \mathcal{F}[a]$$

This principle guides us in computing the weakest precondition of Post.

2.2.4 Translation Graphs into FOL Formula

Since the beginning of this report, we have not seen how graphs are really described in a logics. For our proofs, I proposed in my former article this interpretation. (Briefly, the concepts are unary predicates and the roles are binary predicates).

Let $\mathcal{I} : \mathcal{F} \to \mathbb{B}$ be an interpretation. Define the logical boolean operators: $+ : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ (or), $\cdot : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ (and), and $\_ : \mathbb{B} \to \mathbb{B}$ (negation). Let $\top$ represent true and $\bot$ represent false. Clearly, $\neg\top \equiv \bot$ and $\neg\bot \equiv \top$.

Assume the following, where `pred` is a unary predicate, `pred2` is a binary predicate, F and G are formulas, and x and y are nodes:

1. $\mathcal{I}(\top) = \top$
2. $\mathcal{I}(\text{pred}(x)) = \top \iff \text{pred} \in \Phi_N(x)$
3. $\mathcal{I}(\text{pred2}(x, y)) = \top \iff \exists e$ such that $\text{pred2} \in \Phi_E(e) \wedge s(e) = x \wedge t(e) = y$

4. $\mathcal{I}(F \vee G) = \mathcal{I}(F) + \mathcal{I}(G)$
5. $\mathcal{I}(F \wedge G) = \mathcal{I}(F) \cdot \mathcal{I}(G)$
6. $\mathcal{I}(\neg F) = \overline{\mathcal{I}(F)}$
7. $\mathcal{I}(x = y) = \top \iff x$ is the same node as y
8. $\mathcal{I}(\exists x\, F) = \top \iff \exists n \in N, \mathcal{I}(F[x := n]) = \top$
9. $\mathcal{I}(\forall x\, F) = \top \iff \forall n \in N, \mathcal{I}(F[x := n]) = \top$

2.2.5 Substitutions

The $\rightsquigarrow$ symbol denotes the rewriting of a formula with the application of substitutions. The following substitutions are assumed (where α is an action; pred and pred' are unary predicates; pred2 and pred2' are binary predicates; x and y are nodes):

1. $\top[\alpha] \rightsquigarrow \top$
2. $(\neg F)[\alpha] \rightsquigarrow \neg F[\alpha]$
3. $(\exists x, F)[\alpha] \rightsquigarrow \exists x, F[\alpha]$
4. $(\forall x, F)[\alpha] \rightsquigarrow \forall x, F[\alpha]$
5. $(G \vee F)[\alpha] \rightsquigarrow G[\alpha] \vee F[\alpha]$
6. $(G \wedge F)[\alpha] \rightsquigarrow G[\alpha] \wedge F[\alpha]$
7. $\text{pred}(x)[\text{add}_C(i, \text{pred})] \rightsquigarrow \text{pred}(x) \vee i = x$
8. $\text{pred}(x)[\text{del}_C(i, \text{pred})] \rightsquigarrow \text{pred}(x) \wedge \neg(i = x)$
9. $\text{pred'}(x)[\text{add}_C(i, \text{pred})] \rightsquigarrow \text{pred'}(x)$
10. $\text{pred'}(x)[\text{del}_C(i, \text{pred})] \rightsquigarrow \text{pred'}(x)$
11. $\text{pred2}(x, y)[\text{add}_C(i, \text{pred})] \rightsquigarrow \text{pred2}(x, y)$
12. $\text{pred2}(x, y)[\text{del}_C(i, \text{pred})] \rightsquigarrow \text{pred2}(x, y)$
13. $\text{pred}(x)[\text{add}_R(i, j, \text{pred2})] \rightsquigarrow \text{pred}(x)$
14. $\text{pred}(x)[\text{del}_R(i, j, \text{pred2})] \rightsquigarrow \text{pred}(x)$
15. $\text{pred2}(x, y)[\text{add}_R(i, j, \text{pred2})] \rightsquigarrow \text{pred2}(x, y) \vee (x = i \wedge y = j)$
16. $\text{pred2}(x, y)[\text{del}_R(i, j, \text{pred2})] \rightsquigarrow \text{pred2}(x, y) \wedge \neg(x = i \vee y = j)$
17. $\text{pred2'}(x, y)[\text{add}_R(i, j, \text{pred2})] \rightsquigarrow \text{pred2'}(x, y)$

