Reasoning in Descriptions Logics Augmented with Refreshing Variables

Théo Ducros

Université Clermont Auvergne
École Doctorale des Sciences pour l'Ingénieur

Thèse présentée par
Théo Ducros

en vue de l'obtention du grade de
Docteur d'université
Spécialité : Informatique

Reasoning in Description Logics Augmented with Refreshing Variables

Université Clermont Auvergne

Soutenue publiquement le 27/09/2022 devant le jury composé de :

François Goasdoué	Université de Rennes	Rapporteur
Nabil Layaïda	INRIA Grenoble	Rapporteur
Mirian Halfeld Ferrari Alves	Université d'Orléans	Examinatrice
Farouk Toumani	Université Clermont Auvergne	Directeur de thèse
Marinette Bouet	Université Clermont Auvergne	Directrice de thèse

Acknowledgements

Mes premiers remerciements vont tout naturellement vers mes directeurs de thèses, Marinette Bouet et Farouk Toumani. Merci pour votre temps, votre expertise, votre bienveillance et pour tout ce que que vous m'avez apporté aussi bien scientifiquement qu'humainement. Quelques mots ne suffiront jamais à exprimer la gratitude que j'éprouve à votre égard.

Je remercie également François Goasdoué, Nabil Layaïda et Mirian Halfeld Ferrari Alves d'avoir accepté d'évaluer ce travail de thèse ainsi que pour tous leur retours.

Je souhaite aussi remercier l'ensemble des membres du LIMOS avec une pensée toute particulière pour Béatrice qui trouve toujours une solution à mes problèmes.

Merci aux membres du département Génie Mathématique et Modélisation de Polytech de m'avoir fait confiance et permis de m'essayer à l'exercice d'enseignant. La variété des cours que vous m'avez confiée m'a permis de profiter pleinement de cette expérience.

Mes pensées se tournent maintenant vers mes collègues et amis qui ont égayé ces moments Merci Marina et Alexis C. pour vos précieux conseils et l'embellissement du bureau à Noël. Merci Matthieu pour ta gentillesse et ton aide avec les cours. Merci Alexis P. d'avoir partagé avec moi ce qui auraient pu être tes soliloques. Merci Ivan et Weixuan de m'avoir supporté au quotidien et d'avoir partagé avec moi votre culture.

Je termine ces remerciements par un immense Merci à mes proches pour leur soutien sans faille.

Abstract

Description logics are a family of knowledge representation that have been widely investigated and used in knowledge-based systems. The strength of description logics is beyond their modeling assets, it's their reasoning abilities. Reasoning takes the shape of mechanisms that make the implicit knowledge explicit. One of the most common mechanism is based on the subsumption relationship. This relationship is a hierarchical relationship between concepts which aims to state if a concept is more general than another. The associated reasoning tasks aims to determine the subsumption relationship between two concepts. Variables have been introduced to description logic to answer the needs of representing incomplete information. In this context, deciding subsumption evolved into two non-standard reasoning tasks known as matching and unification. Matching aims to decide the subsumption relationship between a concept and a pattern (i.e. a concept expressed with variables). Unification extends matching to the case where both entries are patterns. The semantics associated to variables can be qualified as non-refreshing semantics where assignment are fixed.

In this thesis, we investigate reasoning with variables augmented with refreshing semantics. Refreshing semantics enables variables to be released and then given a new assignment. We define recursive pattern queries as terminologies that may contain variables leading to investigation of problems to answer recursive pattern queries over description logic ontologies. More specifically, we focus on the description logic $\mathcal{EL}$. Recursive pattern queries are expressed in the logic $\mathcal{EL}_{\mathcal{RV}}$, an extension of the description logic $\mathcal{EL}$ with variables equipped with refreshing semantics. We study the complexity of query answering and query containment in $\mathcal{EL}_{\mathcal{RV}}$, two reasoning mechanisms that can be viewed as a variant of matching and unification in presence of refreshing variables. Our main technical results are derived by establishing a correspondence between this logic and a variant of variable automata. While the upper bound is given by specific algorithms which are proven to be optimal, the lower bound is achieved by a reduction to halting problem of alternating turing machine. Thus leading to these problems being EXPTIME-COMPLETE.

Keywords: Description logic; Refreshing semantics; Reasoning; Matching; Pattern containment.

Contents

List of Figures

List of Tables

Introduction

Description Logics (DLs) are a family of knowledge representation and reasoning formalisms that have been proven useful in many application domains[8]. DLs provide means for well-structured and formal representation of the conceptual knowledge of an application domain and various inference procedures to reason about the represented knowledge.

A DL Knowledge Base (KB) is composed of two components, called TBox and ABox, containing respectively general assertions describing relevant concepts and specific assertions about individuals and relationships among them.

For example, Table 1 depicts a KB made of two *concept definitions*, $Professor$ and $PhDStudent$, where the concept $Professor$ is described as a $person$ who works in a $university$ and has as a *doctoral student* : a *PhD student*. This knowledge base includes in addition assertions about individuals, stating that the individual *Alan* is an instance of $Professor$ and *Alice* is an instance of $PhDStudent$, as well as assertion about a relationships between them (*Alice* is a doctoral student of *Alan*).

Query answering over DL KBs has recently emerged as an advanced mechanism for accessing data sources through an ontology [18, 29, 26, 36]. Indeed, in many application contexts, an ontology is used to formalize the conceptual information about the contents of multiple data sources. This knowledge is then exploited during query evaluation to deduce additional information to enrich query answers beyond the data that is explicitly stored in a source. General forms of queries investigated in the literature are first-order formula with possibly free variables [36]. As an example, consider the following conjunctive query which asks for a $person$ that has a *doctoral student*: $q_{ex}(x) \leftarrow Pers(x), doctStudent(x, y)$

The answers to such a query are a set of valid substitutions for the variables such that the KB *entails* the query. For example, an answer to the query q_{ex} over KB_1 is *Alan* because the query $\sigma(q_{ex})$ obtained from q_{ex} by substituting the variable x by $\sigma(x) = Alan$ and y by $\sigma(y) = Alice$ is entailed by the KB of Table 1.

Beyond such conventional queries in the pure database style, the description logic community has been interested very early by query languages that include explicit *structural queries*, i.e., queries asking about properties of individual and concepts [24]. In particular, the notion of concept patterns, i.e., concept descriptions containing variables, has been introduced in the mid-nineties as a declarative approach to specify queries over knowledge bases where the answers to such queries can be concepts : *"Instead of just returning sets of individuals, our queries match concepts and filtered fragments of descriptions"*[24]. As an example, consider the following pattern Q defined as an unknown part X with a certain relationship y with a $university$.

$$Q \equiv X \sqcap \exists y.Univ$$

$$
\begin{array}{rcl}
Professor & \equiv & Pers \sqcap \exists worksIn.Univ \sqcap \exists doctStud.PhDStudent \\
PhDStudent & \equiv & Pers \sqcap \exists studyIn.Univ \sqcap \exists advisor.Professor
\end{array}
$$

$$
Professor(Alan), doctStud(Alan, Alice), PhDStudent(Alice)
$$

Table 1: A DL knowledge base KB_1.

Here, the variable X (called a concept variable) takes its values from a set of possible descriptions while the variable y (called a role variable) takes its values from a set of possible atomic role names. Such a query Q can be evaluated against an individual i or against a concept C. The query semantics is given by the notion of *matching*. For example, the individual *Alan* of the knowledge base KB_1 of Table 1 matches the pattern Q because if we consider the substitution σ such that $\sigma(X) = Pers \sqcap \exists doctStudent.PhDStudent$ and $\sigma(y) = worksIn$ then KB_1 entails the assertion $\sigma(Q)(Alan)$ (i.e., *Alan* is an instance of $\sigma(Q)$). Similarly, the concept $Professor$ matches the pattern Q because by considering the same substitution σ the knowledge base KB_1 entails $Professor \sqsubseteq \sigma(Q)$ (i.e., $Professor$ is subsumed by the concept $\sigma(Q)$). A natural way to give a formal meaning to matching is through the notion of *subsumption*: given a description C and a pattern Q, the matching problem modulo subsumption asks whether there is a variable substitution such that C is subsumed by $\sigma(Q)$ [24].

This thesis focuses on the extension of the notion of patterns to capture recursive queries, a class of queries which is intensively used in many modern application domains such as graph databases and semantic web. However, such an extension is far from being trivial and it requires to revisit the semantics of variables as illustrated below.

Example 1. *Consider the following recursive query specified as a cyclic pattern and evaluated over the knowledge base KB_1 of Table 1:*

$Academic \equiv \quad Pers \sqcap \exists x.Univ \sqcap \exists y.Academic$

Using standard semantics of variable substitution, whatever the considered substitution of the role variables x and y, neither of the concept $Professor$ nor $PhDStudent$ of KB_1 match the pattern Academic. However, the situation becomes different if we exploit a different semantics that enables to refresh the values of the variable x and y when unfolding the pattern Academic. We show below a partial unfolding of Academic where the variable x and y are refreshed (i.e., replaced by new variables) at each iteration over the concept Academic.

$$Academic \equiv \quad Pers \sqcap \exists x_1.Univ \sqcap \exists y_1.\underbrace{(Pers \sqcap \exists x_2.Univ \sqcap \exists y_2.(\ldots))}_{Academic}$$

With such a semantics at hand, it becomes possible to compute a matcher that makes the concept $Professor$ matching the pattern Academic (e.g., take a substitution that maps the first occurrences of x and y to $worksIn$ and $doctStud$ while their second occurrences are respectively mapped to $studyIn$ and $advisor$). The obtained instantiation of the pattern Academic is the following:

$$
\begin{array}{rcl}
\sigma(Academic) \equiv & & Pers \sqcap \exists worksIn.Univ \sqcap \\
& & \exists doctStud.(Pers \sqcap \exists studyIn.Univ \\
& & \sqcap \exists advisor.\sigma(Academic))
\end{array}
$$

This thesis studies the extension of description logics with variables equipped with refreshing semantics in order to capture the expression of recursive structural queries as

cyclic concept patterns. More specifically, we focus on a new description logic, called $\mathcal{EL}_{\mathcal{RV}}$, that extends the description logic $\mathcal{EL}$ with refreshing role variables. Our definition of $\mathcal{EL}_{\mathcal{RV}}$-patterns deviates from the one used in the literature with respect to the following features: (i) our definition of patterns is restricted to role variables while the literature mainly focuses on concept variables, and (ii) we support cyclic pattern definition and allow two different types of semantics for variables (i.e., refreshing and not refreshing semantics).

Viewing patterns as queries, we study three fundamental reasoning problems in the context of the logic $\mathcal{EL}_{\mathcal{RV}}$, namely, matching , weak-subsumption and pattern containment. *Matching* is used as core mechanisms to evaluate patterns over knowledge bases (i.e., computing answers to a query pattern) while *pattern containment* enables to determine when the answers of a pattern are contained in the answers of another pattern whatever the considered knowledge base. On an other side, *weak-subsumption* and *matching* can be viewed as extensions of respectively matching and unification to variables with refreshing semantics. More precisely, we make the following technical contributions:

- We introduce the description logic $\mathcal{EL}_{\mathcal{RV}}$ which extends the logic $\mathcal{EL}$ with refreshing variables. This extension impacts the semantics of pattern which can now produce infinite instances. We differentiate two kinds of instances : regular and irregular instances. A regular instances can be represented with a finite number of descriptions which corresponds to a finite $\mathcal{EL}$-TBox. An instance is irregular if it can not be represented by such a TBox. Subsumption between regular instances in $\mathcal{EL}_{\mathcal{RV}}$ is equivalent to subsumption with regard to the greatest fix-point semantics in $\mathcal{EL}$.

- We define three different reasoning tasks over $\mathcal{EL}_{\mathcal{RV}}$ ontologies. Matching and weak-subsumption are extension of respectively matching and unification, two non-standard reasoning tasks of description logic with non-refreshing variables. Moreover, we introduce a brand new reasoning task: pattern containment. Pattern containment emphasizes on comparing patterns independently of the knowledge base. We demonstrate that if a solution exists then there exists a regular solution.

- We establish a correspondence between $\mathcal{EL}_{\mathcal{RV}}$ and a specific form of variable automata. This form, called description automata, entails all the instances of a pattern (i.e. regular and irregular) in a finite form. Reasoning in $\mathcal{EL}_{\mathcal{RV}}$ is then proven to be equivalent to study variants of simulation between description automata.

- We devise an algorithm to solve the different reasoning tasks resulting in proving that they are EXPTIME-COMPLETE. We demonstrate their correctness leading to an EXPTIME upper bound. The lower bound is obtained by reducing matching (which is a special case of both pattern containment and weak-subsumption) to halting problem of alternating turing machine working on polynomially bounded input which is known to be EXPTIME-COMPLETE [27].

This document presents research conducted in the context of this thesis separated in five chapters. Chapter 1 deals with preliminary notions of trees and description logics. It offers insights of both syntax and semantics of description logics. This chapter ends by presenting terminologies and how to reason over in order to make implicit knowledge explicit with the inference task known as subsumption.

Chapter 2 presents how variables and description logics can be combined. It first discusses the inferences tasks matching and unification that deals with non-refreshing

variables. Introducing $\mathcal{EL}_{\mathcal{RV}}$, a logic extending $\mathcal{EL}$ with refreshing variables, allows to defined pattern queries. Patterns queries are a set of $\mathcal{EL}_{\mathcal{RV}}$-patterns allowing to query the knowledge base through different reasoning mechanisms : matching, weak-subsumption and pattern-containment. Finally, this chapter discusses the existence of regular solutions which can be expressed with finite $\mathcal{EL}$-TBox.

Chapter 3 presents description automata, a class of automata that handles refreshing semantics. We strengthen the link between reasoning in description logics and automata theory by reducing reasoning in $\mathcal{EL}_{\mathcal{RV}}$ to reasoning with description automata. This reduction are based on variants of simulation which allow to solve matching, weak-subsumption and pattern containment.

Chapter 4 solves matching problem in $\mathcal{EL}_{\mathcal{RV}}$. We design an algorithm, Check_Match, that proves matching decidability in $\mathcal{EL}_{\mathcal{RV}}$. This algorithm is inspired of product automata and consists in running simultaneously the two automata to construct a solution to the issued problem.

Chapter 5 emphasizes on discussing decidability for pattern containment and weak-subsumption. In these problems, variables may appear on both sides. Moreover, in case of pattern containment, the domain of variable valuation is infinite. The correct algorithm *Check_Simu* ,inspired of Check_Match, demonstrates that pattern containment and weak-subsumption are decidable. We prove that reasoning in $\mathcal{EL}_{\mathcal{RV}}$ is EXPTIME-COMPLETE

Chapter 1

Preliminaries

In this chapter, we introduce the technical background required for this thesis. We define trees and basics of description logics. The different notations introduced in this chapter are summarized in the Table 1.1.

Symbol	Description
(τ, λ, δ)	Labeled tree with τ the set of nodes, λ the node labeling function and δ the edge labeling function.
$\top$	Universal concept
$\sqcap$	Concept conjunction
$\exists R.C$	Existential restriction
N_A	Set of primitive concepts
N_R	Set of primitive roles
N_{def}	Set of defined concepts
$\mathcal{T}$	Terminology (or TBox) over a signature N_A, N_R
$(\Delta^{\mathcal{I}}, .^{\mathcal{I}})$	Interpretation of a knowledge base
$\sqsubseteq$	Subsumption, i.e. hierarchical relationship between concepts
$\equiv$	Equivalence relationships between concepts

Table 1.1: Notations introduced in Chapter 1

1.1 Trees

We use the following definition of a tree [30]: A tree is a set $\tau \subseteq \mathbb{N}^*$ such that if $xn \in \tau$, for $x \in \mathbb{N}^*$ and $n \in \mathbb{N}$, then $x \in \tau$ and $xm \in \tau$ for all $0 \leq m < n$. The elements of τ represent nodes: the empty word ε is the root of τ, and for each node x, the nodes of the form xn, for $n \in \mathbb{N}$, are children of x. Given a pair of sets S and M, an $\langle S, M \rangle$-labeled tree is a triple (τ, λ, δ), where τ is a tree, $\lambda : \tau \to S$ is a node labeling function that maps each node of τ to an element in S, and $\delta : \tau \times \tau \to M$ is an edge labeling function that maps each edge (x, xn) of τ to an element in M.

We recall now the notion of tree homomorphism. Let $t_1 = (\tau_1, \lambda_1, \delta_1)$ and $t_2 = (\tau_2, \lambda_2, \delta_2)$ be two trees. A homomorphism from t_1 into t_2 is a mapping $Z : \tau_1 \to \tau_2$ such that:

(i) $Z(\varepsilon) = \varepsilon$, and

(ii) $\delta_2(Z(c_1), Z(c_1')) = r$, for all $\delta_1(c_1, c_1') = r$.

A partial tree $(tau', \lambda', \delta')$ of a tree (τ, λ, δ) is such that there exists an homomorphism Z from $(\tau', \lambda', \delta')$ to (τ, λ, δ) and in addition, $\lambda'(i) = \lambda(Z(i))$.

1.2 Description Logic

1.2.1 Introduction

"Description Logics" (DL) denotes a family of knowledge representation formalism that aims to transcript information about a specific domain. The representation offers a formal logic-based semantics used to describe an application domain. The domain is formalized by defining its relevant concepts (its terminology). They are of latter use to characterize the instances of this domain. Representing a domain requires to not only defines entities and their features but also to capture their relationships. The following vocabulary will be used :

- A *concept* is an unary predicate that represents a set of individual having specific features in commons.
- A *role* is a binary relationship between individuals.
- An *individual* is an instance of a concept.

Description logics do not limit themselves to description. Indeed, they offer a wide variety of reasoning mechanisms allowing to infer implicit knowledge from the explicit knowledge represented.

A *Description logic* $\mathcal{L}$ is a tuple made of (N_A,N_R, $\mathcal{C}$). N_A and N_R represents respectively *primitive concepts* and *primitive roles* corresponding to elementary knowledge. Elementary knowledge are by essence knowledge that can not be described and carry meaning by themselves. The *constructors* $\mathcal{C}$ allow to combine elementary knowledge in order to create complex descriptions denoted as $\mathcal{L}$-descriptions. Since constructors formalize how knowledge can be associated, they dictate a logic $\mathcal{L}$ expressiveness.

In the remaining, we use the letters A, B to range over N_A; R, S to range over N_R; and C, D to range over $\mathcal{L}$-concept descriptions (or simply, $\mathcal{L}$-concepts).

The syntax and semantics of the description logic $\mathcal{EL}$ will now be explained.

1.2.2 The Description Logic $\mathcal{EL}$

Syntax.

The Description logic $\mathcal{EL}$ which stands for Existential Language has been provided with three constructors :

- top concept ($\top$),
- conjunction ($\sqcap$) and
- existential restriction ($\exists R.C$).

Given a set of atomic concepts N_A and a set of roles N_R, $\mathcal{EL}$-descriptions are built according to the following syntax rules :

$$C := \top | A | C \sqcap D | \exists R.C$$

where A $\in N_A$, R $\in N_R$ and C,D are $\mathcal{EL}$-concept descriptions.

Even though $\mathcal{EL}$ offers few constructors it has been of interest for some application domains while displaying reasonable complexity for reasoning tasks. For instance, it can be used to define biomedical ontologies like Snomed CT [28] or the Gene Ontology. These ontologies can be seen as $\mathcal{EL}$-TBoxes. Moreover, an extension of $\mathcal{EL}$ became also a standard for a subset of OWL 2. OWL stands for the Web Ontology Language which can be used to exploit reasoning procedures [31].

Example 2. *To give examples of what can be expressed in $\mathcal{EL}$, we suppose that University and Person are atomic concepts (i.e. $\{University, Person\} \subseteq N_A$) and study and teach are atomic roles (i.e. $\{study, teach\} \subseteq N_R$). Intuitively, $Person \sqcap \exists study.University$ and $Person \sqcap \exists teach.University$ are concepts describing respectively students and professors of a university. Existential restriction can be weakened by using the concept $\top$ to describe $Person \sqcap \exists teach.\top$ which defines a teacher as a person who is teaching at any educational level.*

Now that the syntax for each constructors has been provided, their semantics will be discussed.

Semantics.

The semantics of $\mathcal{EL}$ is formalized in terms of *interpretation*. An interpretation $\mathcal{I}$ is a pair $(\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ where $\Delta^{\mathcal{I}}$ is a non-empty set called the *domain* and $.^{\mathcal{I}}$ is an interpretation function that assigns binary relations on $\Delta^{\mathcal{I}}$ to role names and subsets of $\Delta^{\mathcal{I}}$ to $\mathcal{EL}$-concepts as shown in the semantics column of the following Table in 1.2.

Name	Syntax	Semantics
Top concept	$\top$	$\Delta^{\mathcal{I}}$
Concept name	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
Role name	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
Conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Existential restriction	$\exists R.C$	$\{ a \in \Delta^{\mathcal{I}} \mid \exists b \in C^{\mathcal{I}}.(a,b) \in R^{\mathcal{I}}\}$

Table 1.2: Syntax and semantics of $\mathcal{EL}$ constructors

We say that two concepts C, D are equivalent, and write $C \equiv D$, if $C^{\mathcal{I}} = D^{\mathcal{I}}$ for all interpretations $\mathcal{I}$. A concept C is less general than a concept D, noted, $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. This relation is called the *subsumption relationship*. As expected, if $C \sqsubseteq D$ and $D \sqsubseteq C$ then we have $C \equiv D$.

In $\mathcal{EL}$, it is possible to construct $\mathcal{EL}$-description trees based on $\mathcal{EL}$-description. The root of the tree corresponds to the $\mathcal{EL}$-description. Each node of the tree corresponds to

a concept, label of a node is made of primitive concepts appearing without existential restriction. For each existential restriction there is an edge labeled by the corresponding role to a new node representing the reached concept. A subsumption relationship, $C \sqsubseteq D$ is characterized by a homomorphism from the tree of D into the tree of C [13].

A DL Knowledge Base (KB) is composed of two components, called TBox $\mathcal{T}$ and ABox $\mathcal{A}$, containing respectively general assertions describing relevant concepts and specific assertions about individuals and relationships among them.

Terminologies.

Concept descriptions enable to describe class of objects. In order to specify how concepts relate to each other, description logics make use of terminological axioms. A *terminological axiom* can take the form of :

- An equality $C \equiv D$ or
- An inclusion $C \sqsubseteq D$.

Semantics of axioms is also determined through an interpretation $\mathcal{I}$. As expected, a terminological axiom $C \sqsubseteq D$ (resp. $C \equiv D$) is *satisfied* by an interpretation $\mathcal{I}$, if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ (resp. $C^{\mathcal{I}} = D^{\mathcal{I}}$). Let $\mathcal{T}$ be a set of terminological axioms, $\mathcal{I}$ satisfies $\mathcal{T}$ if and only if $\mathcal{I}$ satisfies each axiom of $\mathcal{T}$. Two axioms (or two sets of axioms) are equivalent if they have the same models.

Different kinds of TBoxes are considered regarding properties of their axioms' set. From the less expressive to the most expressive, we have :

- **Empty TBoxes** : This class refers to TBoxes with no axiom.
- **Simple TBoxes** : TBoxes of this class are made of a set of equality axioms denoted as *concept definitions*. A concept definition is an equality axiom where the left-hand side is a concept name.

  Example 3. *The following definitions form a simple TBox $\mathcal{T}$.*

  $$France \equiv Country \sqcap \exists hasCapital.Paris$$
  $$FrenchUniversity \equiv \exists locate.France \sqcap University$$
  $$Student \equiv \exists study.University \sqcap Person$$
  $$Teacher \equiv \exists teach.University \sqcap Person$$

- **General concept inclusion (GCI) TBoxes** : TBoxes of this class are made of a set of axioms without any restriction.

  Example 4. *Let us consider the CGI made of the following definition :*

  $$FrenchUniversity \equiv \exists locate.France \sqcap University$$
  $$Teacher \sqsubseteq \exists teach.University \sqcap Person$$
  $$Female \sqcap \exists study.University \sqsubseteq \exists study.University \sqcap Person$$

Note that any equality axiom can be transformed into two inclusion axioms. For example $FrenchUniversity \equiv \exists locate.France \sqcap University$ *would become* $\{FrenchUniversity \sqsubseteq \exists locate.France \sqcap University, \exists locate.France \sqcap University \sqsubseteq FrenchUniversity\}$. *In order to deal with a unique kind of axiom, this transformation is handful.*

The remaining will emphasize over simple *TBox* $\mathcal{T}$ which will be of later interest. This class corresponds to a set of concept definitions of the form $P \equiv C$, with $P \in N_{def}$ and C an $\mathcal{EL}$-concept such that no P appears more than once on the left-hand side of a definition in $\mathcal{T}$. Concept names appearing on the left-hand side of a definition are called *defined concepts*, and denoted by the set N_{def}. All the other concepts occurring in $\mathcal{T}$ are called *atomic concepts* and are denoted by the set N_A.

Example 5. *Back to Example 3, we have :*

- $N_A = \{Person, University, Country, Paris\}$
- $N_R = \{teach, locate, study, hasCapital\}$
- $N_{def} = \{FrenchUniversity, Student, Teacher, France\}$

An interpretation $\mathcal{I}$ is a model of $\mathcal{T}$ if and only if for all definitions $A \equiv C \in \mathcal{T}$ we have $P^{\mathcal{I}} = C^{\mathcal{I}}$. We say that C is subsumed by D w.r.t. $\mathcal{T}$, written $C \sqsubseteq_{\mathcal{T}} D$, iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ holds for every model $\mathcal{I}$ of $\mathcal{T}$.

The semantics introduced previously is denoted as the descriptive semantics. However, terminologies may require to represent cyclic dependencies between defined concepts, i.e., a definition of an $\mathcal{EL}$-concept P directly or indirectly refers to P itself.

Example 6. *Example 3 displays a TBox that can be qualified as acyclic. However, if we add the following definitions :*

$$Doctor \equiv Person \sqcap \exists getPhDIn.University \sqcap \exists formerly.PhDStudent$$

$$PhDStudent \equiv Student \sqcap \exists supervisedBy.Doctor$$

Doctor appears in the definition of PhDStudent and PhDStudent appears in the definition of Doctor. Therefore, these definitions creates an indirect cycle. Thus illustrating cyclic TBoxes.

When cycles are involved, this semantics may be limiting [34] and other semantics could be used, in particular greatest fix-point and least fix-point semantics [5]. The general idea is to extend a primitive interpretation $\mathcal{J}$ (i.e. an interpretation of primitive concepts and primitive roles) to the defined concepts of the TBox. An interpretation $\mathcal{I}$ is based on $\mathcal{J}$ if it has the same domain as $\mathcal{J}$ (i.e. $\Delta^{\mathcal{I}} = \Delta^{\mathcal{J}}$ and its interpretation function coincides with the one of $\mathcal{J}$ on N_A and N_R (i.e. $A^{\mathcal{I}} = A^{\mathcal{J}}$ and $R^{\mathcal{I}} = R^{\mathcal{J}}$ $\forall A \in N_A$ and $\forall R \in N_R$). Given two interpretations $\mathcal{I}_1$ and $\mathcal{I}_2$ based on the same interpretation $\mathcal{J}$, we have $\mathcal{I}_1 \preceq_{\mathcal{J}} \mathcal{I}_2$ if and only if $C^{\mathcal{I}_1} \subseteq C^{\mathcal{I}_2}, \forall C \in N_{def}$. It has been demonstrated in [5] that there exists a unique model of $\mathcal{I}$ of $\mathcal{T}$ such that :

1. $\mathcal{I}$ is based on the primitive interpretation $\mathcal{J}$, and

2. $\mathcal{I}' \preceq_{\mathcal{J}} \mathcal{I}, \forall \mathcal{I}'$ of $\mathcal{T}$ based on $\mathcal{J}$

Such a model is defined as a gfp-model of $\mathcal{T}$. Lfp-model can be defined similarly by exchanging $\mathcal{I}'$ and $\mathcal{I}$ roles in (2). In order to illustrate and to develop benefits of considering different semantics, we will use the following Example 7 from [5].

Example 7. *Let $\mathcal{T} = \{INode \equiv Node \sqcap \exists edge.INode\}$. This TBox contains a primitive concept, Node and a primitive role, edge. The only definition INode is cyclic and represents node involved in an infinite path. Let consider the primitive interpretation $\mathcal{J}$ defined as follows :*

- $\Delta^{\mathcal{J}} = \{m_0, m_1, m_2, \ldots\} \cup \{n_0\}$
- $Node^{\mathcal{J}} = \Delta^{\mathcal{J}}$
- $edge^{\mathcal{J}} = \{(m_i, m_{i+1}) | i \geq 0\} \cup \{(n_0, n_0)\}$

Nodes can be involved in an infinite path if it belongs to a cycle (n_0) or if it is involved in an infinite path (m_i). As a consequence, there are four ways to extend $\mathcal{J}$.

1. $\{m_0, m_1, m_2, \ldots\} \cup \{n_0\}$
2. $\{m_0, m_1, m_2, \ldots\}$
3. $\{n_0\}$
4. $\emptyset$

All of this models are models for the descriptive semantics. The last possibility, which represents the least-fix point model, is not relevant w.r.t the aimed definition. The first possibility, which is the greatest fix-point model, captures exactly the semantics of INode we are aiming for. Indeed the remaining models ignores either cycles (2) or infinite paths (3).
Greatest-fix point semantics is not always the best choice regarding concepts involved. Let consider the following concepts : $Tiger = Animal \sqcap \exists Parent.Tiger$ and $Lion = Animal \sqcap \exists Parent.Lion$ From a gfp point of view, Tiger and Lion will always be interpreted the same way. The descriptive semantics on the other hand may interpret them differently which seems more appropriate.

Those definitions can easily be transferred to CGI by using semantics of equality and inclusion axioms. Moreover in case of an acyclic TBoxes, descriptive, greatest fix-point and least fix-point semantics are equivalent.

Normal Form

Alongside TBoxes, normal form of concept descriptions have been introduced. Normal form of an $\mathcal{EL}$-description is a conjunction of $\mathcal{EL}$-atoms. An $\mathcal{EL}$-atom is either a primitive concept (i.e. an element of N_A) or an existential restriction of the form $\exists R.C$ with $C \in N_{def}$. Concretely, we say that a description $C \equiv D$ is normalized if D is of the form :

$$A_0 \sqcap \ldots \sqcap A_n \sqcap \exists r_0.B_0 \sqcap \ldots \sqcap \exists r_m.B_m$$

where $A_0, ..., A_n \in N_A$, $r_0, ...r_m \in N_R$ and $B_0, ..., B_m \in N_{def}$.

Any $\mathcal{EL}$-description can be transformed into a normalized form. In order to achieve the normalization process, definitions may be added to the terminology. This process has proven to be polynomial in [5]. As stated in this report, the resulting TBox may change w.r.t to the considered semantics. Normal forms are often used to solve reasoning tasks. Indeed, they define a standard form for description which simplifies the algorithm without damaging the complexity since this transformation is polynomial for $\mathcal{EL}$.

Example 8. *Back to Example 3, France and FrenchUniversity are concepts in normal form. However, concepts France, Student and Teacher are not normalized since a primitive concept appears after an existential restriction. As a consequence, two new defined concept C_1 and C_2 are created. Their definition will respectively correspond to the primitive concept Paris and University. However, C_1 and C_2 do fulfill the requirements of normal form. The normalized form of this TBox is :*

$$France \equiv Country \sqcap \exists hasCapital.C_1$$
$$FrenchUniversity \equiv \exists locate.France \sqcap University$$
$$Student \equiv \exists study.C_2 \sqcap Person$$
$$Teacher \equiv \exists teach.C_2 \sqcap Person$$
$$C_1 \equiv Paris$$
$$C_2 \equiv University$$

Reasoning in $\mathcal{EL}$ w.r.t a Terminology

Reasoning mechanisms unleash the potential of a terminology. Indeed, reasoning mechanisms allow to infer knowledge that is not explicitly expressed. There exists many reasoning procedure which can be qualified as standard like satisfiability or even non-standard procedures like computing the least common subsumers, concept difference [3, 4, 10, 13, 37]. However, we focus in this thesis on the subsumption inference problems and its non-standard extensions which will be discussed later. This problem is named after the relationship subsumption. It aims to statute whether a concept C is more general than a concept D or not.

Definition 1 (Inference Problems)**.** ***Subsumption*** *: Let $\mathcal{T}$ be a terminology and C,D two defined concepts of $\mathcal{T}$*

- *C is subsumed by D w.r.t. the descriptive semantic ($C \sqsubseteq_{\mathcal{T}} D$) if and only if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{T}$*
- *C is subsumed by D w.r.t. the greatest fix-point semantic ($C \sqsubseteq_{\mathcal{T},gfp} D$) if and only if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for every greatest fix-point model $\mathcal{I}$ of $\mathcal{T}$*
- *C is subsumed by D w.r.t. the least fix-point semantic ($C \sqsubseteq_{\mathcal{T},lfp} D$) if and only if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for every least fix-point model $\mathcal{I}$ of $\mathcal{T}$*

Subsumption in $\mathcal{EL}$ is proven to be polynomial and that, with or without TBox. In many cases, extensions of $\mathcal{EL}$ with more constructors leads to EXPTIME-COMPLETE [7, 25]. However, compared to other logics with few constructors like $\mathcal{FL}_0$, it is an interesting feature. Indeed, $\mathcal{FL}_0$ and its derived logics suffered from a blow-up of complexity while considering TBoxes. Subsumption in $\mathcal{FL}_0$ is co-NP-COMPLETE [35], PSPACE-COMPLETE [32] and EXPTIME-COMPLETE [10] for respectively acyclic, cyclic and general TBoxes.

It is worth to note that $\mathcal{EL}$-concept descriptions of simple TBoxes can be viewed as directed labeled graphs. This representation unravel another characterization of subsumption in $\mathcal{EL}$. Subsumption between $\mathcal{EL}$-concept descriptions can be reduced to the existence of simulation between the graphs of the concept descriptions. In case of cyclic terminologies, there exists additional properties in order to support the different descriptive and fix-point semantics [5].

1.3 Conclusion of Chapter 1

This chapter covers the notion of trees which are seen as a tuple (τ, λ, δ) as well as the definition of homomorphism between trees. This relationship plays a key-role since it is linked with subsumption in description logics.

This chapter also emphasizes on the formal preliminaries of description logics using $\mathcal{EL}$, which will be of later interest, as supporting example. It has introduced the syntax of a description logic as well as the semantics associated with. Considering a terminology impacts the semantics since cycle may requires different care. There are three main semantics known as descriptive semantics, greatest fix-point semantics and least fix-point semantics. It ends with the definition of the subsumption inference tasks named after the relationship between two concepts. Next chapters shows how logics can be enhanced with variables and the consequence over the subsumption inference mechanism.

Chapter 2

Refreshing Semantics in $\mathcal{EL}$

This chapter presents how variables and description logics can be combined. Under the non-refreshing semantics, this combination notably led to the inference tasks : matching [1, 9, 12, 15] and unification [6, 14, 16]. After briefly exposing the benefits of considering variables with a refreshing semantics, $\mathcal{EL}_{\mathcal{RV}}$ is defined. $\mathcal{EL}_{\mathcal{RV}}$ extends $\mathcal{EL}$ by allowing refreshing variables as well as pattern queries. Patterns queries are a set of $\mathcal{EL}_{\mathcal{RV}}$-patterns allowing to query the knowledge base through different reasoning mechanisms : matching, weak-subsumption and pattern-containment. Finally, this chapter discusses the existence of regular solution which can be expressed with finite $\mathcal{EL}$-TBox. Notation introduced in this chapter are summarized in Table 2.1

Symbol	**Description**
$\mathcal{V}$	Set of variables
$N_{\mathcal{X}}^{R}$	Set of role variables
N_{T}	Set of role variables and primitive roles
$N_{\mathcal{X}}^{C}$	Set of concept variables
N_{V_R}	Set of refreshing variables
N_{V_N}	Set of non-refreshing variables
P	Pattern
$u(P)$	Unfolded pattern
$\sigma(P)$	Unfolded pattern instance
$P \sqsubseteq^{?}_{\mathcal{T}} D$	Matching problem w.r.t $\mathcal{T}$
$P \sqsubseteq^{?}_{\mathcal{T}} Q$	Weak-subsumption problem w.r.t $\mathcal{T}$
$P \sqsubseteq^{?} Q$	Pattern containment problem

Table 2.1: Notations introduced in Chapter 2

2.1 TBoxes with Variables

2.1.1 Introduction

Recently, expressiveness of description logics have been pushed forward with the introduction of variables. The set $N_{\mathcal{X}}$ complements the sets of concept names, N_A and N_{def}, as well as roles N_R. This set is made of both concept and role variables leading to two subsets $N_{\mathcal{X}}^R$ which denotes the set of variable replacing roles names and, $N_{\mathcal{X}}^C$ corresponding to variables standing for concept descriptions. Like any role or concept, variables can be used by constructors to build description that may contain variables. In this context, those descriptions are denoted as *patterns*. A *ground description* opposes this definition in the sense that it is a description without any variable.

Example 9.

$$Doctor \equiv Person \sqcap \exists getPhDIn.Univ \sqcap \exists formerly.PhDStudent$$

$$PhDStudent \equiv Person \sqcap \exists studyIn.Univ \sqcap \exists supervisedBy.Doctor$$

$$Pattern \equiv X \sqcap \exists y.Univ$$

In this example, Doctor and PhDStudent are ground description since they do not contain any variables. However, Academic possesses two variables X a concept variable and y a role variable.

An $\mathcal{EL}$-Pattern is in normal form if:

$$P \equiv V_1 \sqcap ... \sqcap V_n \sqcap \exists r_1.B_1 \sqcap ... \sqcap r_m.B_m$$

where $V_0, ..., V_n \in N_A \cup N_{\mathcal{X}}^C$, $r_0, ..., r_m \in N_R \cup N_{\mathcal{X}}^R$ and $B_0, ..., B_m \in N_{def} \cup N_{\mathcal{X}}^C$.

Even if the syntax for variables is clear, their semantics remains to be defined. A natural way to define semantics is through *variable substitutions*. Formally, a substitution function σ is a mapping from $N_{\mathcal{X}}$ into the set of $\mathcal{EL}$-concept description and primitive roles. It allows to map role (resp. concept) variables into roles (resp. concept descriptions). Obviously σ is extended to element of N_A and N_R by considering identity. Substitution functions are applied directly on $\mathcal{EL}$-patterns using the following rules :

- $\sigma(A) = A$ if $A \in N_A \cup \{\top\}$
- $\sigma(C \sqcap D) = \sigma(C) \sqcap \sigma(D)$ with C,D two $\mathcal{EL}$-patterns.
- $\sigma(\exists R.C) = \exists \sigma(R).\sigma(C)$ with $\sigma(R) = R$ if $R \in N_R$

Example 10. *Let consider σ_1, σ_2 such that :*

- $\sigma_1(X) = Person \sqcap \exists doctStudent.PhDStudent$ *and* $\sigma_1(y) = worksIn.$
- $\sigma_2(X) = Person$ *and* $\sigma_2(y) = studyIn$.

Applying σ_1 and σ_2 to the previously introduced concepts results in :

$$\sigma_1(Doctor) \equiv \sigma_2(Doctor) \equiv Doctor$$

$$\sigma_1(PhDStudent) \equiv \sigma_2(PhDStudent) \equiv PhdStudent$$
$$\sigma_1(Pattern) \equiv \underbrace{Person \sqcap \exists doctStudent.PhDStudent}_{\sigma_1(X)} \sqcap \exists \underbrace{worksIn}_{\sigma_1(y)}.Univ$$
$$\sigma_2(Pattern) \equiv \underbrace{Person}_{\sigma_2(X)} \sqcap \exists \underbrace{studyIn}_{\sigma_2(y)}.Univ$$

Subsumption changed in order to handle patterns and their variables. It evolved in two, interesting and non-standard, reasoning tasks known as *matching* and *unification*.

The matching problem aims to compare a pattern P and a ground description C w.r.t to the subsumption relationship. It consists in looking for a substitution σ such that the resulting concept $\sigma(P)$ fulfills a subsumption relationship with C. If such a substitution exists, it is considered as a solution and called a *matcher*. State of the art defines variations of the matching problem which are known as matching problem modulo equivalence and matching problem modulo subsumption.

Definition 2 (Matching Problem)**.** *Let $\mathcal{T}$ be a TBox, P an $\mathcal{EL}$-pattern and C an $\mathcal{EL}$-ground-description.*

- *A matching problem modulo equivalence w.r.t. a TBox $\mathcal{T}$ is an equation of the form $C \equiv^?_{\mathcal{T}} P$. It has a solution if there exists a substitution σ such that $C \equiv_{\mathcal{T}} \sigma(P)$*
- *A matching problem modulo subsumption w.r.t. a TBox $\mathcal{T}$ is an equation of the form $C \sqsubseteq^?_{\mathcal{T}} P$. It has a solution if there exists a substitution σ such that $C \sqsubseteq_{\mathcal{T}} \sigma(P)$*

Since matching aims to compare a ground-description and a pattern, matching problems modulo subsumption differs depending on the side of the pattern [1]. When the pattern is on the left, we talk about right-ground matching problem. On the other hand, if the pattern is on the right, it is a left-ground matching problem. Note that since $\{C \sqsubseteq^? P\} \equiv \{C \sqcap P \equiv^? C\}$, any left-ground matching problem modulo subsumption can be resumed to a matching problem modulo equivalence.

However, in the case of a right-ground matching problem then we have $\{P \sqsubseteq^? C\} \equiv \{P \sqcap C \equiv^? P\}$. The resulting equation modulo equivalence involve a pattern in both sides which is exactly the definition of unification.