18. $\text{pred2'}(x,y)[\text{del}_R(i,j,\text{pred2})] \rightsquigarrow \text{pred2'}(x,y)$
19. $\text{pred}(x)[i \gg j] \rightsquigarrow \text{pred}(x)$
20. $\text{pred2}(x,y)[i \gg j] \rightsquigarrow (\text{pred2}(x,i) \wedge y = j) \vee (\text{pred2}(x,y) \wedge \neg(y = i))$
21. $\text{pred}(x)[\text{merge}(i,j)] \rightsquigarrow (\text{pred}(x) \vee (\text{pred}(j) \wedge x = i)) \wedge \neg(x = j)$
22. $\text{pred2}(x,y)[\text{merge}(i,j)] \rightsquigarrow \neg(x = j \vee y = j) \wedge (\text{pred2}(x,y) \vee (\text{pred2}(j,y) \wedge x = i) \vee (x = i \wedge y = i \wedge \text{pred2}(j,j)) \vee (\text{pred2}(x,y) \vee (\text{pred2}(x,j) \wedge y = i))$

Important Notice : The semantics translate the actions. In fact, we never rewrite graphs, but we represent changements and constraints in logics. For example, our implementation does not satisfy the formalism: all nodes and edges must have at least one label. To make satisfied the formalism ($\Phi_N : N \to P(\mathcal{C})$ and not $\Phi_N : N \to \mathcal{C}$), I proposed, at the beginning of the intership, a special unary predicate "Node", and the substitutions are as follows:

1. $\text{Node}(x)[\text{add}_N(n)] \rightsquigarrow \text{Node}(x) \vee n = x$
2. $\text{Node}(x)[\text{del}_N(n)] \rightsquigarrow \text{Node}(x) \wedge \neg(n = x)$
3. $\text{Node}(x)[i \gg j] \rightsquigarrow \text{Node}(x)$
4. $\text{Node}(x)[\text{merge}(i,j)] \rightsquigarrow \text{Node}(x) \wedge \neg(x = j)$
5. $\text{Node}(x)[\alpha] \rightsquigarrow \text{Node}(x)$ (for other actions)

The removal of a node can significantly alter a graph, as edges connected to the removed node are also removed. Additional substitutions are needed:

1. $\text{pred2}(x,y)[\text{del}_N(n)] \rightsquigarrow \text{pred2}(x,y) \wedge \neg(x = n \vee y = n)$
2. $\text{pred}(x)[\text{del}_N(n)] \rightsquigarrow \text{pred}(x) \wedge \neg(x = n)$

2.2.6 Weakest precondition (strategies)

Overview

Invariant and Closure

The closure requires an invariant to determine when the strategy can repeat. Let inv_s be such an invariant formula. We define $wp(s^*, Q) = inv_s$, which is a weakest liberal precondition (wlp) since the termination of s is not guaranteed. Ensuring inv_s holds for each iteration requires $inv_s \Rightarrow wp(s, inv_s)$, validating the loop's behavior.

$$
\begin{array}{ll}
wp(a,\ Q) = Q[a] & wp(a;\alpha,\ Q) = wp(a, wp(\alpha, Q)) \\
wp(\epsilon,\ Q) = Q & wp(s_0; s_1,\ Q) = wp(s_0, wp(s_1,\ Q)) \\
wp(s_0 \oplus s_1,\ Q) = wp(s_0, Q) \wedge wp(s_1, Q) & wp(s^\star,\ Q) = inv_s \\
wp(\rho,\ Q) = App(\rho) \Rightarrow wp(\alpha_\rho, Q) & wp(\rho!, Q) = App(\rho) \wedge wp(\alpha_\rho, Q) \\
wp(\rho?, Q) = (App(\rho) \Rightarrow wp(\alpha_\rho, Q)) \wedge (\neg App(\rho) \Rightarrow Q) &
\end{array}
$$