Definition 3 (Unification Problem)**.** *Let $\mathcal{T}$ be a TBox and P, Q two $\mathcal{EL}$-patterns.*

- *A unification problem modulo equivalence w.r.t. a TBox $\mathcal{T}$ is an equation of the form $P \equiv^?_{\mathcal{T}} Q$. It has a solution if there exists a substitution σ such that $\sigma(P) \equiv_{\mathcal{T}} \sigma(Q)$*
- *A unification problem modulo subsumption w.r.t. a TBox $\mathcal{T}$ is an equation of the form $P \sqsubseteq^?_{\mathcal{T}} Q$. It has a solution if there exists a substitution σ such that $\sigma(P) \sqsubseteq_{\mathcal{T}} \sigma(Q)$*

For a unification problem [6, 14, 16], the solutions are called *unifiers*. Since we have $\{P \sqsubseteq^? Q\} \equiv \{P \sqcap Q \equiv^? P\}$, any unification problem modulo equivalence has a corresponding unification problem modulo subsumption. Although being recent, these problems have been attracting attention of many research for different settings. The next section focuses on known results for this reasoning tasks with regard to the state of the art.

2.1.2 Reasoning with Variables

Matching and unification problems aroused in description logics in late 90s. Those problem have shown proficiency to filter out the unimportant aspects of large concept descriptions appearing in knowledge base [23]. They can also be used as a tool in databases to detect redundancies [16] or support integration [22]. All of those applications exploit the capacity of a pattern to express a not completely specified form.

The literature around matching and unification problems mainly focuses on concept variables. Indeed, role variables are little considered because in logics without role constructors, solving the problem can be done by enumerating possibilities [16].

So far, two families of logics have been widely investigated known as $\mathcal{FL}_0$ based logics and $\mathcal{EL}$ based logics. Note that both of this family offer conjunction and either existential restriction ($\exists R.C$) or value restriction ($\forall R.C$). As a direct consequence, concept variables can be substitute by an infinity of possibilities. Thus preventing from enumerating potential solutions. $\mathcal{FL}_0$ is a logic that allows for concept conjunction $\sqcap$, value restriction $\forall R.C$ and universal concept $\top$. Among the extensions of $\mathcal{FL}_0$ that have been of interest, there are $\mathcal{FL}_\bot$, $\mathcal{FL}_\neg$ and $\mathcal{ALN}$. $\mathcal{FL}_0$ is extended by successively enriching its constructors to form these logics. Starting with $\mathcal{FL}_\bot$ that unlocks the unsatisfiable concept ($\bot$). Then $\mathcal{FL}_\neg$ allows $\bot$ and atomic negation ($\neg$) and finally $\mathcal{ALN}$ extends $\mathcal{FL}_0$ with $\bot$, atomic negation ($\neg$), unqualified number restriction ($\leq n.R$ —$\geq n.R$). On the other hand $\mathcal{ALE}$ offers limited existential restriction ($\exists R.C$) instead of unqualified number restriction.

Matching

Research on matching problems in $\mathcal{FL}_0$ led to prove that this problem is polynomial without TBox [1] as well as in its extensions except $\mathcal{ALE}$ which will be discussed below. Those results have been achieved by exploiting the reduction, presented in [2], of subsumption to language inclusion. Combining role variables and concept variables for $\mathcal{FL}_0$ without TBox increases the complexity and is NP-COMPLETE [16].

Considering a general TBox blows up the complexity to EXPTIME-COMPLETE [9]. This results have been proved by extending the previous method based on automata theory and language inclusion. In their work, the authors introduced restricted TBoxes which offers a reduced complexity for this reasoning task. Those TBoxes are called forward Tboxes, i.e. TBoxes where the role depth on the left-hand side of a GCI is not larger than the role depth on the right-hand side. Forward TBoxes bears the advantage to lower the complexity to PSPACE-COMPLETE in this case. Similarly, considering backward Tboxes, i.e. TBoxes where the role depth on the right-hand side of a GCI is not larger than the role depth on the left-hand side, bears the same property.

$\mathcal{EL}$-matching problem has a higher complexity by achieving NP-COMPLETE even without TBoxes. Baader et al. [12] demonstrated it by reducing matching into finding a homomorphism between description trees. The same method has been applied to $\mathcal{ALE}$ leading to the same complexity. $\mathcal{EL}$ does not suffer any blow up of complexity when considering a general TBox. The approach proposed in [15] exploits structural subsumption through a goal-oriented algorithm. Complexity results regarding matching problem are summarized in Table 2.2.

Description Logic	TBox	Complexity
$\mathcal{FL}_0$	$\emptyset$	PTIME [1]
	General Forward Tbox	PSPACE-COMPLETE [9]
	General Backward Tbox	PSPACE-COMPLETE [9]
	General Tbox	EXPTIME-COMPLETE [9]
$\mathcal{FL}_\bot, \mathcal{FL}_\neg, \mathcal{ALN}$	$\emptyset$	PTIME [1]
	General Tbox	open
$\mathcal{EL}$	$\emptyset$	NP-COMPLETE [12]
	Acyclic Tbox	NP-COMPLETE [12]
	General Tbox	NP-COMPLETE [15]
$\mathcal{ALE}$	$\emptyset$	NP-COMPLETE [12]

Table 2.2: Complexity results for matching

Unification

Unification considers equations of the form $P \sqsubseteq^? Q$ where both, P and Q are patterns. The major result for $\mathcal{FL}_0$ is that unification of $\mathcal{FL}_0$-concept terms (i.e. w.r.t to empty Tbox) is EXPTIME-COMPLETE [16]. The authors used a similar approach to the one used to solve matching (i.e. reducing the problem to language inclusion problems). The extension of unification in $\mathcal{FL}_0$ to general TBoxes is still an open-problem. $\mathcal{FL}_\bot$ has also been investigated leading to unification w.r.t to an empty TBox to be in EXPTIME [33]. The method employed consists in eliminating $\bot$ from a given problem leading to a $\mathcal{FL}_0$ problem that can be solved by known methods.

Likewise $\mathcal{EL}$-matching, $\mathcal{EL}$-unification is NP-COMPLETE for empty TBox [14]. Authors demonstrated that a local unifier could be defined and necessarily exists if a unifier exists. Then a goal-oriented algorithm has been designed to decide if such a local unifier can be computed. Research towards unification with general TBoxes are more advanced than those for $\mathcal{FL}_0$. Indeed, even if $\mathcal{EL}$-unification w.r.t general TBoxes is still an open problem, in [6] the authors achieved NP-COMPLETE for cyclic restricted TBoxes. The restriction prohibits cycle of the form $A \sqsubseteq \exists R_1.\exists R_2. \ldots \exists R_n.A$. Such a cycle contradicts the definition of local unifier thus making their algorithm incomplete.

In an attempt to lower unification complexity variants of unification have been investigated. These variants are known as restricted unification [11, 17]. It can be either syntactically restricted (i.e. limiting the role depth of concepts) or semantically restricted (i.e. limiting the length of interpretation). Such restrictions are derived from observations when it comes to choose a unifier among the proposed solutions. They allow to obtain a resulting concept close to the shape of those in the knowledge base. Regarding complexity of these reasoning tasks in $\mathcal{FL}_0$, let k be the limit depth allowed. Then, it is shown

in [11] that it remains in EXPTIME if k is encoded in binary but drops to PSPACE if k is encoded in unitary. Unfortunately, none of the result followed for $\mathcal{EL}$, it remains NP-COMPLETE [17] . The different results for concept variables are summarized in Table 2.3.

Description Logic	TBox	Complexity
$\mathcal{FL}_0$	$\emptyset$	EXPTIME-COMPLETE [16]
	General Tbox	open
$\mathcal{FL}_\bot$	$\emptyset$	in EXPTIME[33]
$\mathcal{FL}_\neg, \mathcal{ALN}$	$\emptyset$	open
$\mathcal{EL}$	$\emptyset$	NP-COMPLETE [14]
	Acyclic Tbox	NP-COMPLETE [14]
	Cyclic Restricted	NP-COMPLETE [6]
	General Tbox	open
$\mathcal{ALE}$	$\emptyset$	open

Table 2.3: Complexity results for unification

In the context of description logic with variables, we will introduce variables with refreshing semantics in $\mathcal{EL}$ leading to the definition of the logic $\mathcal{EL}_{\mathcal{RV}}$. Matching and unification problems will be extended in this scope and a new reasoning task known as pattern containment will be defined.

2.2 Definition of $\mathcal{EL}_{\mathcal{RV}}$

2.2.1 Differences Between Variable Semantics

This section aims to illustrate the potential of introducing refreshing semantics in description logics using $\mathcal{EL}$ as example. Consequently, we will save technical aspects for later sections and emphasize here on presenting how valuable it can be. The example is based on the terminology containing these two definitions :

$$Doctor \equiv Person \sqcap \exists getPhDIn.Univ \sqcap \exists formerly.PhDStudent$$

$$PhDStudent \equiv Person \sqcap \exists studyIn.Univ \sqcap \exists supervisedBy.Doctor$$

To this simple TBox, the pattern *Academic* is added and defined as :

$$Academic \equiv Person \sqcap \exists x.University \sqcap \exists y.Academic$$

with x, y two role variables.

The matching problem $Doctor \sqsubseteq^?_{gfp,\mathcal{T}} Academic$ does not have any solution from a state of the art point of view. Refreshing semantics increases substitution possibilities

with, in particular, θ such that the resulting concept $\theta(Academic)$ is a matcher :

$$\begin{array}{lcl} Doctor & \equiv & Person \sqcap \exists getPhDIn.Univ \sqcap \exists formerly.(\\ & & Person \sqcap \exists studyIn.Univ \sqcap \exists supervisedBy.(\ldots)) \\ \sqsubseteq_{gfp,\mathcal{T}} & & \\ \theta(Academic) & \equiv & Person \sqcap \exists \underbrace{getPhdIn}_{\theta(x)}.Univ \sqcap \exists \underbrace{formerly}_{\theta(y)}(\\ & & Person \sqcap \exists \underbrace{studyIn}_{\theta(x)}.Univ \sqcap \exists \underbrace{supervisedBy}_{\theta(y)}((\ldots)) \end{array}$$

To achieve this solution, the variable x (resp. y) is bound to $getPhdIn$ (resp. $formerly$). Once Academic is unfold, the variables x and y are released from their respective bound. Variables x and y are now ready to be bound to a new value. In $\theta(Academic)$, the released x (resp. y) is bound to $studyIn$ (resp. $supervisedBy$). Reaching $Academic$ once again implies to unfold $Academic$ which releases variables x and y and so on. The substitution θ is then defined as the substitution that alternates between $\{x = getPhDIn, y = formerly\}$ and $\{x = studyIn, y = supervisedBy\}$.

A *refreshing variable* is a variable that has the ability to be released under some conditions in order to receive a new assignment. *Refreshing semantics* is the semantics that allows to use refreshing variables. Comparatively to non-refreshing semantics, refreshing semantics offers many more substitutions. Indeed, in the running problem, the set of primitive roles contains 4 roles ($N_R = \{getPhDIn, formerly, studyIn, supervisedBy\}$). It means that there are 4 possibilities for x and 4 possibilities for y leading to a total of 16 possible substitutions under the non-refreshing semantics. On the other hand, refreshing semantics will release variables with each release associated to its own bound. In other words, we deal with an infinite number of variables which makes the set of possible substitutions infinite. This new possibilities may be solutions to reasoning problems as shown in the example.

Research conducted in this thesis differ from previous works in particular regarding variable semantics. Indeed, semantics for variables in description logics can be qualified as *non-refreshing* semantics. In our framework - role variables with refreshing and non-refreshing semantics -, we have not only considered known problems such as matching and unification but also a new reasoning task denoted as pattern containment. Pattern containment aims to compare patterns w.r.t to their substitutions whatever the considered TBox.

The next section formally introduces refreshing semantics and its associated reasoning tasks in the scope of $\mathcal{EL}_{\mathcal{RV}}$.

2.2.2 The $\mathcal{EL}_{\mathcal{RV}}$ Description Logic

We consider pattern queries expressed using the description logic $\mathcal{EL}$ extended with refreshing role variables. The obtained logic, called $\mathcal{EL}_{\mathcal{RV}}$, is introduced below. An $\mathcal{EL}_{\mathcal{RV}}$-signature is a pair $\Sigma = (N_C, N_T)$, where N_C is the set of concept names (i.e. $N_A \cup N_{def}$ and $N_T = N_R \cup \mathcal{V}$ the set of role terms. A role term $t \in N_T$ is either a role name (when $t \in N_R$) or a variable (when $t \in \mathcal{V}$). We consider the set of variables $\mathcal{V} = N_{V_R} \cup N_{V_N}$ as made of two disjoint sets of variables: N_{V_R} the set of *refreshing* variables and N_{V_N} the set of *non refreshing* variables. The sets N_C, N_R, N_{V_R} and N_{V_N} are pairwise disjoint.

The description logic $\mathcal{EL}_{\mathcal{RV}}$ extends the logic $\mathcal{EL}$ with role variables. Given a signature $\Sigma = (N_C, N_T)$, $\mathcal{EL}_{\mathcal{RV}}$-concept descriptions are built similarly to $\mathcal{EL}$ concepts while using roles terms instead of only role names. The concept *Academic* given in Example 1 is an example of an $\mathcal{EL}_{\mathcal{RV}}$-definition with x and y refreshing variables. The particular case of $\mathcal{EL}_{\mathcal{RV}}$-concepts without variables (called ground concepts) are $\mathcal{EL}$-concepts.

An $\mathcal{EL}_{\mathcal{RV}}$-TBox is a set of $\mathcal{EL}_{\mathcal{RV}}$-concept definitions. We present now the notion of normalized $\mathcal{EL}_{\mathcal{RV}}$-TBoxes. Let $\Sigma = (N_C, N_T)$, with $N_T = N_R \cup \mathcal{V}$ and $\mathcal{V} = N_{V_R} \cup N_{V_N}$, be an $\mathcal{EL}_{\mathcal{RV}}$-signature and let $\mathcal{T}$ be an $\mathcal{EL}_{\mathcal{RV}}$-TBox over the signature Σ. We say that $\mathcal{T}$ is normalized iff $C \equiv D \in \mathcal{T}$ implies that D is of the form:

$$A_0 \sqcap ... \sqcap A_n \sqcap \exists r_0.B_0 \sqcap ... \sqcap \exists r_m.B_m$$

where $A_0, ..., A_n \in N_A$, $r_0, ...r_m \in N_T$ and $B_0, ..., B_m \in N_{def}$.

In the sequel, we assume that the $\mathcal{EL}_{\mathcal{RV}}$-TBoxes are normalized. An $\mathcal{EL}_{\mathcal{RV}}$ pattern P is given by an $\mathcal{EL}_{\mathcal{RV}}$-TBox, noted $\mathcal{T}^P$, which contains a definition of P.

Example 11. *Assume a signature $\Sigma = (N_C, N_T)$ and let $x \in N_{V_R}$ and $y \in N_{V_N}$ be respectively a refreshing and a non-refreshing variable. Consider a query Q_1 expressed as an $\mathcal{EL}_{\mathcal{RV}}$-pattern defined by the following TBox $\mathcal{T}^{Q_1}$ over the signature Σ.*

$$\mathcal{T}^{Q_1} = \left\{ \begin{array}{lcl} Q_1 & \equiv & A_1 \sqcap \exists x.C_1 \\ C_1 & \equiv & A_2 \sqcap \exists y.Q_1 \end{array} \right\}$$

We explain now the difference between the set N_{V_R} of refreshing variables and the set N_{V_N} of non-refreshing variables. Given an $\mathcal{EL}_{\mathcal{RV}}$-TBox $\mathcal{T}$, a substitution σ maps a variable in N_{V_N} to a fixed value while the value assigned to a variable in N_{V_R} can be *refreshed* at each iteration over a cyclic definition. This semantics is captured through the notion of *unfolding* which turns (cyclic) $\mathcal{EL}_{\mathcal{RV}}$-definitions with refreshing variables to equivalent (infinite) $\mathcal{EL}_{\mathcal{RV}}$-definitions with non-refreshing variables. This is achieved by an *unfolding process* which replaces refreshing variables appearing in cyclic definitions of a given terminology by an infinite set of non-refreshing variables.

The notion of unfolding is formally defined below and then illustrated on an example.

Definition 4. *(Pattern Unfolding.)*
Let $\mathcal{T}$ be an $\mathcal{EL}_{\mathcal{RV}}$-TBox over an $\mathcal{EL}_{\mathcal{RV}}$-signature $\Sigma = (N_C, N_T)$, with $N_T = N_R \cup \mathcal{V}$ and $\mathcal{V} = N_{V_R} \cup N_{V_N}$. The unfolding of the Tbox $\mathcal{T}$ is a new Tbox, noted $u(\mathcal{T})$, over the $\mathcal{EL}_{\mathcal{RV}}$-signature $(N_C, N_R \cup N_{V_N})$ such that each $\mathcal{EL}_{\mathcal{RV}}$-pattern $P = A_0 \sqcap ... \sqcap A_n \sqcap \exists r_0.B_0 \sqcap ... \sqcap \exists r_m.B_m$ of $\mathcal{T}$ is mapped into an $\mathcal{EL}_{\mathcal{RV}}$-pattern $u(P)$ in $u(\mathcal{T})$. The unfolding u is defined as follows:

- $u(P) = u(A_0) \sqcap ... \sqcap u(A_n) \sqcap \exists u(r_0).u(B_0) \sqcap ... \sqcap \exists u(r_m).u(B_m)$.
- $u(t) = t, \forall t \in N_A \cup N_R \cup N_{V_N}$, *i.e., u is the identity function over atomic concept names, role names and non-refreshing variables.*
- *if $r_i \in N_{V_R}$ then each new call to $u(r_i)$ in the scope of $u(P)$ returns a new "fresh" variable from N_{V_N}. Note that, for $r_i = r_j$ in the description P, the calls to $u(r_i)$ and to $u(r_j)$ return the same fresh variable while recursive calls to $u(r_i)$ return different fresh variables.*

Hence, an unfolding of an $\mathcal{EL}_{\mathcal{RV}}$-pattern P enables to replace recursively each refreshing variable x by a new non-refreshing variable. Note that, in the case of refreshing variables that appear inside a cyclic definition of an $\mathcal{EL}_{\mathcal{RV}}$-pattern P, the unfolding of P leads to an infinite $\mathcal{EL}_{\mathcal{RV}}$-pattern $u(P)$ which uses an infinite set of variables.

We give below partial unfolding of the $\mathcal{EL}_{\mathcal{RV}}$-pattern Q_1 of Example 11.

$$u(Q_1) \equiv A_1 \sqcap \exists x_0.(A_2 \sqcap \exists y.(A_1 \sqcap \exists x_1.(A_2 \sqcap \exists x_2.(\ldots))))$$

Note that during the unfolding process, each iteration through the concept Q_1 generates a new variable x_i (a refreshing variable) while a unique variable y (non-refreshing variable) is used.

It is worth noting that an unfolding of an $\mathcal{EL}_{\mathcal{RV}}$-pattern $P \equiv A_0 \sqcap ... \sqcap A_n \sqcap \exists r_0.B_0 \sqcap ... \sqcap \exists r_m.B_m$ can be viewed as a $\langle N_{def} \cup \{q_f\}, N_R \cup N_A \cup N_{V_N} \rangle$-labeled tree $(\tau_P, \lambda, \delta)$ which is recursively defined as follows:

- $\lambda(\varepsilon) = P$
- $\forall i \in [0, n]$, we have: $i \in \tau_P$, $\delta(\epsilon, i) = A_i$ and $\lambda(i) = q_f$. The label q_f is a specific keyword used to label the leaves of the tree.
- $\forall i \in [n+1, n+m+1]$, we have: $i \in \tau_P$, $\delta(\varepsilon, i) = u(r_{i-n-1})$ and i is the root of the tree $\tau_{B_{i-n-1}}$

Example 12. *Figure 2.1 depicts the tree representation of the unfolded pattern Q_1 of Example 11. Each node is labeled by the corresponding concept in the TBox. Leaves are labeled by q_f and have an incoming edge labeled by an atomic concept name.*

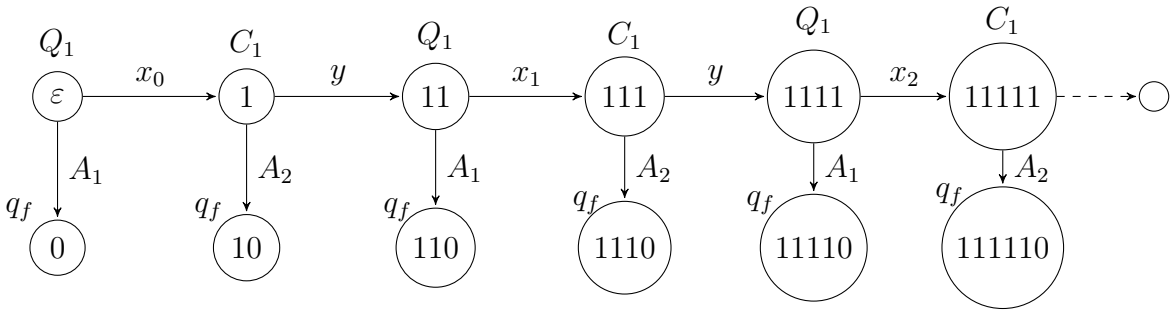


Figure 2.1: Infinite Tree of the Unfolded Pattern Q_1 of Example 11.