Figure 2.1: Weakest preconditions

2.2.7 Verification Conditions

Define vc as a FOL function. For basic strategies, $vc(\epsilon, Q)$, $vc(\rho, Q)$, $vc(\rho!, Q)$, and $vc(\rho?, Q)$ are trivially true. For a loop strategy s, $vc(s^*, Q)$ must verify:

$$vc(s^*, Q) = vc(s, inv_s) \wedge (inv_s \Rightarrow (App(s) \Rightarrow wp(s, inv_s))) \wedge (inv_s \Rightarrow (\neg App(s) \Rightarrow Q))$$

This ensures s is correct, inv_s is preserved if s can be applied, and the postcondition holds when s can no longer be applied. Additionally, vc handles sequences and choices of strategies:

$$vc(s_0; s_1, Q) = vc(s_0, wp(s_1, Q)) \wedge vc(s_1, Q)$$
$$vc(s_0 \oplus s_1, Q) = vc(s_0, Q) \wedge vc(s_1, Q)$$

2.2.8 Total Correctness

Annotation

Users must provide invariants for loops. Thus, an annotated strategy is written as $s^*\{inv\}$, where inv is the loop invariant. An annotated specification includes such annotated strategies.

Formula and Soundness

To validate an annotated specification $\mathcal{ASP} = \{Pre\}(R, s)\{Post\}$, we compute the weakest precondition and verify the correctness condition vc:

$$correct(\mathcal{ASP}) = Pre \Rightarrow (wp(s, Post) \wedge vc(s, Post))$$

If $correct(\mathcal{ASP})$ is valid, then for all graphs G and G', if $G \Rightarrow_s G'$, it follows that $(G \models Pre) \Rightarrow (G' \models Post)$.

3. Implementation

Throughout the internship, I work on an application of an assistant which follows the previous theory. I learn to use Z3 and the application. My first task was to make counterexamples more readable.

3.1 Overview

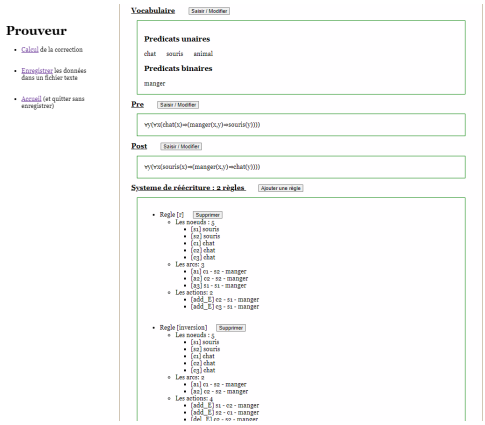

Figure 3.1: Example of a preparation of a proof

3.1.1 Architecture

The architecture is MCV. There are two main folder `Interface` and `Prouveur`. `Interface` contains the Jinja templates, the views and the frontal Flask controler. `Prouveur` can be intrepreted as the backend. There are calculus, datas, methods, etc. `Prouveur` has 3 folders `DatasNoyau`, `FormuleLogique`, and `Noyau`. I usually worked and changed the codes of `Noyau/Correction` where the correctness is managed, and the views too.

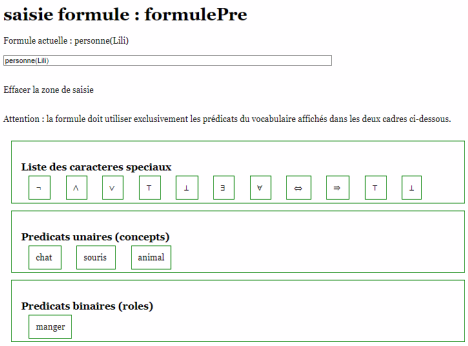

Figure 3.2: Input of formulas

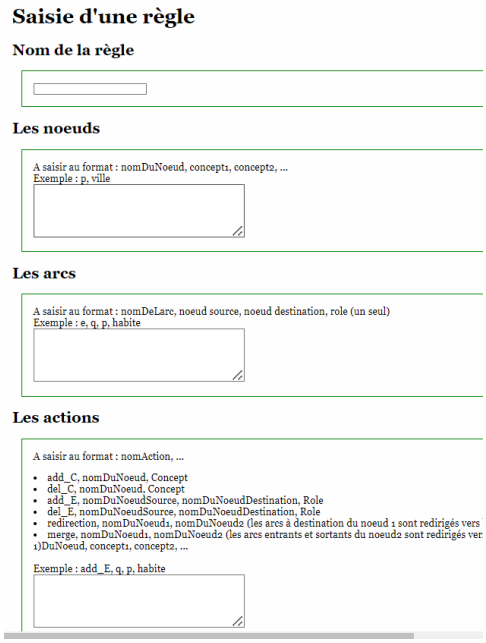

Figure 3.3: Form for a new rule

3.1.2 Datas, Logics to Z3

Datas are highly represented by a orientated object programmation. There are classes of Node, Edge, LeftHandSide, Rule, RightHandSide, and so much..