Instantiations of $\mathcal{EL}_{\mathcal{RV}}$-concept definitions (respectively, $\mathcal{EL}_{\mathcal{RV}}$-TBoxes) are given by variable *substitutions*. Given a TBox $\mathcal{T}$ with a signature $\Sigma = (N_C, N_T)$, where $N_T = N_R \cup \mathcal{V}$, a *substitution* σ is a mapping from $\mathcal{V}$ into the set of role names N_R. A substitution σ is extended to $\mathcal{EL}_{\mathcal{RV}}$-concepts in the obvious way, i.e.:

- $\sigma(T) = T$ if $T \in N_C \cup \{\top\} \cup N_R$;
- $\sigma(C \sqcap D) = \sigma(C) \sqcap \sigma(D)$ with C,D two $\mathcal{EL}_{\mathcal{RV}}$-concepts;
- $\sigma(\exists R.C) = \exists \sigma(R).\sigma(C)$.

Definition 5. *(Pattern Instances.)*
Let $\mathcal{T}$ be an $\mathcal{EL}_{\mathcal{RV}}$-TBox over an $\mathcal{EL}_{\mathcal{RV}}$-signature $\Sigma = (N_C, N_T)$, with $N_T = N_R \cup \mathcal{V}$ and $\mathcal{V} = N_{V_R} \cup N_{V_N}$ and let $P \equiv A_0 \sqcap ... \sqcap A_n \sqcap \exists r_0.B_0 \sqcap ... \sqcap \exists r_m.B_m$ be an $\mathcal{EL}_{\mathcal{RV}}$-pattern in $\mathcal{T}$. Let $\sigma : N_{V_N} \to N_R$ be a variable substitution. Then $\sigma(u(P))$ is an instance of P w.r.t. the variable substitution σ.

In the sequel, we abuse of notation and we write $\sigma(P)$ instead of $\sigma(u(P))$ for a pattern instance of P w.r.t. σ. Figure 2.2 shows an instance $\sigma(Q_1)$ of the pattern Q_1 of Example 11 given by a substitution σ that maps the variable y to $\sigma(y) = S$ and each variable x_i to $\sigma(x_i) = R_1$ if i is even and $\sigma(x_i) = R_2$ if i is odd.

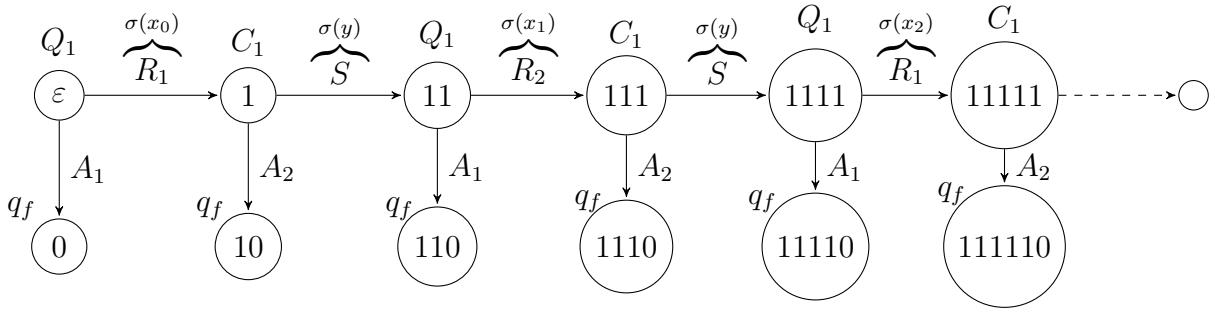


Figure 2.2: An Instantiation of the Pattern Q_1 of Example 11.

In addition, a substitution σ maps each $\mathcal{EL}_{\mathcal{RV}}$-TBox $\mathcal{T}$ into an $\mathcal{EL}$-TBox $\sigma(\mathcal{T})$ which is obtained by converting each $\mathcal{EL}_{\mathcal{RV}}$-concept definition $P \equiv C$ in $\mathcal{T}$ into an $\mathcal{EL}$-concept definition $\sigma(P) \equiv \sigma(C)$. In the example, we have $\sigma(\mathcal{T}^{Q_1}) = \{\sigma(Q_1)\}$.

Pattern instances can be split into two categories : regular and irregular instances. A regular instances will make substitute its variables using a regular choice. As a consequence, regular instances can be represented by a finite $\mathcal{T}$. For example, Figure 2.2 depicts a regular solution where the substitution of x alternates between R_1 and R_2. This alternation is regular and can be represented in a finite way. The associated finite TBox would be $\sigma(\mathcal{T}^{Q_1}) = \{\sigma(Q_1) \equiv A_1 \sqcap \exists R_1.\sigma(C_1); \sigma(C_1) \equiv A_2 \sqcap \exists S.\sigma(Q_1'); \sigma(Q_1') \equiv A_1 \sqcap \exists R_2.\sigma(C_1'); \sigma(C_1') \equiv A_2 \sqcap \exists S.\sigma(Q_1)\}$.

Irregular instances are by opposition instances that can not be represented in a finite way. To illustrate this notion, we will use the TBox $\{P \equiv \exists x.P'; P' \equiv \exists y.P\}$ where both x and y are refreshing. The idea is to construct a substitution such that the resulting automata is the language $R_1^n.R_2^n$ which is known to be irregular. Figure 2.3 depicts the corresponding tree.

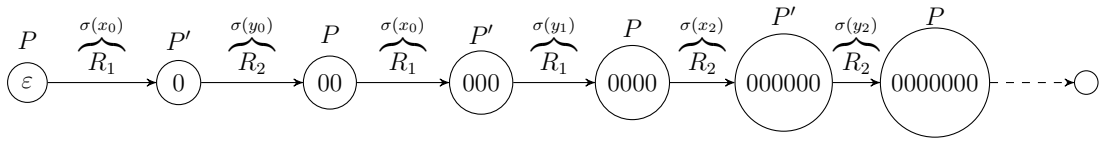


Figure 2.3: Irregular Instance of the Pattern P.

Lemma 1. *Let $\sigma(P)$ and $\theta(Q)$ be two instances of respectively two patterns P and Q. Then $\sigma(P)$ is subsumed by $\theta(Q)$ (i.e., $\sigma(P) \sqsubseteq \theta(Q)$) iff there exists a homomorphism from $\theta(Q)$ to $\sigma(P)$.*

Proof. It is a direct extension of the characterization of subsumption in $\mathcal{EL}$ using the so-called $\mathcal{EL}$-description trees [13]. In fact, unfolding transforms potential cyclic definition with TBox into non-cyclic infinite description without TBox. As a consequence, the different subsumption semantics are all equivalent and subsumption is equivalent to tree homomorphism. □

Subsumption between regular instances of two patterns is linked to subsumption w.r.t to the greatest fix-point semantics in $\mathcal{EL}$ as stated by the next lemma.

Lemma 2. *Let $\sigma(P)$ and $\theta(Q)$ be two regular instances of respectively two patterns P and Q. Let $\sigma_{reg}(P)$ and $\theta_{reg}(Q)$ the corresponding $\mathcal{EL}$ finite descriptions. If $\sigma(P) \sqsubseteq \theta(Q)$ then $\sigma_{reg}(P) \sqsubseteq_{gfp,\mathcal{T}} \theta_{reg}(Q)$ in $\mathcal{EL}$.*

Proof. By definition $\sigma(P) \sqsubseteq \theta(Q)$ implies that there exists a homomorphism from $\theta(Q)$ to $\sigma(P)$ in $\mathcal{EL}_{\mathcal{RV}}$ (Lemma 1). Since $\sigma(P)$ and $\theta(Q)$ being regular means that they can be represented with a finite $\mathcal{EL}$-TBox. These $TBox$ will corresponds to $\sigma_{reg}(P)$ and $\theta_{reg}(Q)$. Subsumption w.r.t to the greatest fix-point semantics in $\mathcal{EL}$ is characterized by simulation between the respective description graph [5]. Homomorphism between unfolded description graph preserves simulation. As a consequence we have $\sigma_{reg}(P) \sqsubseteq_{gfp,\mathcal{T}} \theta(Q)_{reg}(Q)$ in $\mathcal{EL}$. □

2.2.3 Reasoning with Refreshing Variables

In this thesis, patterns are viewed as queries defined in an $\mathcal{EL}_{\mathcal{RV}}$-TBox and evaluated over an $\mathcal{EL}$ knowledge base. We focus in the sequel on three reasoning mechanisms:

(i) matching, which is used as a mechanism to evaluate patterns over $\mathcal{EL}$ knowledge bases,

(ii) weak-subsumption that extends unification and

(iii) pattern containment, i.e., determining whether the result of a pattern is included in the result of another pattern whatever the considered knowledge base.

Definition 6 (Matching)**.** *Let $\mathcal{T}^P$ be an $\mathcal{EL}_{\mathcal{RV}}$ TBox, let $\mathcal{T}_g$ be an $\mathcal{EL}$ TBox and let $\mathcal{T} = \mathcal{T}^P \cup \mathcal{T}_g$. An $\mathcal{EL}_{\mathcal{RV}}$ matching problem is of the form $C \sqsubseteq^?_{\mathcal{T}} P$ where C is a concept in $\mathcal{T}_g$ and P is a pattern of $\mathcal{T}^P$. A solution (or matcher) of this problem is a substitution σ such that $C \sqsubseteq_{\sigma(\mathcal{T})} \sigma(P)$.*

Matching is used as a base mechanism to evaluate a pattern against concepts and individuals of a knowledge base. Let $\mathcal{T}^P$ be an $\mathcal{EL}_{\mathcal{RV}}$-TBox including a defined pattern P and let $KB = (\mathcal{T}, \mathcal{A})$ be an $\mathcal{EL}$ knowledge base. A defined concept C of $\mathcal{T}$ is an answer of P over KB iff the matching problem $C \sqsubseteq^?_{\mathcal{T}^P \cup \mathcal{T}} P$ has a solution.

To match a pattern P against an individual i of an ABox $\mathcal{A}$ of KB we make use of the notion of *most specific concept* (msc). The msc of an individual i, noted $msc(i)$, is a non-standard reasoning in description logics that enable to generalizes an individual to a concept by computing *the least concept description in the available description language that has this individual as an instance* [4]. Hence, it is natural to consider that an individual i of KB is in the answer of a pattern P over KB iff the matching problem

$msc(i) \sqsubseteq^?_{\mathcal{T}^P \cup \mathcal{T}} P$ is solvable. In [4] it is shown that for the description logic $\mathcal{EL}$, if one considers cyclic terminologies under greatest fix-point semantics, the msc always exists and can be computed in polynomial time.

Consequently, as explained above, evaluating a pattern P over a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ turns to matching P against concepts and individuals (more precisely, their msc) of $\mathcal{K}$. The computed matchers provides an explanation to why a concept C matches a pattern P.

Example 13. *For example, with the matcher σ of Example 1, the pattern instance $\sigma(Academic)$ provides an explanation to why we can say that the concept $Professor$ matches the pattern Academic w.r.t. the knowledge base of Table 1.*

As for non-refreshing semantics, this problem can be extended with variables on both sides. It leads to the following definition for weak-subsumption that extends unification problem.

Definition 7 (Weak-subsumption)**.** *A pattern P of a TBox $\mathcal{T}^P$ is weakly-subsumed by a pattern Q of a TBox $\mathcal{T}^Q$, noted $P \sqsubseteq_{\mathcal{T}} Q$, iff $\exists\sigma, \exists\theta$ s.t. $\sigma(P) \sqsubseteq_{\theta(\sigma(\mathcal{T}))} \theta(Q)$, with $\mathcal{T} = \mathcal{T}^P \cup \mathcal{T}^Q$.*
A weak-subsumption problem, noted $P \sqsubseteq^?_{\mathcal{T}} Q$ is the problem of testing whether $P \sqsubseteq_{\mathcal{T}} Q$.

Pattern containment is a new reasoning task that aims to compare two patterns with a different semantics than unification.

Definition 8 (Pattern containment)**.** *A pattern P of a TBox $\mathcal{T}^P$ is contained in a pattern Q of a TBox $\mathcal{T}^Q$, noted $P \sqsubseteq Q$, iff $\forall\sigma, \exists\theta$ s.t. $\sigma(P) \sqsubseteq_{\theta(\sigma(\mathcal{T}))} \theta(Q)$, with $\mathcal{T} = \mathcal{T}^P \cup \mathcal{T}^Q$.*
A pattern containment problem, noted $P \sqsubseteq^? Q$ is the problem of testing whether $P \sqsubseteq Q$.

Pattern containment enables to compare patterns w.r.t. their respective answers as stated by the following lemma which is a direct consequence of Definition 8.

Lemma 3. *Let P and Q be two patterns respectively defined in the $\mathcal{EL}_{\mathcal{RV}}$ TBoxes $\mathcal{T}^P$ and $\mathcal{T}^Q$. Let $\mathcal{T}$ be an arbitrary $\mathcal{EL}$ TBox and let C be a concept of $\mathcal{T}$. If $P \sqsubseteq Q$ then we have:*
$C \sqsubseteq^?_{\mathcal{T}^P \cup \mathcal{T}} P$ is solvable $\Rightarrow$ $C \sqsubseteq^?_{\mathcal{T}^Q \cup \mathcal{T}} Q$ is solvable

Note that matching is also a particular case of weak-subsumption where the pattern on the left is ground. The next section gives an insight in the shape of matchers considered later on.

2.2.4 Regular Matchers

This section discusses two main issues regarding the matching problem in the context of the logic $\mathcal{EL}_{\mathcal{RV}}$:

(i) a matching problem may have infinitely many matchers that lead to instances of P that are incomparable w.r.t. subsumption and,

(ii) some matchers can be represented by a finite $\mathcal{EL}_{\mathcal{RV}}$-terminology (hereafter called regular matchers), there exists matchers that are not regular (i.e., a matcher σ such that the derived instance $\sigma(\mathcal{T}^{Q_2})$ cannot be represented by a finite $\mathcal{EL}_{\mathcal{RV}}$-terminology).

$$\mathcal{T}_2 = \left\{ \begin{array}{lcl} C & \equiv & \exists R.A_1 \sqcap \exists S.C \sqcap \exists R.D, \\ D & \equiv & \exists S.A_1 \sqcap \exists R.D \sqcap \exists S.C, \end{array} \right\}$$
$$\mathcal{T}^{Q_2} = \{Q_2 \equiv \exists x.A_1 \sqcap \exists y.Q_2\} \quad \text{with } x, y \in N_{V_R}$$

Table 2.4: The terminologies $\mathcal{T}^{Q_2}$ and $\mathcal{T}_2$.

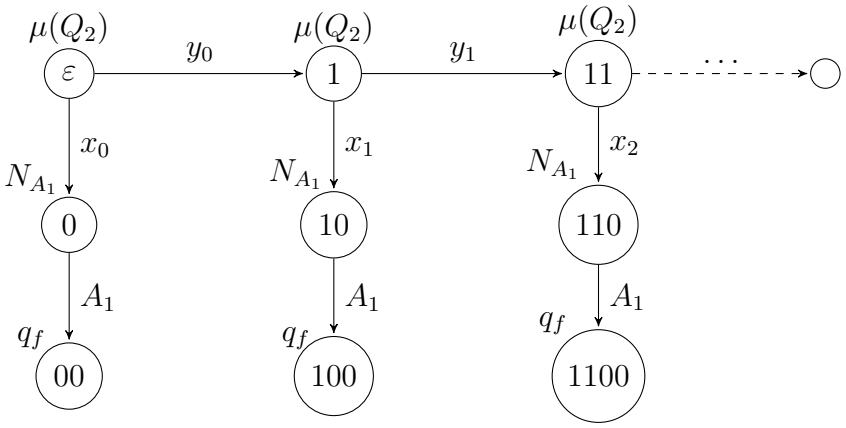


Figure 2.4: Unfolding of Q_2

Consider the matching problem $C \sqsubseteq^?_{\mathcal{T}^{Q_2} \cup \mathcal{T}_2} Q_2$, where the terminologies $\mathcal{T}^{Q_2}$ and $\mathcal{T}_2$ are shown at Table 2.4. The variables x and y are refreshing variables of Q_2. Unfolding of Q_2 is depicted in Figure 2.4. Table 2.5 exhibits several possible matchers that solve this problem.

$$\left\{ \sigma_1(x_i) = R \text{ and } \sigma_1(y_i) = S, \forall i \in \mathbb{N} \right. \tag{2.1}$$

$$\begin{cases} \sigma_2(x_i) = R \text{ and } \sigma_2(y_i) = R & \text{if } i \text{ is even} \\ \sigma_2(x_i) = S \text{ and } \sigma_2(y_i) = S & \text{if } i \text{ is odd} \end{cases} \tag{2.2}$$

$$\begin{cases} \phi_k(x_i) = R \text{ and } \phi_k(y_i) = S & \forall i \in [1, k[\\ \phi_k(x_k) = R \text{ and } \phi_k(y_k) = R & \\ \phi_k(x_i) = S \text{ and } \phi_k(y_i) = R & \forall i > k \end{cases} \tag{2.3}$$

Table 2.5: Regular matchers.

We make the following observations.

- σ_1 and σ_2 are regular matchers in the sense that $\sigma_1(\mathcal{T}^{Q_2})$ (respectively, $\sigma_1(\mathcal{T}^{Q_2})$) can be described by a finite $\mathcal{EL}_{\mathcal{RV}}$-Tbox. Table 2.6 shows the corresponding TBoxes.
- ϕ_k, for $k \in \mathbb{N}$ defines an (infinite) family of matchers that solve our matching problem. Note that, for a fixed integer k, ϕ_k is a regular matcher (Table 2.6 shows a corresponding TBox $\phi_k(\mathcal{T}^{Q_2})$).
- The matchers σ_1, σ_2 and ϕ_k, for $k \in \mathbb{N}$, are pairwise incomparable w.r.t. subsumption in the sense that for any $i, j \in \mathbb{N}$, with $i \neq j$, the concepts

$$\sigma_1(\mathcal{T}^{Q_2}) = \quad \{\sigma_1(Q_2) \equiv \exists R.A_1 \sqcap \exists S.\sigma_1(Q_2)\}$$

$$\sigma_2(\mathcal{T}^{Q_2}) = \left\{ \begin{array}{ll} \sigma_2(Q_2) & \equiv \exists R.A_1 \sqcap \exists R.\sigma_2(Q'_2), \\ \sigma_2(Q'_2) & \equiv \exists S.A_1 \sqcap \exists S.\sigma_2(Q_2) \end{array} \right\}$$

$$\phi_k(\mathcal{T}^{Q_2}) = \left\{ \begin{array}{ll} \phi_k(Q_2) & \equiv \exists R.A_1 \sqcap \exists S.\phi_k(Q'_2), \\ \phi_k(Q'_2) & \equiv \exists R.A_1 \sqcap \exists S.\phi_k(Q'_3), \\ \dots & \\ \phi_k(Q'_k) & \equiv \exists R.A_1 \sqcap \exists R.\phi_k(Q'_{k+1}), \\ \phi_k(Q'_{k+1}) & \equiv \exists S.A_1 \sqcap \exists R.\phi_k(Q'_{k+1}) \end{array} \right\}$$

Table 2.6: Representation of regular matchers as finite TBoxes.

$\phi_i(Q_2), \phi_j(Q_2), \sigma_1(Q_2), \sigma_2(Q_2)$ are pairwise incomparable w.r.t. the subsumption relation.

We exhibit now a non regular matchers for our matching problem. Let u be an infinite geometric sequence $u_0, u_1, \dots$ with $u_0 > 1$ and $\forall i \in \mathbb{N}$, we have $u_{i+1} - u_i > 3$. Let v and w be respectively two infinite sequences $v_0, v_1, \dots$ and $w_0, w_1, \dots$ such that $v_i = u_i + 1$ and $w_i = u_i + 2, \forall i \in \mathbb{N}$. We define the substitution ρ^u as follows:

- $\rho^u(x_l) = R$ and $\rho^u(y_l) = S$ if l is not a member of the sequences v or w
- $\rho^u(x_{v_i}) = R$ and $\rho^u(y_{v_i}) = R$ for each member v_i of the sequence v
- $\rho^u(x_{w_i}) = S$ and $\rho^u(y_{w_i}) = S$ for each member w_i of the sequence w

The substitution ρ^u is a matcher of the considered matching problem. However, ρ^u is not regular (i.e., it cannot be described by a finite TBox) as stated by the following lemma.