Datas are translated to the FOL logic. The class `ArbreFormule` transforms these latters to formulas.

3.1.3 Z3

Z3 is a high-performance theorem prover developed by Microsoft Research, used for solving logical formulas and verifying software and hardware. It supports various theories, including linear and nonlinear arithmetic, bit-vectors, arrays, and, importantly for our purposes, First-Order Logic (FOL).

An SMT (Satisfiability Modulo Theories) solver works by checking if logical formulas are satisfiable. It decomposes the problem into simpler SAT[1] (Satisfiability) problems and uses specialized solvers for each theory involved. The solver assigns values to variables while managing constraints. If it finds a consistent assignment, it declares the formula satisfiable; otherwise, it proves it unsatisfiable.

The main object in Z3 is the `Solver`. In our implementation, objects of the `ArbreFormule` class are translated to Z3 formulas using the `abre_to_z3` method from `ConversionZ3`. The `check()` method in Z3 verifies the formula. There are two boolean results, `sat` and `unsat`, which determine the satisfiability and unsatisfiability of the formula, respectively. When a formula is satisfiable, Z3 provides a model, which serves as a counterexample if the formula to be proved is negated (indicating the formula is incorrect). This is why the author chose to input the negative form of the formula to Z3.

model()

The `model()` method in Z3 returns a concrete assignment to the variables that satisfies the given formula. This can be used to understand why a formula is considered satisfiable. For example, if `eat(x, y)` should hold, `model()` provides specific values for x and y that satisfy this predicate under the constraints provided.

3.2 Counterexamples

The model() of z3 is hardly understable by a unspecialist. Hence I coded a model analyser (class `ModelAnalyser`

[1] `https://en.wikipedia.org/wiki/Boolean_satisfiability_problem`

Assignements

On models of Z3, arbitrary values are assigned to variables. `ModelAnalyser` recognises these assignments through simple pattern matching and saves them to a Python dictionary named `arbitrary_values_map`. The keys of the dictionary are the arbitrary values (e.g., `A!val!x`), and the values are the corresponding variables. This choice was made because, in many contexts, it is necessary to translate the arbitrary values back to the names of the nodes.

Representations of Straightforward Predicates

`ModelAnalyser` distinguishes predicates. They are saved in a special Python dictionary, `list_unvariable_predicate`, within `ModelAnalyser`. To illustrate, consider this model:

$[x = A!val!0, y = A!val!1, pred = [A!val!1 \to False, A!val!0 \to False, else \to True], pred2 = [else \to True]]$

Here, after running, `list_unvariable_predicate` is:

```
{
    'pred': {'A!val!0': False, 'A!val!1': False, 'else': True},
    'pred2': {'else': True}
}
```

Then, the new Jinja2 template `predicat.html` is tasked with representing `list_unvariable_predicate`. It shows sorts of 'cards' summarising the information of these predicates.