Lemma 4. *Consider the matching problem $C \sqsubseteq^?_{\mathcal{T}^{Q_2} \cup \mathcal{T}_2} Q_2$, where the terminologies $\mathcal{T}^{Q_2}$ and $\mathcal{T}_2$ are given at Table 2.4. Let $\rho^u(Q_2)$ be the matcher defined as previously. For any substitution σ such that $\sigma(\mathcal{T}^{Q_2})$ is a finite TBox, we have $\sigma(Q_2) \not\equiv \rho^u(Q_2)$.*

Proof. W.l.o.g. assume $u_0 > 1$. The sequence $y_0, y_1, \dots$ forms an infinite word $\tau = S^{u_0} R S^{u_1 - u_0} R \dots R S^{u_n - u_{n-1}} \dots$
Note that the sequence $u_1 - u_0, \dots u_n - u_{n-1}$ forms a geometric progression. Consequently, the occurrences of the sequences of S follows a geometric progression pattern and hence cannot be represented by a finite state automaton. □

Fortunately, it is sufficient to look for regular matchers because if a matching problem is solvable then it necessarily has a regular matcher (Section 4.3 Corollary 2). The proof derives from the completeness of our matching algorithm.

2.3 Conclusion of Chapter 2

Even though being recent add-ons to description logics, variables have been a source of interesting research. At a terminological level, it mainly focuses over two non-standard reasoning tasks known as matching and unification.

These reasoning tasks have been investigated for two families of logics based on $\mathcal{EL}$ and $\mathcal{FL}_0$. It has been demonstrated that complexity in $\mathcal{EL}$ is NP-Complete for both cases except for unification w.r.t general TBox which remains an open problem. It is an advantage comparatively to $\mathcal{FL}_0$ which suffers a blow-up of complexity while considering the different kind of TBoxes.

The main contribution of this chapter is the definition of $\mathcal{EL}_{\mathcal{RV}}$. $\mathcal{EL}_{\mathcal{RV}}$ extends $\mathcal{EL}$ by allowing refreshing semantics for role variables as well as considering cyclic pattern which can be seen as pattern queries. Pattern instance, which represents a pattern whose variables have been substitutes can be either regular or irregular. Regular solutions can be express with finite $\mathcal{EL}$-TBox and preserve the subsumption relationship with regard to the greatest fix-point semantics.

Moreover, this new semantics for variables questioned known reasoning task such as matching and unification. We showed that unsolvable matching problems ($Doctor \sqsubseteq^?_{\mathcal{T}} Academic$) in non-refreshing semantics would be solvable in refreshing semantics. Three reasoning tasks have then be introduced. Matching and unification have respectively been extended to matching and weak-subsumption to support refreshing variables. On the other hand, we define a brand new reasoning task : pattern containment. It allows to compare pattern queries results whatever the considered knwoledge base.

The next chapter introduces a framework based on the notion of description automata which will be of later used to solve reasoning mechanisms in $\mathcal{EL}_{\mathcal{RV}}$.

Chapter 3

From $\mathcal{EL}_{\mathcal{RV}}$ Description Logic to Automata

Adding refreshing semantics to description logic unlocked new ways to reason through $\mathcal{EL}_{\mathcal{RV}}$-TBox and pattern queries. Due to the differences stated in the previous chapter, known algorithms can not handle refreshing semantics. This chapter will focus on introducing description automata. By extending, fresh variable automata, this class of automata will handle refreshing variables. Moreover, reasoning in $\mathcal{EL}_{\mathcal{RV}}$ will be reduced to simulation problems in the scope of description automata. Table 3.1 summarizes notations introduced in this chapter.

Symbol	**Description**
A_P	Description automata corresponding to the pattern P
$\mathcal{Q}$	Set of states of A_P
$\mathcal{L}$	Set of edges labels of A_P
$\mathcal{V}ar$	Set of variables of A_P
q_0	Initial state of A_P
q_f	Final state of A_P
δ	Transition function of A_P
κ	Refreshing function of A_P
$(q_p, \vec{I})$	Configuration of a description automata
$T(A_P, \sigma)$	Configuration tree of A_P w.r.t σ
$\ll_\forall$	Universal simulation
$\ll_\exists$	Existential simulation

Table 3.1: Notations introduced in Chapter 3

3.1 $\mathcal{EL}_{\mathcal{RV}}$-Description Automata

Our reasoning procedures over $\mathcal{EL}_{\mathcal{RV}}$-terminologies are built on the notions of $\mathcal{EL}_{\mathcal{RV}}$-description automata. Such automata recognize *configuration trees* which are nothing

other than a syntactic variant of pattern instances. We associate with each $\mathcal{EL}_{\mathcal{RV}}$-pattern P an $\mathcal{EL}_{\mathcal{RV}}$-description automata A_P such that there is a one-to-one correspondence between the configuration trees recognized by A_P and the instances of P. Consequently, an $\mathcal{EL}_{\mathcal{RV}}$-description automaton A_P characterizes all the possible instances of its associated $\mathcal{EL}_{\mathcal{RV}}$-pattern P.

Definition 9. *($\mathcal{EL}_{\mathcal{RV}}$-description automaton.)*
Let $P \equiv A_0 \sqcap ... \sqcap A_n \sqcap \exists r_0.B_0 \sqcap ... \sqcap \exists r_m.B_m$ be an $\mathcal{EL}_{\mathcal{RV}}$-pattern defined in a $\mathcal{T}^P$ over the signature $\Sigma = (N_C, N_T)$, with $N_C = N_{def} \cup N_A$, $N_T = N_R \cup \mathcal{V}$ and $\mathcal{V} = N_{V_R} \cup N_{V_N}$. The $\mathcal{EL}_{\mathcal{RV}}$-description automaton associated with P, denoted A_P, is a tuple $A_P = (\mathcal{Q}, \mathcal{L}, \mathcal{V}ar, p_0, q_f, \delta, \kappa)$ built as follows:

(i) $\mathcal{L} \subseteq N_A \cup N_R$ *is a finite alphabet,*

(ii) $\mathcal{V}ar \subseteq \mathcal{V}$ *is a finite set of variables,*

(iii) $\mathcal{Q} = N_{def} \cup \{q_f\}$ *is a finite set of states,*

(iv) $q_0 = P$ *is the initial state and* q_f *is the final state,*

(v) $\delta \subseteq \mathcal{Q} \times (\mathcal{L} \cup \mathcal{V}ar) \times \mathcal{Q}$ *is a transition relation defined as follows: For each* $Q \equiv A_0 \sqcap ... \sqcap A_n \sqcap \exists r_0.B_0 \sqcap ... \sqcap \exists r_m.B_m \in \mathcal{T}$ $\{(Q, A_i, q_f) : for\ i \in [0, n]\} \cup \{(Q, r_j, B_j) : for\ j \in [0, m]\} \in \delta$

(vi) $\kappa : \mathcal{V}ar \rightarrow 2^{\mathcal{Q}}$ *is the refreshing function defined as follows: For each* $Q \equiv A_0 \sqcap ... \sqcap A_n \sqcap \exists r_0.B_0 \sqcap ... \sqcap \exists r_m.B_m \in \mathcal{T}$, $\forall r_j \in \mathcal{V}ar$ *we have:* $\kappa(r_j) = \{Q\}$ *if* $r_j \in N_{V_R}$, *or* $\kappa(r_j) = \emptyset$ *if* $r_j \in N_{V_N}$.

Definition 9 associates with each defined concept $P \equiv A_0 \sqcap ... \sqcap A_n \sqcap \exists r_0.B_0 \sqcap ... \sqcap \exists r_m.B_m$ in a TBox $\mathcal{T}^P$ an $\mathcal{EL}_{\mathcal{RV}}$-description automaton A_P whose states are made of the set of defined concept names of $\mathcal{T}$ in addition to a special final state q_f. Transitions of A_P are labelled either with letters, taken from an alphabet made of the atomic concept names and role names, or variables taken from the set of role variables. More precisely, each atomic concept name A_i that appears in the definition of P leads to a transition from the node P to q_f labeled with the letter A_i. Each description $\exists r_i.B_i$ that appears in the definition of P leads to a transition from the node P to the node B_i (the initial state of the automaton A_{B_i}) labeled with the term r_i. When the term r_i is a refreshing variable, its refreshing state is given by the function κ (i.e., in this case $\kappa(r_i) = \{P\}$). Note that Definition 9 extends naturally to ground concepts (i.e., when P do not contain variables). The obtained automaton A_P is then a ground automaton without variables.

Figure 3.1 depicts the description automaton of the TBoxes $\mathcal{T}^{Q_2}$ and $\mathcal{T}_2$ given at Example 2.6. The variables x and y are refreshed at state Q_2, hence we have $\kappa(x) = \kappa(y) = \{Q_2\}$.

$\mathcal{EL}_{\mathcal{RV}}$-description automata can be viewed as a variant of variable automata with refreshing variables introduced in [19, 20], where the notion of a *run* is however substantially modified in order to capture instantiation trees of $\mathcal{EL}_{\mathcal{RV}}$ patterns. Indeed, a description automaton of a given pattern P is in fact a compacted representation of all the possible instantiations of P. To make this statement more precise, we introduce the notions of

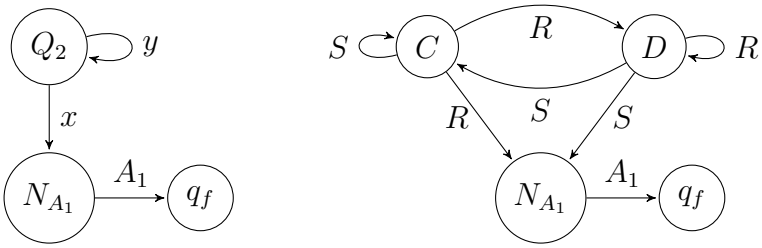


Figure 3.1: Description Automata of $\mathcal{T}^{Q_2}$ and $\mathcal{T}_2$.

configurations and configuration trees and we show that these latter ones are *equivalent* to the instances of the pattern P.

A run of a description automaton A_P is defined over configurations. Informally, a configuration gives the values assigned to variables at a given state of the *execution* of the automaton A_P. Since, on one side a given state may be visited (infinitely) many times and on another side refreshing variables may see their assigned value changing at their refreshing states, a configuration includes a vector of integer used to distinguish between multiple value assignations to a given refreshing variable. More precisely, we define a configuration as a pair $(q, \vec{I})$ where q is a state of A_P and $\vec{I}$ is a vector of integers, where the i^{th} component of $\vec{I}$ records the current index of the i^{th} variable, assuming that the variables are sorted according to their lexicographic order. By this way, we are able to generate several copies of a refreshing variable by incrementing its corresponding component in the vector $\vec{I}$. In the sequel, we use the following notations. Let $\vec{G} \in \mathbb{N}^{|\mathcal{V}ar|}$ be a tuple of integer. $\vec{G}$ is called a counter and is used to associate an integer value with each variable in $\mathcal{V}ar$. More precisely, we assume that each variable $x \in \mathcal{V}ar$ is associated with a fixed position in $\vec{G}$, noted G_x, which gives the value of the counter of x in $\vec{G}$.

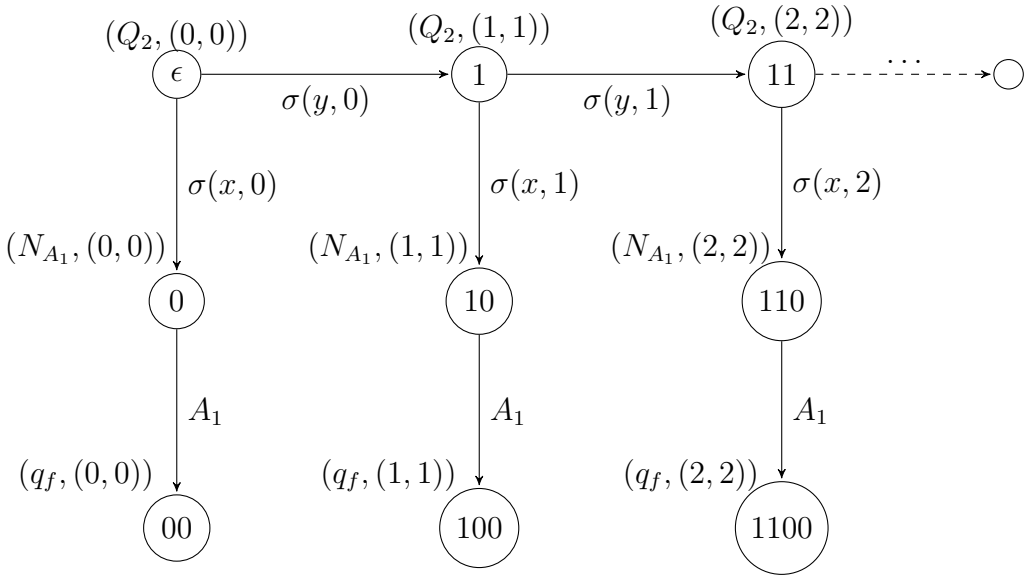


Figure 3.2: A Configuration Tree of the Automaton A_{Q_2}.

Example 14. *Figure 3.2 shows a configuration tree of A_{Q_2}. Since the automaton A_{Q_2} contains two variables, x and y, the counter associated to its configurations is 2 dimensional. As an example, $(Q_2, (0,0))$ is a configuration in this tree which indicates that the current state of the execution of A_{Q_2} is Q_2 and the current copies of the variables are $(x,0)$ and $(y,0)$. This is because this configuration uses a counter $\vec{I} = (0,0)$ with leads to $I_x = 0$ and $I_y = 0$ (the copies of the variables x and y). The configuration $(q_f, (2,2))$ records a run which is at state q_f while using the copies $(x,2)$ and $(y,2)$ of the variables x and y.*

The formal definition of a run is given below.

Definition 10. *(Run of an $\mathcal{EL}_{\mathcal{RV}}$-description automaton.)*
Let $A_P = (\mathcal{Q}, \mathcal{L}, \mathcal{V}ar, q_0, \delta, q_f, \kappa)$ be a description automaton and let σ: $\mathcal{V}ar \times \mathbb{N} \mapsto N_R$ be a variable substitution. Let $\vec{G} \in \mathbb{N}^{|\mathcal{V}ar|}$ be a counter and let $S \subseteq \mathcal{Q} \times \mathbb{N}^{|\mathcal{V}ar|}$. A run of A_P using a substitution σ, denoted $T(A_P, \delta)$ and called a configuration tree, is a $\langle S, \mathcal{L}\rangle$-labeled tree $(\tau_P, \lambda_P, \delta_P)$ constructed as follows:

- *$\lambda(\epsilon) = (q_0, \vec{0})$, a root of the tree, and let $\vec{G} = \vec{0}$, a global counter initialized to a vector of zero.*
- *Let $n \in \tau_P$ be a node such that $\lambda_P(n) = (q, \vec{I})$. For every transition $(q, t, q') \in \delta$ a new child ni of n is generated according to the following sequence:*

  (1) $\delta_P(n, ni) = \begin{cases} \sigma(t, I_t) & \text{if } t \in \mathcal{V}ar \\ t & \text{if } t \in \mathcal{L} \end{cases}$

  (2) $\forall y \in \mathcal{V}ar$, *if* $q' \in \kappa(y)$ *then* $G_y := G_y + 1$, *and*

  (3) Let $\vec{I'} = \begin{cases} I'_y = G_y, & \forall y \in \mathcal{V}ar, \text{ s.t. } q' \in \kappa(y) \\ I'_y = I_y & \text{otherwise} \end{cases}$

  (4) $\lambda_P(ni) = (q', \vec{I'})$

Intuitively, when a run of a description automaton A_P is at configuration $(q, \vec{I})$ and there is a transition (q, t, q') in A_P then the automaton A_P moves to a configuration $(q', \vec{I'})$ upon the input t, if t is a letter in the alphabet $\mathcal{L}$, or upon $\sigma(t, I_t)$, if t is a variable. Note that, in this case $\sigma(t, I_t)$ corresponds to the value assigned to the copy t_{I_t} of the variable t by the substitution σ. For each variable y that is refreshed at the state q' a new copy of y is created at the configuration $(q', \vec{I'})$ which is materialized by a new value I'_y in the vector $\vec{I'}$. To ensure that I'_y is a new value (not already used before) a global counter $\vec{G}$ is associated with each run of the automaton A_P. For each variable x, G_x stores the index of the newest copy of the variable x considered.

Definition 10 defines a run of a description automaton A_P as a tree rooted at the configuration $(q_0, \vec{0})$. Note that, a ground automaton A_C (i.e., an automaton without variable) has a unique configuration tree, which is noted hereafter $T(A_C, id)$. All the configurations of $T(A_C, id)$ have a zero dimension vector as a counter and hence the counter is omitted from the notation (i.e., we write (q) for such configurations).

Note that a *partial configuration tree* $PT(A_P, \sigma)$ is a partial tree of the configuration tree $T(A_P, \sigma)$;

Now that the definitions of the automata and their configuration tree has been given, the remaining will be devoted to unravel their link to pattern instances in order to reduce reasoning in $\mathcal{EL}_{\mathcal{RV}}$ into simulation with $\mathcal{EL}_{\mathcal{RV}}$-description automata.

3.2 Reducing Reasoning in $\mathcal{EL}_{\mathcal{RV}}$ to Simulation

This section emphasizes on the link between substitutions and configuration trees. More particularly, we introduce a notion of equivalence between substitutions which is used in the subsequent lemma (Lemma 5) to establish a tight relationship between valuation in description logics (i.e., pattern instances) and configuration trees.

Definition 11 (Equivalence between substitutions)**.** *Let $\mathcal{V}, \mathcal{V}'$ be two sets of variables and let Val be a set of constants. Let $\sigma : \mathcal{V} \cup Val \to Val$ and $\phi : \mathcal{V}' \cup Val \to Val$ be two substitutions. Then we say that σ is equivalent to ϕ, and we write $\sigma \equiv \phi$, iff there exists an isomorphism f between σ and ϕ which is the identity for element of Val.*

Lemma 5 given below establishes a strong connection between patterns instances and configuration trees.

Lemma 5. *Let P be an $\mathcal{EL}_{\mathcal{RV}}$-pattern of a terminology $\mathcal{T}^P$ and let A_P be its corresponding description automaton. Then, we have:*

(i) *$\forall \phi$: $\phi(P)$ is an instance of P, then there exists σ such that $\sigma \equiv \phi$ and $T(A_P, \sigma)$ is a configuration tree of A_P*

(ii) *$\forall \sigma$: $T(A_P, \sigma)$ is a configuration tree of A_P, then there exists ϕ such that $\phi \equiv \sigma$ and $\phi(P)$ is instance of P*

Proof. (i): we show that, given an instance $\phi(P) = (\tau, \lambda, \delta)$, we construct a configuration tree $T(A_P, \sigma) = (\tau', \lambda', \delta')$ and an isomorphism f between ϕ and σ. Let $\vec{G}$ be a global counter associated with the execution of A_P. $T(A_P, \sigma)$ and f are inductively defined as follows:

- Initialization: $\varepsilon \in \tau'$ and $\lambda'(\varepsilon) = (\lambda(\varepsilon), \vec{0})$.
- Let $n \in \tau'$ with $\lambda'(n) = (\lambda(n), \vec{I})$. For every $ni \in \tau$:
  - $ni \in \tau'$.
  - For every $\delta(n, ni) = \phi(u(t))$:
    - Let $t_x = (t, I_t)$ if t is a variable and $t_x = t$ if t is a constant.
    - $\delta'(n, ni) = \sigma(t_x)$ with $\sigma(t_x) = \phi(u(t))$. This is because $\delta(n, ni) = \phi(u(t))$ implies that the description $\lambda(n)$ includes an atom $\exists x.\lambda(ni)$. Hence, by construction (Definition 9,(v), the automaton A_P includes a transition $(\lambda(n), t, \lambda(ni))$.
    - $f(u(t)) = t_x$.

$*$ $\lambda'(ni) = (\lambda(ni), \vec{I'})$ with $I'_x = G_x$ if $\lambda(ni) \in \kappa(x)$ and $I'_x = I_x$ otherwise.

By construction, we have $T(A_P, \sigma)$ is a configuration tree of A_P and f is the identity on constants and forms a one-to-one mapping between each variable $u(x)$ of $u(P)$ and a pair (x, i), with $i \in \mathbb{N}$, of $T(A_P, \sigma)$ such that $\sigma(t) = v$ iff $\delta(f(t)) = f(v)$ (i.e., f is an isomorphism between σ and ϕ).

(ii): : Similar reasoning can be applied by exchanging $T(A_P, \sigma)$ and $\phi(P)$ roles.

□

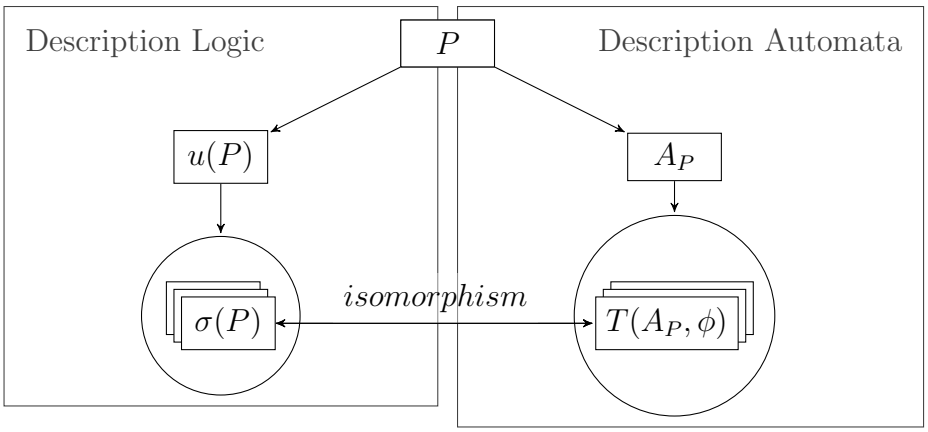


Figure 3.3: Schematical relationships between representations

Figure 3.3 represents schematically the relationships between description logics and description automata. Unfolding leads to instance trees while automata recognizes configuration tree. Automata allows to represent with a finite machine the potential infinite trees due to unfolding which will successively produce new instances for variables.

Our translation of $\mathcal{EL}_{\mathcal{RV}}$ patterns to description automata is used to design decision procedures for the reasoning problems of our interest (i.e., matching, weak-subsumption and pattern containment). We introduce below the notion of universal simulation which is used to compare description automata w.r.t. to their configuration trees.

Definition 12 (Universal simulation)**.** *Let A_P and A_Q two description automata. A_Q is universally simulated by A_P, noted $A_Q \ll_\forall A_P$, if $\forall\sigma, \exists\theta$ such that there exists a homomorphism Z from $T(A_Q, \theta)$ to $T(A_P, \sigma)$ with $Z(\varepsilon) = \varepsilon$. The universal simulation problem is the problem of testing whether $A_Q \ll_\forall A_P$.*

Let Z be an homomorphism from $T(A_Q, \theta) = (\tau, \lambda, \delta)$ to $T(A_P, \sigma) = (\tau, \lambda', \delta)$. We extend universal simulation to configurations as follows if $Z(i) = j$ then we write $\lambda(i) \ll_\forall \lambda'(j)$ to denote the homomorphism from $Z(i)$ to j

Note that $A_Q \ll_\forall A_P$ is equivalent to $(q_0, \vec{0}) \ll_\forall (p_0, \vec{0})$. We give now our main technical result consisting in the characterization of pattern containment between $\mathcal{EL}_{\mathcal{RV}}$-patterns in terms of universal simulation between $\mathcal{EL}_{\mathcal{RV}}$-description automata.

Theorem 1. *Let P and Q be two $\mathcal{EL}_{\mathcal{RV}}$-patterns and let A_P and A_Q be respectively the description automata of P and Q. A pattern containment problem $P \sqsubseteq^? Q$ has a solution if and only if $A_Q \ll_\forall A_P$.*

This theorem is a direct consequence of Definition 12, Lemma 1 and Lemma 5. In the case P is a ground $\mathcal{EL}$-concept, Theorem 1 also provides a characterization of matching using description automata as stated below.

Corollary 1. *Let Q be an $\mathcal{EL}_{\mathcal{RV}}$-pattern and C a ground $\mathcal{EL}_{\mathcal{RV}}$-description. Let A_Q and A_C be respectively the description automata of Q and C. A matching problem $C \sqsubseteq^? Q$ has a solution if and only if $A_Q \ll_\forall A_C$.*

In order to characterize weak-subsumption with simulation, we will use a slight variation of simulation called existential simulation noted $\ll_\exists$.

Definition 13 (Existential Simulation)**.** *Let A_P and A_Q two description automata. A_Q is existentially simulated by A_P, noted $A_Q \ll_\exists A_P$, if $\exists\sigma, \exists\theta$ such that there exists a homomorphism Z from $T(A_Q, \theta)$ to $T(A_P, \sigma)$ with $Z(\varepsilon) = \varepsilon$. The existential simulation problem is the problem of testing whether $A_Q \ll_\exists A_P$.*

Let Z be an homomorphism from $T(A_Q, \theta) = (\tau, \lambda, \delta)$ to $T(A_P, \sigma) = (\tau, \lambda', \delta)$. We extend existential simulation to configurations as follows if $Z(i) = j$ then we write $\lambda(i) \ll_\exists \lambda(j)$ to denote the homomorphism from $Z(i)$ to j

Theorem 2. *Let P and Q be two $\mathcal{EL}_{\mathcal{RV}}$-patterns and $\mathcal{T}^Q$ and let A_P and A_Q be respectively the description automata of P and Q. A weak-subsumption problem $P \sqsubseteq^?_{\mathcal{T}} Q$ has a solution if and only if $A_Q \ll_\exists A_P$.*

As Theorem 1 is a direct consequence of Definition 12, Lemma 1 and Lemma 5. It also provides a characterization for matching since matching is a special case of weak-subsumption.

Theorem 1 and Theorem 2 reduce matching, weak-subsumption and containment in the context of the logic $\mathcal{EL}_{\mathcal{RV}}$ into a simulation tests between description automata. Next chapters exploit this link to design algorithms that solve those reasoning tasks.

3.3 Comparison with Variables Automata

Description automata are inspired of fresh variable automata [19, 20]. This class of automata introduced by Belkhir et al. extends variable automata to handle refreshing variables with infinite valuation domain. Research around this class of automata mainly deal with closure properties (i.e. closed under union, concatenation, Kleene operator and intersection). Simulation has been studied as a decision procedure and proved to be EXPTIME-COMPLETE. It has notably been applied to service composition which can be reduced to simulation.

Compared to description automata, the formal definition is similar to fresh variable automata. However, our definition of runs differs. Thus preventing from directly using those results. The difference lies in the semantics of variables themselves. Indeed, there

are two states for a variable in fresh variable automata, it is either free or bound. If it is not bound then any bound can be associated and this bound will be kept for the remaining until refreshment. Consequently, a variable refreshed by the same state may be bound to different value while running different branches. Description automata can not be so lenient regarding variables bound. Indeed, a strong property ensures that an instance may be synchronized within run.

Figure 3.4 presents an automaton that will be used to illustrate the main difference. This automaton can be either seen as a fresh variable automaton or a description automaton.

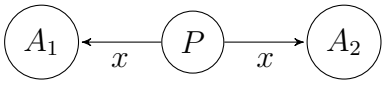
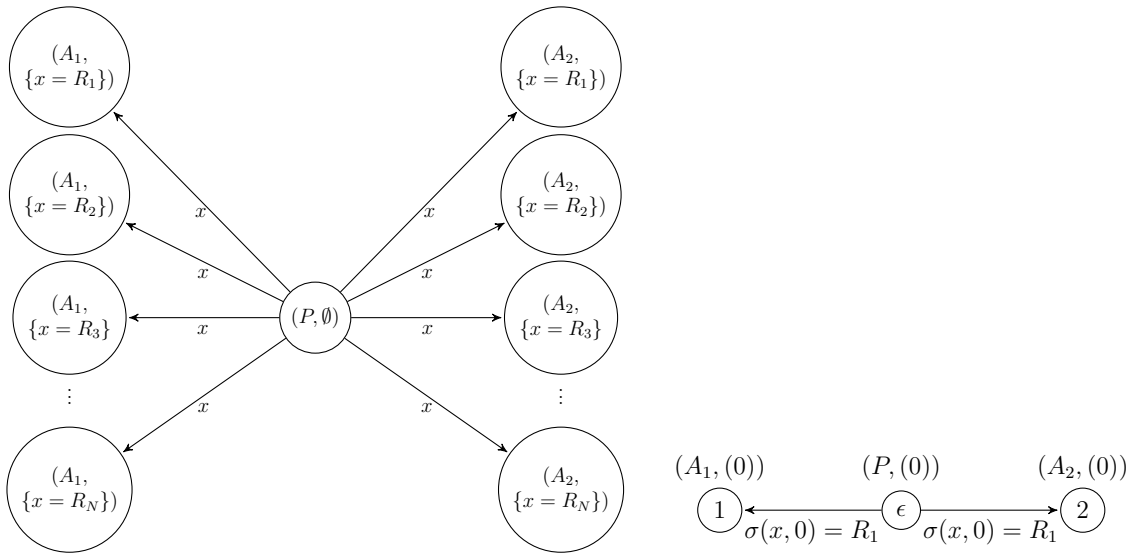


Figure 3.4: Variable Automata A_P

A language of a fresh variable automaton is given by its configuration automaton. Configurations of a fresh variable are pairs of the form $(q, \mathcal{S})$ where q is a state and $\mathcal{S}$ the current value assignment. For example, the configuration $(A_1, \{x = R_1\})$ means that the current state is A_1 and x is assigned to R_1. When a variable is bound, its assignment is added. When a variable is refreshed, its assignment is removed. As expected, the initial configuration is made of the initial state with an empty variable assignment. Figure 3.5(a) corresponds to the complete configuration automaton of the running example. We assume here that the alphabet is made of the (infinite) set $\{R_1, R_2, ..., R_N\}$. In comparison, Figure 3.5(b) depicts the run associated to $\sigma(x, 0) = R_1$.



(a) Configuration Automata of A_P (b) Configuration Tree $T(A_P, \sigma)$

Figure 3.5: Runs of Fresh Variable Automata

As illustrated in the figures, the main difference it that a free variable transition of a fresh variable automata represents many transitions at the same time. On the other hand, in our definition of configuration tree a variable transition represent one transition among the many possible. The difference is not limited to choosing one possibility. Indeed, we

ensure that any outgoing transition labeled by a same instance of a variable is assigned to the same value until refreshment. Thus making the definitions of runs intrinsically different. As demonstrated previously, this properties are important to capture the semantics of description logics.

3.4 Conclusion of Chapter 3

Reasoning in Description Logics with refreshing variables bring up new challenges to handle for matching, weak-subsumption and pattern containment.

This chapter entails the definition of $\mathcal{EL}_{\mathcal{RV}}$-description automata which allow to reduce reasoning tasks to simulation tests. $\mathcal{EL}_{\mathcal{RV}}$-description automata are a class of automata inspired from fresh variable automata with however different notions of runs. This class naturally handles the refreshing variables while allowing to distinguish variable instances. This feature imply a one-to-one comparison between instances of a pattern and runs of its corresponding description automata.

We consider two kinds of simulation, universal and existential simulations. The main difference lies in the quantifier associated. Pattern-containment is then reduced to universal simulation and weak-subsumption to existential simulation. Matching is a special case of both pattern containment and weak-subsumption. As a consequence it can be reduced to the two simulations.

In the remaining, we exploit this reduction to design algorithms in order to solve the different reasoning tasks.

Chapter 4

Solving Matching in $\mathcal{EL}_{\mathcal{RV}}$

Even though matching is a sub-problem of pattern containment and weak-subsumption, we will present an algorithm to solve it. The main motivation is pedagogical since it bears the advantage to only consider variable on one side. In order to ensure correctness of the algorithm, the required formal definitions of *pconf* and *pcover* are introduced. As a result, the algorithm Check_Match is proven to be correct making matching in $\mathcal{EL}_{\mathcal{RV}}$ decidable. We postpone discussion about complexity of the problem to the next chapter since maching, weak-subsumption and pattern containment are in the same class of complexity. Table 4.1 summarizes notations introduced in this chapter.

Symbol	Description
$((q, \vec{I}), c, \mathcal{M}_q)$	Product configuration (pconf)
$\mathcal{M}_q$	Variable assignment
$\lhd$	Product cover (pcover)
$\mathcal{M}_{q \mapsto c}$	Mappings of outgoing transitions from q into c
$Exec_{A_P, A_C}$	Product execution tree
$PT(A_P, \sigma)$	Partial configuration tree

Table 4.1: Notations introduced in Chapter 4

4.1 Presentation of Check_Match

This section emphasizes on solving matching in $\mathcal{EL}_{\mathcal{RV}}$ using its reduction to universal simulation. Given an $\mathcal{EL}_{\mathcal{RV}}$-description C and an $\mathcal{EL}_{\mathcal{RV}}$-pattern Q, to solve $C \sqsubseteq^?_{\mathcal{T}} Q$, the algorithm Check_Match will test universal simulation between A_Q and A_C. This algorithm is inspired from product automata. The main idea of Check_Match is to run synchronously A_Q and A_C, trying at each step to guess appropriate value assignments to variables of A_Q in order to construct σ, called hereafter a witness substitution, such that there is a homomorphism from $T(A_Q, \sigma)$ into $T(A_C, id)$. Recall that $T(A_C, id)$ is the unique configuration tree of A_C (since A_C is ground). A given state in such a synchronized product is called a *pconf* (for product configuration).

In the rest of this section, we define formally the notions of *pconf* and *pcover* before presenting the algorithm Check_Match.

Definition 14 (*pconf*)**.** *Let A_Q be an $\mathcal{EL}_{\mathcal{RV}}$-description automata and A_C be a ground $\mathcal{EL}_{\mathcal{RV}}$-description automata. A pconf is a triple $((q, \vec{I}), c, \mathcal{S}_Q)$ where $(q, \vec{I})$ is a configuration of A_Q, c is a state of A_C and $\mathcal{S}_Q$ is a mapping from $\mathcal{V}ar_Q \times \mathbb{N}$ into N_R.*

The domain of $\mathcal{S}_Q$ denoted $dom(\mathcal{S}_Q) \subseteq \mathcal{V}ar_Q \times \mathbb{N}$ corresponds to the variable instances that appear in $\mathcal{S}_Q$. An assignment $\mathcal{S}_Q$ is inconsistent if there exists $(x, i) \in dom(\mathcal{S}_Q)$ such that $\mathcal{S}_Q(x, i) = a$, $\mathcal{S}_Q(x, i) = b$ and $a \neq b$ with a and b two constants.

Presence of counters implies an infinity of configurations from A_Q. Consequently, an infinity of *pconfs* can be produced accordingly. Thus leading any naive exploration algorithm to an infinite run. In order to prevent from such situations, we present the notion of *pcover* to cut exploration of infinite branches.

Definition 15 (pcover)**.** *Let A_Q be an $\mathcal{EL}_{\mathcal{RV}}$-description automata and A_C be a ground $\mathcal{EL}_{\mathcal{RV}}$-description automata. Let $pc = ((q, \vec{I}), c, \mathcal{S}_Q)$ and $pc' = ((q', \vec{I'}), c', \mathcal{S}'_Q)$ be two pconfs. We say that pc' is covered by pc, and we note $pc' \lhd pc$, if and only if the following conditions hold:*

(i) $c = c'$,

(ii) $q = q'$ *and,*

(iii) For all $(x, I_x) \in dom(\mathcal{S}_Q)$, there exists $(x, I'_x) \in dom(\mathcal{S}'_Q)$ such that $\mathcal{S}_Q(x, I_x) = \mathcal{S}'_Q(x, I'_x)$

Example 15. *Let consider the three following pconfs :*

$$\begin{aligned} pc_1 &= (Q_2, (3, 3), C, \{(x, 3.) = R\}) \\ pc_2 &= (Q_2, (5, 5), C, \{(x, 5) = S, (y, 5) = R\}) \\ pc_3 &= (Q_2, (6, 6), C, \emptyset) \end{aligned}$$

The two first criteria of pcover are clearly respected for all the combination of these pconf. There are no pcover relationship between pc_1 and pc_2. Indeed, if we focus on $(x, 3) = R$ of pc_1 there does not exists $(x, 5) = R$ in pc_2 which contradicts the third condition and reversely for $(x, 5) = S$. On the other hand, pc_3 having only free variables naturally leads to $pc_1 \lhd pc_3$ and $pc_2 \lhd pc_3$.

Check_Match (Algorithm 1) is based on a synchronized product of executions of description automata. Given two $\mathcal{EL}_{\mathcal{RV}}$-description automata A_C and A_Q, the algorithm tests whether $A_Q \ll_\forall A_C$. To achieve this task, Check_Match explores a search space made of *pconfs* describing a specific state of a synchronous execution of A_Q and A_C. The algorithm Check_Match starts at an initial $pconf_0 = ((Q, \vec{0}), C, \mathcal{S}_0 = \emptyset)$, corresponding to the two initial configurations of $T(A_Q, \sigma)$ and $T(A_C, id)$ with initially an empty set of variable assignments. Then the idea is to explore the synchronous product of $T(A_Q, \sigma)$ and $T(A_C, id)$ to incrementally construct a witness σ by guessing at each $pconf = ((q, \vec{I}), c, \mathcal{S}_Q)$ the appropriate values for the variables

that are free at state q in order to preserve the homomorphism from $T(A_Q, \sigma)$ to $T(A_C, id)$.

More precisely, this amounts to constructing mappings from the outgoing transitions of the configuration $(q, \vec{I})$ into the outgoing transitions of c (Algorithm 1 Line 4). Let $\mathcal{M}_{q \mapsto c}$ be the set of such mappings. Each mapping $M \in \mathcal{M}_{q \mapsto c}$ is made of a set of pairs $(((q, \vec{I}), x, (q', \vec{J})), (c, a, c'))$ indicating that the transition $((q, \vec{I}), x, (q', \vec{J}))$ of $T(A_Q, \sigma)$ is mapped to the transition (c, a, c') of $T(A_C, id)$.
The algorithm makes a non-deterministic guess among the possible mappings. For the chosen mapping, $\mathcal{S}_Q$ is extended into $\mathcal{S}'_Q$ containing the new assignments. Indeed, for each element $m_i = (((q, \vec{I}), x, (q', \vec{J})), (c, a, c'))$ of M, the assignment $(x, I_x) = a$ is generated. In other words, $\mathcal{S}'_Q = \mathcal{S}_Q \cup \bigcup_{m_i}((x, I_x) = a)$. Any free remaining variable will receive a non-deterministic assignment to a value of the domain of valuation $\mathcal{L}_C$ and stored in $\mathcal{S}_Q$. If the chosen mapping of $\mathcal{M}_{q \mapsto c}$ generates an inconsistent extension of variable assignments $\mathcal{S}'_Q$ then the Check_Match algorithm returns false (Algorithm 1 Line 7).

If the extended assignment $\mathcal{S}'_Q$ is consistent then it creates a new product configuration $pc'_i = ((q', \vec{J}), c', \mathcal{S}'_Q)$ for each element $m_i = (((q, \vec{I}), x, (q', \vec{J})), (c, a, c'))$ of M (Algorithm 1 Line 12). The algorithm then recursively calls Check_Match for each generated pc'_i. Thus the processing of the mapping M succeeds if all such calls succeed (Algorithm 1 Line 14).

Algorithm 1 Check_Match

Input : A_Q, A_C ; $pconf$: $pc = ((q, \vec{I}), c, \mathcal{S}_Q)$; Pconf's historic : Hist
Output : True if $A_Q \ll_\forall A_C$, False otherwise

1: **if** q is leaf or there exists a cover of pc in Hist **then**
2: **return** True
3: **else**
4: Compute the mappings $\mathcal{M}_{q \mapsto c}$ w.r.t. $\mathcal{S}_Q$
5: Guess $M \in \mathcal{M}_{q \mapsto c}$; let $|M| = n$
6: $\mathcal{S}'_Q \leftarrow \mathcal{S}_Q$ extended according to pairs of M
7: **if** $\mathcal{S}'_Q$ is inconsistent **then**
8: **return** False
9: **else**
10: $Hist \leftarrow Hist \cup \{pc\}$
11: **for** $m_i = ((q, \vec{I}), x, (q', \vec{J})), (c, y, c')) \in M$ **do**
12: Compute $pc'_i = ((q', \vec{J}), c', \mathcal{S}'_Q)$
13: **end for**
14: **return** $\bigwedge_{i \in [1,n]}$ Check_Match$(A_Q, A_C, pc'_i, Hist)$
15: **end if**
16: **end if**

The algorithm stops the exploration of a branch and return true when it reaches a product configuration $pc' = ((q', \vec{J}), c', \mathcal{S}'_Q)$ where q' is a leaf of $T(A_Q, \sigma)$ or when *pc covers* another product configuration already explored in the historic $Hist$ of the actual branch (Algorithm 1 Line 1).

In order to show correctness of the algorithm we need to define the execution tree of Check_Match. The next section will explain step by step how it is obtained before giving its formal definition.

4.2 Product Execution Tree of Check_Match

The algorithm Check_Match starts with the initial $pconf_0 = ((Q, \vec{0}), C, \mathcal{S}_0 = \emptyset)$. In order to construct a witness σ, the algorithm can non-deterministically choose a mapping of the set $\mathcal{M}_{q \mapsto c}$ to explore (Algorithm 1 Line 5).