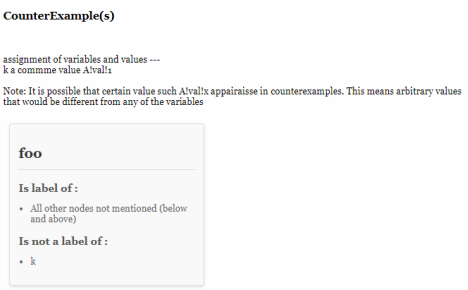

Figure 3.4: Example of the representation of the foo predicate

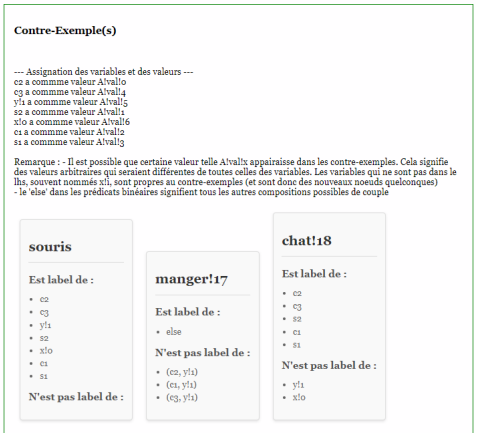

Figure 3.5: Example of an other conterexample

3.3 notices, bug fixing, exhaustive list

Throughout the internship I also find bug or incomplete feature. The most important to mention would be the incomplete pattern-matching. All the nodes of the left-hand-side were transformed into Z3 constants, while they should be variables.

I also found some mistakes on the formalism. The correctness formula was not correct. I fixed it.

I added features and inputs mainly in forms. For example, before we cannot modify a rule, I decide to make a proper form for this.

I have done lot of things which are no this important to be present in this report...

4. Futur tasks and Objectives

This report is written at the middle of the internship. My futur purpose/-tasks will be to add some features to the implementation for respecting the new ISO of *properties graph*.

4.1 Property Graphs

4.1.1 Abstraction

The current trend in graph databases is to add properties to decorated graphs. Similar to a hashmap or a dictionary, a property is a set of key-value pairs. These properties can be associated with nodes as well as edges.

To facilitate this, we need to consider new alphabets such as $\mathcal{K}$ (representing the type of keys) and $\mathcal{V}$ (representing the type of values). We define $\mathcal{P} = \mathcal{K} \times \mathcal{V}$. Consequently, we introduce functions $\Phi^N_{\mathcal{P}} : N \to P(\mathcal{P})$ (for nodes) and $\Phi^E_{\mathcal{P}} : E \to P(\mathcal{P})$ (for edges) that return the properties of a node or an edge, respectively. Additionally, the following functions are essential:

$$\mathcal{V}^N_{\mathcal{K}} : \mathcal{K} \times N \to P(\mathcal{V})$$
$$\mathcal{V}^E_{\mathcal{K}} : \mathcal{K} \times E \to P(\mathcal{V})$$

These functions return the value of a property based on the key.

Thus, a property graph can be defined as:

$$(N, E, \Phi_N, \Phi_E, \text{source}, \text{target}, \Phi^E_{\mathcal{P}}, \Phi^N_{\mathcal{P}}, \mathcal{V}^E_{\mathcal{K}}, \mathcal{V}^N_{\mathcal{K}})$$

4.1.2 Implementation

In collaboration with my supervisor, we devised a strategy to code property graphs and adapt the logic accordingly from the initial weeks of my project. I decided to retain a significant portion of the existing edge code to represent properties. For instance, if a node x has the label "person," we can add a property with the key "name" and the value "Bob." In our proofs, this would be represented as $name(x, \text{"Bob"})$.

This approach should not pose significant issues. However, I shall have to consider a new form to assist users effectively and ensure smooth data transfer to the application. This entails a clear and precise specification of the functions and structures needed to handle property graphs.

4.2 Constraints on Rules

My final task is to add constraints to rules. To illustrate this, consider nodes labeled "person" with a property ("age", x) where x is an integer. The goal of the transformation is to change the labels from "person" to "teen," "adult," or "senior" based on the ages of the individuals. We define three rules: ρ_1, ρ_2, ρ_3. Each rule has a left-hand side (lhs) consisting of a node x labeled "person" and right-hand sides (rhs) as follows:

- ρ_1: $del_C(x, \text{"}person\text{"}); add_C(x, \text{"}teen\text{"})$
- ρ_2: $del_C(x, \text{"}person\text{"}); add_C(x, \text{"}adult\text{"})$
- ρ_3: $del_C(x, \text{"}person\text{"}); add_C(x, \text{"}senior\text{"})$

Each rule has associated constraints:

- ρ_1: $x < 20$
- ρ_2: $20 \leq x < 80$
- ρ_3: $x \geq 80$

The objective is to ensure that after applying the strategy $(\rho_1 \oplus \rho_2 \oplus \rho_3)^*$, there are no nodes labeled "person" left, and the constraints are respected.

To achieve this, I will add a section to the rule definitions to incorporate constraints and adapt the code to handle integers and attribute constraints. This requires a comprehensive approach to rewriting the predicate `App()` to include these constraints.