Example 16. *In Figure 4.1, the algorithm starts at the initial product configuration* $pconf_0 = ((Q_2, (0,0)), C, \emptyset)$ *and tries to construct a witness* σ*. There are two outgoing transitions from the configuration* $(Q_2, (0,0))$*:*

$$\begin{aligned} t_1 &= ((Q_2, (0,0)), (x, 0), (N_{A_1}, (0,0))) \\ t_2 &= ((Q_2, (0,0)), (y, 0), (Q_2, (1,1))) \end{aligned}$$

There are three outgoing transitions from (C)*:*

$$\begin{aligned} t'_1 &= (C, R, N_{A_1}) \\ t'_2 &= (C, R, D) \\ t'_3 &= (C, S, C) \end{aligned}$$

Consequently, the set of mappings $\mathcal{M}_{Q_2 \mapsto C}$ *includes all the possible mappings from* $\{t_1, t_2\}$ *into* $\{t'_1, t'_2, t'_3\}$*. Among them, two mappings,* $M_1^\varepsilon = \{(t_1, t'_3), (t_2, t'_3)\}$ *and* $M_2^\epsilon = \{(t_1, t'_1), (t_2, t'_2)\}$ *are depicted in Figure 4.1.*
In the figure, non-deterministic choices made by the algorithm are symbolized by the $\vee$ *nodes.*

Each choice leads to consider a different product execution tree. Nonetheless, the algorithm then constructs children pc'_i issued of the chosen mapping (Algorithm 1 Line 12). Check_Match then makes a recursive call using the resulting children (Algorithm 1 Line 14). This recursive call tries to complete the witness σ for the reached children. There are three possibilities for a child :

- A child fulfills cover criteria.
- A child fails to produce a consistent mapping.
- A child makes a non-deterministic choice.

Example 17. *In Figure 4.1, the two mappings are represented for* $pconf_0$*, each of them leading to specific execution trees.*

Let focus on the first mapping M_1^ε which is represented in the box M_1^ε of the figure. The reached children 0 and 1 are respectively labeled by $((Q_2,(1,1)),C,\emptyset)$ and $((N_{A_1},(0,0)),C,\{(x,0)=S,(y,0)=S\})$. The node 0 is dashed because it fulfills a pcover relationship. Indeed, 0 is such that the label of the previous node ε is covered by $\lambda(0)$ therefore the algorithm stops. Regarding the node 1, the algorithm has to check the homomorphism from $(N_{A_1},(0,0))$ into C. There is no outgoing transition from C labeled by A_1, consequently $((N_{A_1},(0,0)),A_1,(q_f,(0,0)))$ can not be mapped into any outgoing transition from C. The set of mappings is then empty therefore the algorithm returns false *(symbolized by the cross in the figure). All execution trees corresponding to M_1^ε are then completed and correspond to a failing run since there are no valid mapping.*
We will now look toward M_2^ε which is represented in the box labeled M_2^ε. This mapping produces children 0 and 1 such that $\lambda(0)=((Q_2,(1,1)),D,\emptyset)$ and $\lambda(1)=((N_{A_1},(0,0)),N_{A_1},\emptyset)$. None of them are covered and their corresponding sets of mappings M^0 and M^1 are not empty.

A branch of a product execution tree is completed once it reached a covering node, a leaf or fails to find a simulation. If it continues, the algorithm guesses for each child a new mapping. Each non-deterministic choice of mapping and thus their combination leads to a different product execution tree. Chosen mappings may have children that will recursively call the algorithm. To differentiate these choices, we say that a product execution tree is associated to a combination of mappings M.

Example 18. *Figure 4.1 explores two possibilities accessible from the mapping M_2^ε based on the mappings produced by the nodes 0 and 1.*
The only mapping possible for the node 1, M_1^1 maps $((N_{A_1},(0,0)),A_1,(q_f,(0,0)))$ into (N_{A_1},A_1,q_f). Note that the reached $pconf$ contains the leaf q_f so the algorithm naturally stops. There are two outgoing transitions from the configuration $(Q_2,(1,1))$:

$$\begin{aligned} t_1 &= ((Q_2,(1,1)),(x,1),(N_{A_1},(2,2))) \\ t_2 &= ((Q_2,(1,1)),(y,1),(Q_2,(2,2))) \end{aligned}$$

There are three outgoing transitions from (D):

$$\begin{aligned} t'_1 &= (D,S,N_{A_1}) \\ t'_2 &= (D,S,C) \\ t'_3 &= (D,R,D) \end{aligned}$$

The figure illustrates two mappings $M_1^0 = \{(t_1,t'_3),(t_2,t'_3)\}$ and $M_2^2 = \{(t_1,t'_1),(t_2,t'_2)\}$. The mapping M_1^0 fails for the same reasons as M_1^ε. Regarding M_2^0, the reached children are 00 with $\lambda(\varepsilon) \lhd \lambda(00)=((Q_2,(2,2)),C,\emptyset)$ and 01 which is similar to 1 so the execution will naturally reach a leaf by choosing the only mapping possible M_1^{01}.
A successful product execution tree corresponds to the mapping M with $M = M_2^\varepsilon \cup M_2^0 \cup M_1^1 \cup M_1^{01}$.

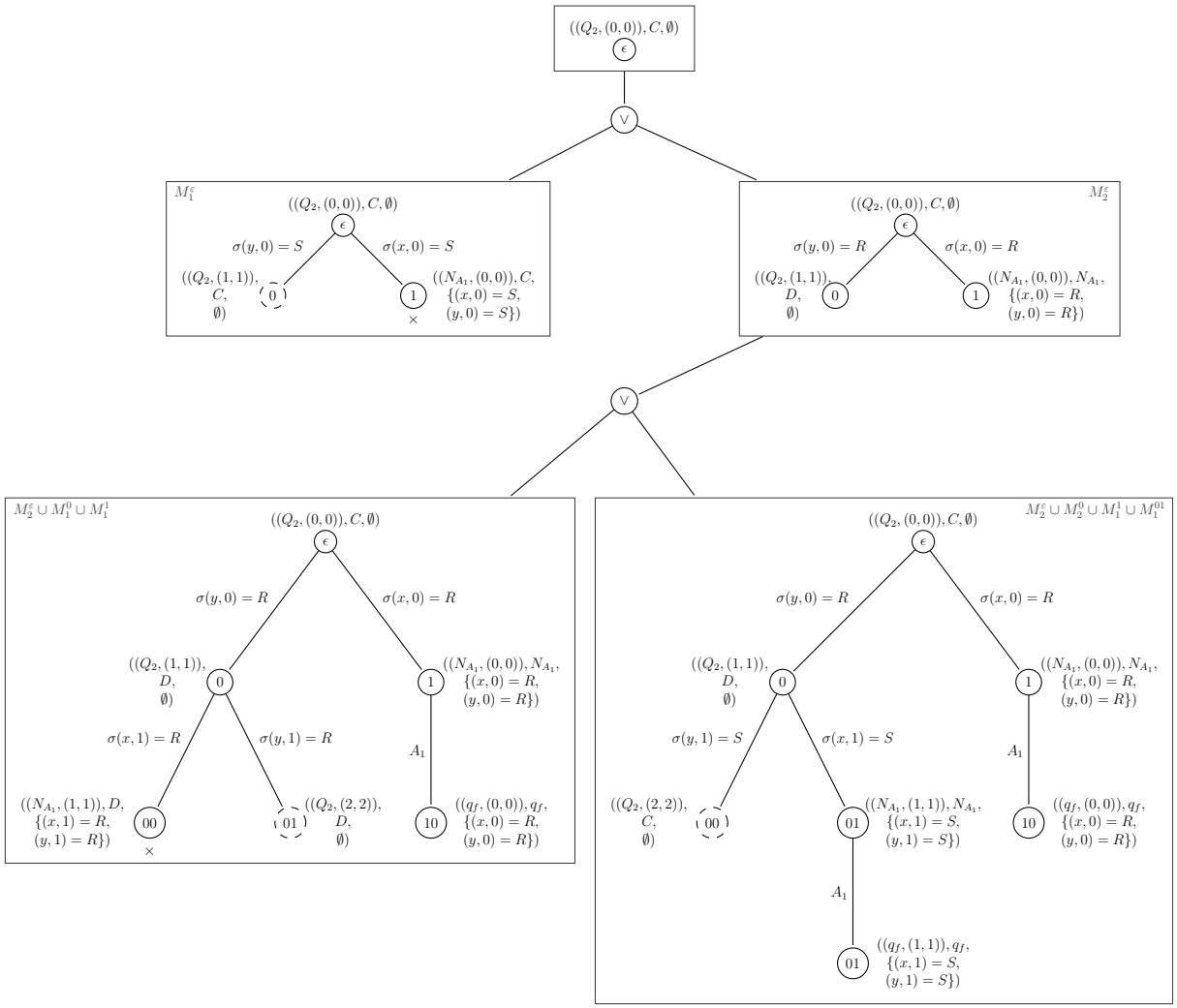


Figure 4.1: Fragment Execution Trees of Check_Match

The notion of product execution tree of Check_Match will then be formally defined with regard to a mapping M. This mapping is a union of mappings corresponding to the recursive choices the algorithm made.

Definition 16 (Product execution tree of Check_Match w.r.t to M)**.** *Let* $A_Q = (\mathcal{Q}_Q, \mathcal{L}_Q, \mathcal{V}ar_Q, q_0, \delta_Q, q_f, \kappa_Q)$ *and* $A_C = (\mathcal{Q}_C, \mathcal{L}_C, \mathcal{V}ar_C, c_0, \delta_C, c_f, \kappa_C)$ *be two description automata. Let* $\overrightarrow{G} \in \mathbb{N}^{|\mathcal{V}ar_Q|}$ *be a counter and let* S *be the set of all possible* $pconf$*. A run of Check_Match, denoted* $Exec_{A_Q,A_C}$ *and called a product execution tree, is a* $\langle S, \mathcal{L}_C\rangle$*-labeled tree* (τ, λ, δ) *constructed as follows:*

- $\lambda(\varepsilon) = ((q_0, \overrightarrow{0}), c_0, \emptyset)$*, a root of the tree, and let* $\overrightarrow{G} = \overrightarrow{0}$*, a global counter initialized to a vector of zero.*

- *Let* $n \in \tau$ *be a node such that* $\lambda(n) = ((q, \overrightarrow{I}), c, \mathcal{S}_q)$ *For each* m_i *in* $((q, \overrightarrow{I}), t, (q', \overrightarrow{J})), (c, a, c')) \in M$*,a new child ni of* n *is generated according to the following sequence:*

$$(i)\ \delta(n, ni) = \begin{cases} \mathcal{S}_q(t, I_t) = a, & t \in \mathcal{V}ar_Q \\ a & t \in \mathcal{L}_C \end{cases}$$

$$(ii)\ \forall y \in \mathcal{V}ar,\ if\ q' \in \kappa(y)\ then\ G_y := G_y + 1,\ and$$

$$(iii)\ Let\ \vec{J} = \begin{cases} J_y = G_y, & \forall y \in \mathcal{V}ar,\ s.t.\ q' \in \kappa(y) \\ J_y = I_y & otherwise \end{cases}$$

$$(iv)\ \lambda(ni) = ((q', \vec{J}), c', \mathcal{S}'_q)$$
$$with\ \mathcal{S}'_q = \mathcal{S}_q \cup \{(x, I_x) \to a | (((q, \vec{I}), x, (q', \vec{J})), (c, a, c')) \in M\}.$$

Everything required to prove correctness of the algorithm is now defined. The next section will then detail termination, soundness and completeness of Check_Match.

4.3 Correctness of Check_Match

4.3.1 Termination of Check_Match

A run of the algorithm with the initial $pconf$ leads to an exploration of its associated execution tree. An execution tree may be infinite, however we aim here to prove that only a finite part is explored leading to the algorithm's halt. The algorithm stops exploring a branch if one of the following occurs:

(i) A leaf has been reached.

(ii) The chosen mapping $\mathcal{M}_{q \mapsto c}$ of a visited $pconf$ failed.

(iii) A pconf fulfills cover criteria.

The two first cases will not be discussed since by definition they terminate. It remains to discuss potential infinite branches. In our context, it means that there exists an infinite sequence of $pconfs$ without $pcover$. The following property ensures that using $pcover$ prevents from an infinite branch.

Property 1. *Let $A_Q = (\mathcal{Q}_Q, \mathcal{L}_Q, \mathcal{V}ar_Q, q_0, q_f, \delta_Q, \kappa_Q)$ be a description automaton and $A_C = (\mathcal{Q}_C, \mathcal{L}_C, \mathcal{V}ar_C, c_0, c_f, \delta_C, \kappa_C)$. For any infinite sequence of pconf $pc_1.pc_2.pc_3...pc_n$, there exists $i < j$ such that $pc_i \lhd pc_j$.*

Proof. Let assume the infinite sequence of $pconf$ $pc_1.pc_2.pc_3...pc_n$. $\mathcal{Q}_P$ and $\mathcal{Q}_C$ are finite sets. As a direct consequence, there exists $q_q \in \mathcal{Q}_Q$ such that there are an infinite number of pc_k of the form $(q_k = (q_q, I_k), c, \mathcal{S}_{q_k})$ withing the infinite sequence.

Let focus on this infinite set of pc_k. For any $pconf$ there are up $|D|^{|\mathcal{V}ar_Q|}$ different possible variable assignments with the finite domain $D = \mathcal{L}_\mathcal{C}$. Therefore, in the considered infinite set, there are at least two $pconf$ with the same assignment.

Consequently, if we take $pc_k = (q_k = (q_q, I_k), c_k = c, \mathcal{S}_{q_k})$ as defined above then there exists $k < k'$ such that $pc_{k'} = (q_{k'} = (q_q, I'_{k'}), c_{k'} = c, \mathcal{S}_{q_{k'}})$ and the following can be observed :

1. $c_k = c_{k'}$,
2. $q_q = q_q$ and

3. For all $(x, I_x) \in dom(\mathcal{S}_{q_k})$, there exists $(x, I'_x) \in dom(\mathcal{S}_{q_k})$ such that $\mathcal{S}_{q'_k}(x, I_x) = \mathcal{S}_{q_{k'}}(x, I'_x)$

Consequently, pc_k and pc'_k are such that $pc_k \lhd pc'_k$ and $k < k'$ which concludes the proof. □

As a direct consequence Check_Match has the following property.

Property 2. *The algorithm Check_Match terminates.*

4.3.2 Soundness of Check_Match

A run of the algorithm with the initial *pconf* leads to an exploration of its associated execution tree. We will consider here, $Exec_{A_Q,A_C} = (\tau, \lambda, \delta)$ corresponding to a successful run of Check_Match.

From $Exec_{A_Q,A_C}$, we are able to extract a partial configuration tree $PT(A_Q, \sigma)$ of A_Q and a partial configuration tree $PT(A_C, id)$. The algorithm ensures the following properties :

(i) There is a homomorphism from $PT(A_Q, \sigma)$ into the partial-tree of $PT(A_C, id)$.

(ii) $PT(A_Q, \sigma)$ and $PT(A_C, id)$ can be extended to configuration trees $T(A_Q, \sigma)$ and $T(A_C, id)$ such that there is a homomorphism between $T(A_Q, \sigma)$ into $T(A_C, id)$ (Lemma 6).

Based on $Exec_{A_Q,A_C} = (\tau, \lambda, \delta)$, we define the partial trees $PT(A_Q, \sigma) = (\tau_Q, \lambda_Q, \delta_Q)$ and $PT(A_C, id) = (\tau_C, \lambda_C, \delta_C)$ as follows :

- $\forall n \in \tau$, $\lambda(n) = ((q, \vec{I}), c, \mathcal{S}_q)$ then $\lambda_Q(n) = (q, \vec{I})$ and $\lambda_C(n) = (c)$.
- $\delta_Q(n, ni) = \delta_C(n, ni) = \delta(n, ni)$.

Configuration trees are associated to a function σ, the corresponding σ of this tree is the union of all variable assignments of the run, i.e. $\sigma = \bigcup \mathcal{S}_q$. By construction, it is clear that there is a homomorphism between those trees (i.e. $Z(i) = i$).

These trees are partial configuration trees for two reasons :

(i) A branch has been cut by *pcover* preventing from the infinite branch of $T(A_Q, \sigma)$ or $T(A_C, id)$.

(ii) Only outgoing transitions of A_C required to mimic A_Q are explored. Some outgoing transitions of A_C may not be explored by the algorithm.

Example 19. *Figure 4.2 depicts how to extract trees from $Exec_{A_{Q_2},A_C}$ on the running example. Based on the successful product execution tree presented before, we extract a configuration tree of A_{Q_2} by keeping only configurations of A_{Q_2} in the labels. For example, the node ε (labeled by $((Q_2, (0, 0)), c, \emptyset)$ in $Exec_{A_Q,A_C}$) is labeled by $(Q_2, (0, 0))$ in the extracted tree (which is exactly the label of the root of a configuration tree of A_{Q_2}). The corresponding witness σ is made of all the mapping choices : $\sigma = \{(y, 0) = S, (x, 0) = R, (y, 1) =, (x, 1) = S\}$*

We apply the same process to extract a partial tree of $T(A_C, id)$. For instance, it leads to C for the label of the node ε. Since C is ground, the substitution function which is the identity does not appear explicitly. This figure also illustrates why the resulting tree is only a partial-tree. The transition (C, S, C) of A_C should have led to the edge $(\varepsilon, S, 2)$ with $\lambda_C(2) = C$. However, it was not used during the simulation process which makes the obtained tree incomplete.

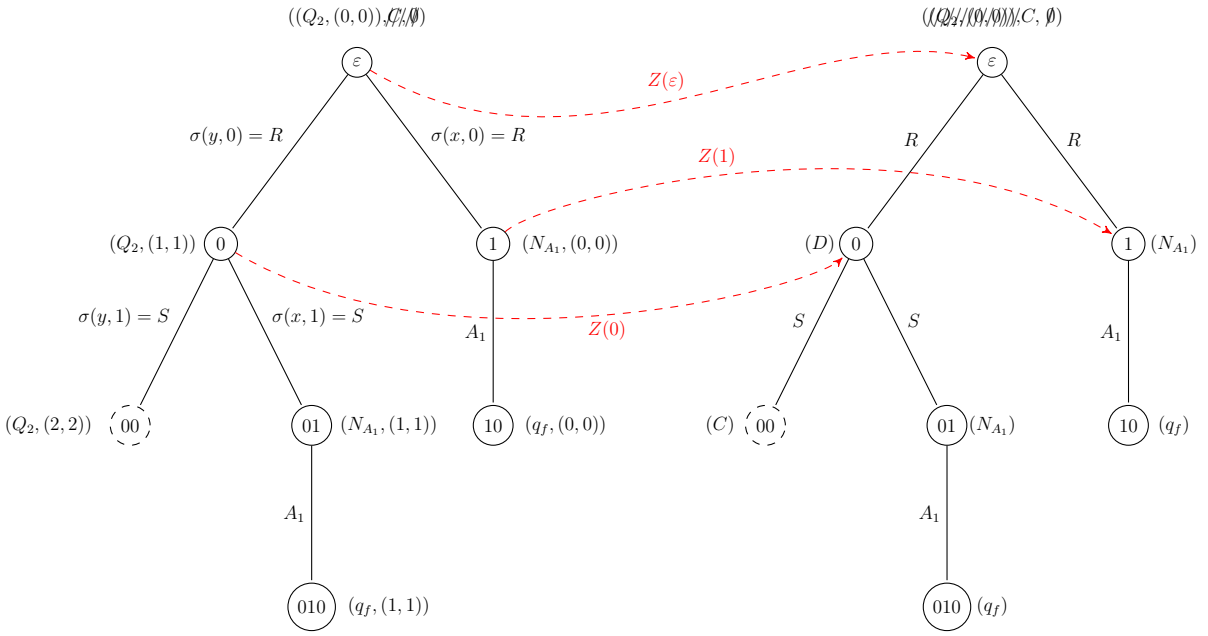


Figure 4.2: Partial Configuration Trees Extracted from $Exec_{A_Q,A_C}$

It remains to show that *pcover* does not damage the existence of a homomorphism. The intuition is to exploit the characteristics of the *pcover*. The space explored by a covering *pconf* is wider than the space explored by a covered one (Definition 15 Item (iii)). Therefore if the covered one succeeded, the covering one, which encloses even more possibilities, will succeed.

Lemma 6. *Let $A_Q = (\mathcal{Q}_Q, \mathcal{L}_Q, \mathcal{V}ar_Q, q_0, q_f, \delta_Q, \kappa_Q)$ be a description automaton and $A_C = (\mathcal{Q}_C, \mathcal{L}_C, \mathcal{V}ar_C, c_0, c_f, \delta_C, \kappa_C)$ be a ground description automaton. For any pconf $pc_i = ((q_i, \vec{I}), c_i, \mathcal{S}_i)$ and $pc_j = ((q_j, \vec{J}), c_j, \mathcal{S}_j)$ with $i < j$ and such that $pc_i \lhd pc_j$. Then $(q_j, \vec{J}) \ll_\forall (q_i, \vec{I})$ and $c_i \ll_\forall c_j$*

Proof. Since $c_i = c_j$ by definition of *pcover* this part is trivial.
Let focus on $(qj, \vec{J}) \ll_\forall (q_i, \vec{I})$ by definition of *pcover* we have :

- $q_i = q_j$ and
- For all $(x, J_x) \in dom(\mathcal{S}_j)$, there exists $(x, I_x) \in dom(\mathcal{S}_i)$ such that $\mathcal{S}_i(x, I_x) = \mathcal{S}_j(x, J_x)$

Then for any outgoing edge of $(q, \vec{J})$ which is of the form $((q, \vec{J}), x, (q', \vec{J'}))$ there is an outgoing edge $((q, \vec{I}), x, (q', \vec{I'}))$.