By the end of this implementation, the system will be able to validate that transformations respect the given constraints, ensuring correctness and consistency in the application of rules based on node properties.

5. Conclusion

During my tenure at LIG, I engaged in several key activities that significantly contributed to both my personal and professional growth. My primary focus was on writing research articles, which involved mastering the intricacies of the research process. This task included developing a more human-centric approach to counterexamples, adapting new forms for variables input, and adjusting objects to fit these new paradigms. Additionally, I identified and corrected several bugs, such as replacing Z3 constants with Z3 variables and ensuring that formalism was consistently adhered to throughout the system.

Link to My Education

This experience was closely aligned with my academic background, allowing me to apply and reinforce my knowledge in several critical areas. I was able to leverage my understanding of proof techniques, formal languages, and classical logic. One of the most significant learning outcomes was gaining a profound comprehension of weakest preconditions. Additionally, I enhanced my knowledge of context-free grammars, regular languages, and their computational applications. These skills and insights are invaluable for my future studies and have solidified my foundation in these essential topics.

Future Perspectives

Reflecting on this internship, I have found myself reconsidering my career path, particularly towards becoming a researcher in fundamental computer science. This experience has reconciled me with the domain, and the exposure to the research lifestyle has been incredibly appealing. I have gained substantial skills in writing, LaTeX, information research, and synthesis. These competencies will not only aid me in my academic journey but also prepare me for a potential future in research. The practical application of theoretical knowledge during this internship has been immensely rewarding and has provided a clear perspective on my future aspirations.

6. References

[1] Verifying Graph Transformation Systems with Description Logics, Jon Hael Brenas, Rachid Echahed, and Martin Strecker. https://lig-membres.imag.fr/echahed/icgt18.pdf

[2] Cours de langage formel, Sylvain Brandel, Université Claude Bernard Lyon1. http://sylvain.brandel.pages.univ-lyon1.fr/langages/LF-CM03.pdf

[3] https://en.wikipedia.org/wiki/Monomorphism

[4] C. A. R. Hoare. An axiomatic basis for computer programming. Commun. ACM, 12(10):576–580, 1969.

[5] La logique du premier ordre, Sophie Pinchinat, IRISA, Université de Rennes 1. https://people.irisa.fr/Sophie.Pinchinat/LOG/LOGcoursFO.pdf

[6] Hoare Logic, Claude Marché, LRI. https://www.lri.fr/~marche/MPRI-2-36-1/2012/poly-chap2.pdf

[7] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. http://doi.acm.org/10.1145/360933.360975

Dedication

This work is dedicated to my dear friend Pierre, whose support and encouragement have been invaluable throughout this journey. I also want to acknowledge the magical power of the ”snail” at the Science Library of UGA, which provided a unique source of inspiration and motivation.

7. Annexes

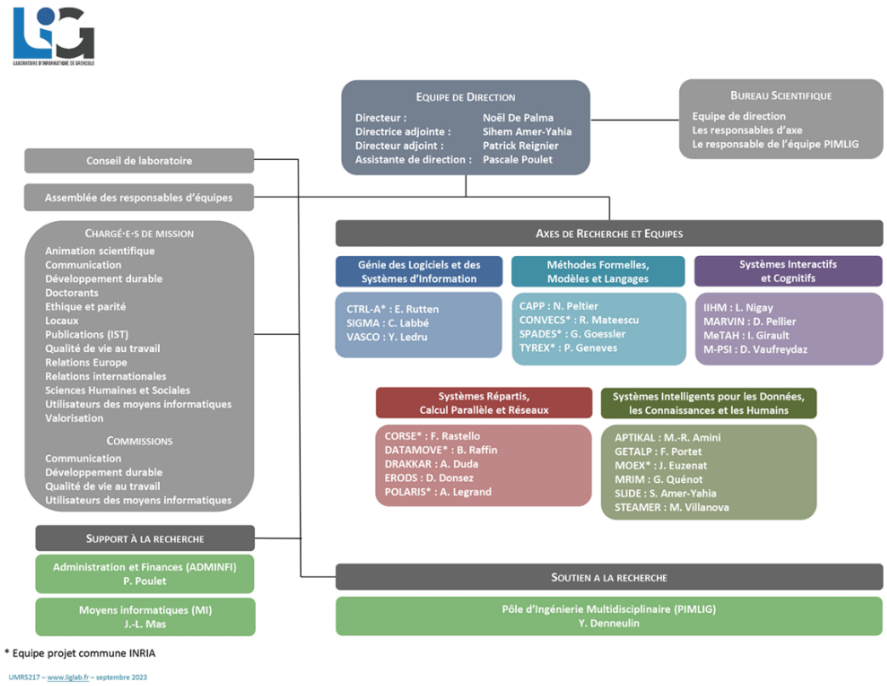

Figure 7.1: LIG Organigram

α = "merge nodes labeled c and d; delete node labeled e".

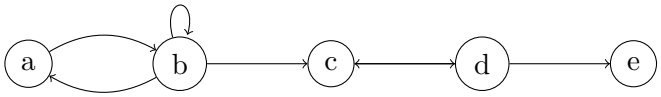

Figure 7.2: Decorated graph L

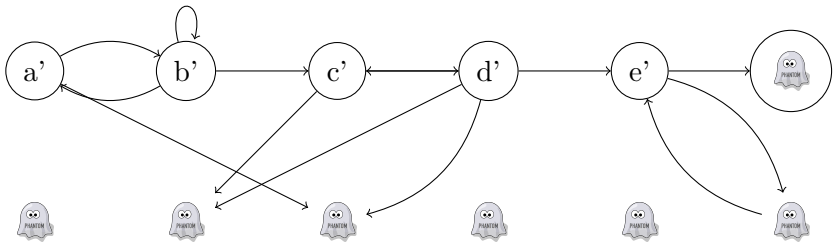

Figure 7.3: Decorated graph G

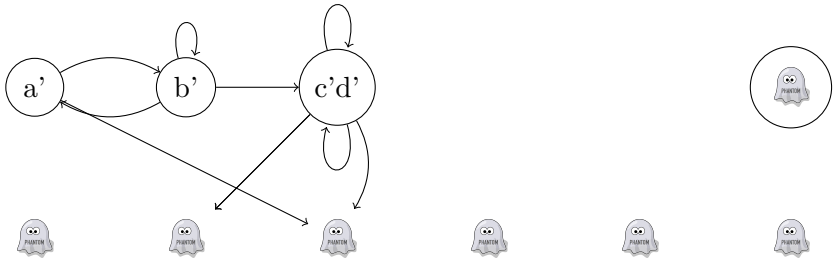

Figure 7.4: $G \rightarrow_\rho G'$

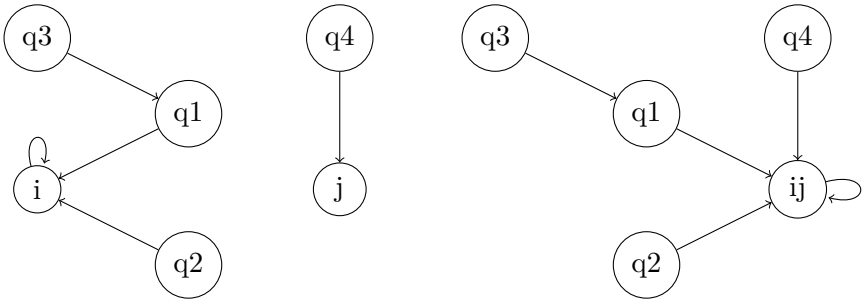

Figure 7.5: Exemple merge(i, j). (initial to the left, rewritten to the right)

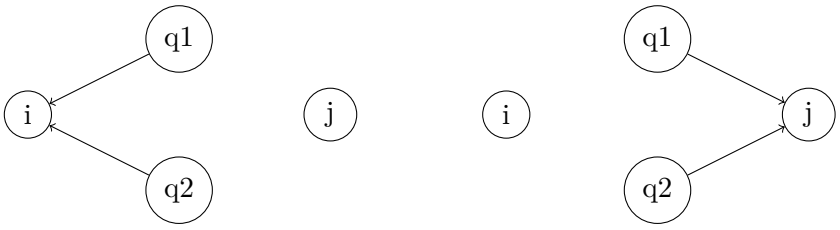

Figure 7.6: Example $i >> j$.