The different cases are :

- x is a constant which is trivial.
- (x, I_x) is free (i.e. $(x, I_x) \notin dom(\mathcal{S}_i)$) then (x, J_x) is free. For all σ such that $\sigma(x, I_x) = a$, we can construct σ' such that $\sigma'(x, J_x) = a = \sigma(x, I_x)$.
- (x, I_x) is not free (i.e. $\mathcal{S}_i(x, I_x) = a$) and (x, J_x) is free then we can take $\mathcal{S}_j(x, J_x) = a$.
- (x, I_x) is not free and (x, J_x) is not free either. We have $\mathcal{S}_i(x, I_x) = \mathcal{S}_j(x, J_x) = a$ by definition of *pcover*.

The reached configuration $(q', \vec{I'})$ and $(q', \vec{J'})$ are such that $(q', \vec{J'}) \lhd (q', \vec{I'})$. The same reasoning can then be recursively applied. Thus leading to $(q_j, \vec{J}) \ll_\forall (q_i, \vec{I})$. □

While looking for a witness, the algorithm checks the *pconf* $pc_i = ((q_i, \vec{I}), c_i, \mathcal{S}_i)$ in order to have $(q_i, \vec{I}) \ll_\forall c_i$. It stops when a leaf is reached which is a trivial or when it finds a cover $pc_j = ((q_j, \vec{J}), c_j, \mathcal{S}_j)$ for this configuration (i.e. $pc_i \lhd pc_j$). In this case, $(q_i, \vec{I}) \ll_\forall c_i$ has been verified during the run. Therefore, thanks to Lemma 6 and transitivity of universal simulation, we have $(q_j, \vec{J}) \ll_\forall c_j$.

Property 3. *The algorithm Check_Match is sound.*

The finite automaton corresponding to the witness can be constructed by fusing covered nodes. Thus leading to consider σ repeating itself with a regular pattern. The witness will then takes the shape of a finite set of definitions giving a regular matcher. Nodes labeled with ground-description are also merged independently of their counters.

Example 20. *The Figure 4.3 depicts the construction of the ground automaton corresponding to the generated witness. To be more precise the nodes ε and 00 are fused and the resulting nodes is named $\sigma(Q_2)$. The node 0 is renamed $\sigma(Q'_2)$ because it represents a different valuation of variables. Regarding the couples $(01, 1)$ and $(010, 10)$, each member of a same couple are fused since they represent the same ground concept. The resulting nodes are respectively named after the ground concept: N_{A_1} and q_f. This automaton is then translated into a fnite TBox $\sigma(\mathcal{T}^{Q_2})$.*

$$\sigma(\mathcal{T}^{Q_2}) = \left\{ \begin{array}{ll} \sigma(Q_2) & \equiv \exists R.A_1 \sqcap \exists R.\sigma(Q'_2), \\ \sigma(Q'_2) & \equiv \exists S.A_1 \sqcap \exists S.\sigma(Q_2) \end{array} \right\}$$

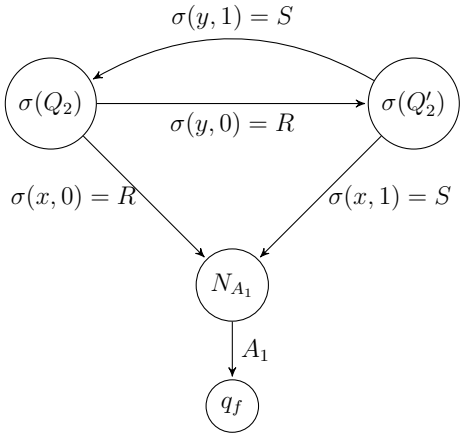


Figure 4.3: Witness of $C \sqsubseteq^? Q_2$

4.3.3 Completeness of Check_Match

Lemma 7. *The algorithm Check_Match is complete.*

Proof. Assume that there exists a witness σ_0 such that there is a homomorphism Z from $T(A_Q, \sigma_0)$ into $T(A_C, id)$. Since the set of mappings $\mathcal{M}_{q \mapsto c}$ are exhaustive, we can use Z to lead the algorithm to choose the mappings corresponding to σ_0.
Thanks to the fact that the algorithm terminates (Property 2), the algorithm necessarily stops. If the algorithm answers false, it means that the variable assignment is not consistent which contradicts the fact that σ_0 is a witness. If the algorithm answers true, the algorithm may have stopped while constructing a witness σ. Due to *pcover*, we may have $\sigma \neq \sigma_0$. However, soundness (Property 3) ensures that σ is also a solution. Consequently, if a solution exists the algorithm answers yes.

□

The proof of the Lemma 7 shows that given a solution to lead the algorithm, there necessarily exists a regular solution σ resulting in the following corollary.

Corollary 2. *An $\mathcal{EL}_{\mathcal{RV}}$ matching problem $C \sqsubseteq^?_{\mathcal{T}} Q$ has a solution iff it has a regular solution.*

Check_Match has been proven to terminate as well as being sound and complete making matching in $\mathcal{EL}_{\mathcal{RV}}$ decidable.

4.4 Conclusion of Chapter 4

This chapter presents the algorithm Check_Match which demonstrates that matching in $\mathcal{EL}_{\mathcal{RV}}$ is decidable. Its main idea is to run synchronously the two automata A_Q and A_C leading to consider product configurations that combine a configuration of A_Q, a configuration of A_C to keep track of the synchronous run and a mapping $\mathcal{S}_Q$ to preserve variable assignments.

This result is used as a baseline for the next chapter which will extends the proposed algorithm to both pattern containment and weak-subsumption.

Chapter 5

Solving Pattern Containment in $\mathcal{EL}_{\mathcal{RV}}$

This chapter represents the main contribution of this thesis. It consists in solving pattern containment by designing an algorithm named *Check_Simu*. The algorithm follows the idea of the synchronous product of Check_Match while introducing the notion of constraints to handle variables on both sides. It leads to the definition of extended product configuration and extended cover with regard to a set of constraints. The algorithm is proven to be correct making pattern containment decidable. Moreover, we study the complexity of pattern containment and more generally of the reasoning tasks in $\mathcal{EL}_{\mathcal{RV}}$. We prove that it is EXPTIME-COMPLETE. The lower-bound is achieved by a reduction of matching problem to halting problem of alternating turing machine which is polynomially bounded by the size of the input. Table 5.1 summarizes notations introduced in this chapter.

Symbol	Description
$pc = ((q_q, \vec{I}), (q_p, \vec{J}))$	Extended product configuration (epconf)
$\mathbb{C}$	Set of constraints
$R_{\mathcal{V}}(q)$	Relevant constraints w.r.t $q = (q_q, \vec{I})$
$\mathbb{C}_{\|q}$	Relevant set of constraints w.r.t q
$QF(\mathbb{C})$	Quantified constraint associated to $\mathbb{C}$
$\lhd_{\mathbb{C}}$	Extended cover w.r.t a set of constraints $\mathbb{C}$

Table 5.1: Notations introduced in Chapter 5

5.1 From Matching to Pattern Containment

We previously demonstrated how to solve matching, a special case of pattern containment. The approach presented is the root of our method to solve pattern containment. However, some differences must be enlighten in order to extend the algorithm. The main difference lies in variable's domain of valuation. Matching focuses on mapping variables into a finite domain made of constants appearing in the ground definition C. Considering variables on both sides requires mapping function to consider mapping :

(i) A variable into a constant,

(ii) A constant into a constant and,

(iii) A variable into a variable.

It enlarges the size of the valuation domain to an infinite size. It is explained by the fact that there are an infinite number of variables induced by refreshment and cycles. Moreover, the domain of valuation of variables can not be limited to constant of P and Q as we did for matching. Indeed, the pattern containment problem asks for such a relationship w.r.t to any TBox. The following example will illustrate the impact of such a constraint. Let consider the following patterns :

- $\mathcal{T}^Q = \{Q \equiv \exists z.Q \sqcap \exists y.A_2 \sqcap \exists x.A_1\}$
- $\mathcal{T}^{P_1} = \left\{ \begin{array}{lcl} P_1 & \equiv & \exists R.P_2 \sqcap \exists R.P_3 \\ P_2 & \equiv & \exists S.P_1 \sqcap \exists R.P_3 \sqcap w.P_3 \\ P_3 & \equiv & A_1 \sqcap A_2 \end{array} \right\}$
- $\mathcal{T}^{P'_1} = \left\{ \begin{array}{lcl} P'_1 & \equiv & \exists R.P'_2 \sqcap \exists R.P'_3 \\ P'_2 & \equiv & \exists S.P'_1 \sqcap \sqcap w.P'_3 \\ P'_3 & \equiv & A_1 \sqcap A_2 \end{array} \right\}$

The variable x is a non-refreshing variable while all the others variables are refreshing variables. Let us first focus on the the pattern containment problem $P'_1 \sqsubseteq Q$. It is worth to note the answer changes if one considers a specific TBox instead of "for all TBoxes" of pattern containment's definition. Indeed, $P'_1 \sqsubseteq Q$ is not solvable for all the ground TBox but only for the ones dealing with $N_R = \{R\}$. For instance, considers a $TBox$ that involves $N_R = \{R, S\}$, then for the substitution σ such that $\sigma(w, i) = S$ there exists no substitution θ such that the subsumption relationships holds. On the other hand, the pattern containment problem $P_1 \sqsubseteq Q$ is solvable and will be used as a running example in the remaining discussion about pattern containment. Figure 5.1 depicts the automata related to those definitions.

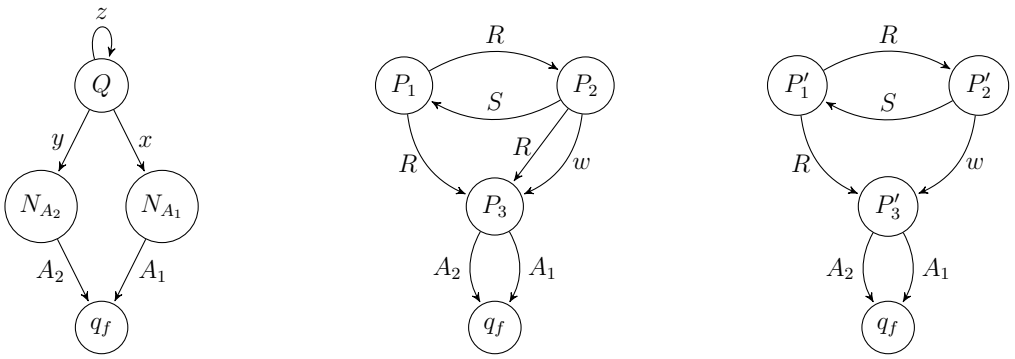


Figure 5.1: Description Automata of Respectively Q, P_1 and P'_1.

5.2 Presentation of Check_Simu

This section presents the algorithm *Check_Simu*, which is used to test universal simulation between $\mathcal{EL}_{\mathcal{RV}}$-description automata. Let $A_Q = (\mathcal{Q}_Q, \mathcal{L}_Q, \mathcal{V}ar_Q, q_0, \delta_Q, q_f, \kappa_Q)$ and $A_P = (\mathcal{Q}_P, \mathcal{L}_P, \mathcal{V}ar_P, p_0, \delta_P, p_f, \kappa_P)$ be two description automata. To test whether $A_Q \ll_\forall A_P$, the main idea behind *Check_Simu* is to run synchronously A_Q and A_P, trying to construct iteratively two *generic* instantiations θ_G and σ_G of, respectively, A_Q and A_P, which ensure that $\forall\sigma, \exists\theta$ such that there exists a homomorphism from $T(A_Q, \theta)$ to $T(A_P, \sigma)$. The two substitutions are called *generic* because θ_G maps variables of A_Q to constants or variables that appear in an execution of A_P (i.e.: $\theta_G : \mathcal{V}ar_Q \times \mathbb{N} \mapsto \mathcal{L}_P \cup (\mathcal{V}ar_P \times \mathbb{N})$) while σ_G is the identity function. More precisely, the algorithm *Check_Simu* runs A_Q and A_P trying to generate two *generic* configurations trees $T(A_Q, \theta_G) = (\tau, \lambda_1, \delta_1)$ and $T(A_P, \sigma_G) = (\tau, \lambda_2, \delta_2)$ over the same tree structure τ. The trees $T(A_Q, \theta_G)$ and $T(A_P, \sigma_G)$ are synchronized in the sense that when processing a node $i \in \tau$, with $\lambda_1(i) = (q, \vec{I})$ and $\lambda_2(i) = (p, \vec{J})$, the algorithm maps each transition (q, t, q') of A_Q to a transition (p, t', p') in A_P such that $t = t'$. To achieve this task, the algorithm creates a new child ij with $\lambda_1(ij) = (q', \vec{I'})$ and $\lambda_2(ij) = (p', \vec{J'})$ and generates a new equality constraint $\delta_1(i, ij) = \delta_2(i, ij)$ to ensure that the two transitions are synchronized. Note that $\delta_1(i, ij) = \theta_G(t, I_t)$ if t is a variable of A_Q and $\delta_1(i, ij) = t$ is t is a constant in A_Q. Similarly, $\delta_2(i, ij)$ is either a constant t' or $\sigma_G(t', I_{t'})$ if t' is a variable of A_P. Consequently, a generated constraint has the form of an equality between either a constant or $\theta_G(t, I_t)$ from one side and a constant or $\sigma_G(t, I_t)$ from the other side (c.f., definition 17). The generated set of constraints is quantified to obtain a quantified constraint formula where variables in $T(A_Q, \theta_G)$ are existentially quantified while variables of $T(A_P, \sigma_G)$ are universally quantified (c.f., definition 19). By construction, the algorithm ensures that the identity mapping $Z_{id} : \tau \to \tau$ such that $Z_{id}(i) = i$ is a homomorphism from $T(A_Q, \theta_G)$ to $T(A_P, \sigma_G)$. In addition, a second important property ensured by the algorithm is that if the generated formula is satisfiable then one can prove that for any substitution σ, there exists $\theta = \sigma \circ \theta_G$ such that there is a homomorphism from $T(A_Q, \theta)$ into $T(A_P, \sigma)$ i.e. $A_Q \ll_\forall A_P$.

In the remaining, we define the notion of *constraints*, *epconf* (extended product configuration) and *epcover* (extended product cover) to formally present the algorithm *Check_Simu*. Variables of A_P are referred as universal variables since A_P is associated to the universal quantifier in pattern containment definition. Similarly, variables of A_Q are referred to as existential variables.

Definition 17 (Constraint)**.** *Let $A_Q \ll_\forall A_P$ be a universal simulation problem. An associated constraint is a statement of one of the following forms:*

- $(x, i) = (y, j)$,
- $(x, i) = a$, *or*
- $a = b$

where x, y are variables from $\mathcal{V}ar_Q \cup \mathcal{V}ar_P$, $i, j \in \mathbb{N}$ and a,b are constants from $\mathcal{L}_Q \cup \mathcal{L}_P$.

In the following $\mathbb{C}$ denotes a set of constraints and $\mathbb{C}^T$ the transitive closure of $\mathbb{C}$ w.r.t. to the equality relationship. The following definition formalizes the notion of $epconf$.

Definition 18 (epconf)**.** *Let A_Q, A_P be two description automata. An epconf is a tuple (q, p) where q is a configuration of A_Q, p is a configuration of A_P.*

Example 21. *For example, we can consider the following epconfs :*

- $((Q, (0, 0, 0)), (P_1, (0)))$ *and;*
- $((Q, (0, 1, 1)), (P_3, (3)))$.

When testing $A_Q \ll_\forall A_P$, we want to check out *for all* valuations of variables in A_P there *exists* a valuation of variables in A_Q. Therefore, universal simulation requires to quantify variables involved in constraints in order to preserve this information.

To preserve this semantics, we will exploit the notion of *quantified constraints formula*. It allows to consider universal quantifier in addition to the classic existential quantifier.

Definition 19 (Quantified constraint formula)**.** *Let $A_Q \ll_\forall A_P$ be a universal simulation problem and let $\mathbb{C} = \{\psi_1, \ldots, \psi_n\}$ be a set of associated constraints using variables $x_1, \ldots, x_m$ from $\mathcal{V}ar_P$ and variables $y_1, \ldots, y_l$ from $\mathcal{V}ar_Q$. The quantified constraint formula associated to $\mathbb{C}$, denoted $QF(\mathbb{C})$, is of the form $\forall x_1, \ldots \forall x_m, \exists y_1, \ldots, \exists y_l(\psi_1 \wedge ... \wedge \psi_j)$.*

All constraints ψ_i can only be equality constraint as stated in Definition 17. In this context, quantified constraint formula corresponds to quantified constraint for positive equality constraint languages. Our objective is to ensure that considered quantified constraint formula are satisfiable for infinite domain since we aims to solve it for any TBox. Satisfiability for quantified constraint formula w.r.t positive equality constraint languages and infinite domain is proven to be NP-COMPLETE [21].

We say that a set of constraints is satisfiable if the corresponding quantified constraint formula is satisfiable. It is clear that any instance of a variable of $\mathcal{V}ar_P$ can be mapped to only one instance of $\mathcal{V}ar_Q$ (or a constant). Otherwise, the quantified formula generated would inevitably be unsatisfiable.

Example 22. *Let x, y, z be variables and a be a constant.*

- $\forall(w, 0), \forall(w, 1), \exists(z, 0), (((w, 0) = (z, 0)) \wedge ((w, 1) = (z, 0)))$ *is not satisfiable. Indeed, we can infer $\forall(w, 0), \forall(w, 1), (w, 1) = (w, 0)$ which is unsatisfiable since w is a universal variable.*
- $\forall(w, 0), \exists(y, 0), (((w, 0) = (y, 0)) \wedge ((y, 0) = R))$ *is not satisfiable. Indeed, we can infer $\forall(w, 0), (w, 0) = R$ which is not satisfiable when dealing with infinite valuation domains.*
- $\forall(w, 0), \exists(y, 0), \exists(z, 0), (((w, 0) = (y, 0)) \wedge ((z, 0) = R))$ *is satisfiable.*

Definition 20 (Constraint set equivalence)**.** *Let $\mathbb{C}_1$ and $\mathbb{C}_2$ be two sets of constraints. The set $\mathbb{C}_1$ is equivalent to $\mathbb{C}_2$, noted $\mathbb{C}_1 \equiv \mathbb{C}_2$ if :*

1. $(x, i) = (y, j) \in \mathbb{C}_1$ *iff* $(x, i') = (y, j') \in \mathbb{C}_2$
2. $(x, i) = a \in \mathbb{C}_1$ *iff* $(x, i') = a \in \mathbb{C}_2$

Example 23. *Let the following sets of constraints :*

- $\mathbb{C}_1 = \{((w,1) = (z,0))\}$
- $\mathbb{C}_2 = \{((w,0) = R), ((w,1) = (z,0))\}$
- $\mathbb{C}_3 = \{((w,3) = R), ((w,4) = (z,1))\}$

Following the definition of set equivalence (Definition 20), we have : $\mathbb{C}_3 \equiv \mathbb{C}_2$. *More precisely, for* $(w,0) = R$ *we have* $(w,3) = R$ *and for* $(w,1) = (z,0)$ *we have* $(w,4) = (z,1)$. $\mathbb{C}_1$ *does not have any constraint of the form* $(w,j) = R$ *which makes it not equivalent to the other two.*

Relevance of a constraint depends on the *epconf* it is associated with. For example, let us consider $\mathbb{C} = \{(x,2) = (w,2), (x,1) = S, (w,4) = (x,3)\}$ and the *epconf* $pc = ((N_{A_1}, (3,3,3)), (P,(4)))$. The configuration $(P,(4))$ has only one counter, whose current value is 4 and we assume that this counter is associated with the variable w. Therefore, we know that the only constraint that constrains $(P,(4))$ is $(w,4) = (x,3)$. Indeed, starting from $(P,(4))$, only the instance $(w,4)$ must be considered therefore $(x,2) = (w,2)$ does not carry any information for this configuration. We introduce the notion of *relevant* variables which limits constraint sets to relevant set of constraints w.r.t to a *epconf*.

Definition 21 (Relevant constraints w.r.t. a configuration c)**.** *Let* $\mathbb{C}$ *be a set of constraint and let* $c = (q_c, I_c)$ *be a configuration.*

- *We define the set of relevant variables of* c *as* $R_{\mathcal{V}}(c) = \{(x, I_{c_x}) | x \in \mathcal{V} \setminus \kappa^{-1}(q_c)\}$.
- *The relevant constraints of* $\mathbb{C}$ *w.r.t. a configuration* c *is defined as:* $\mathbb{C}_{|c} = \{(t_1 = t_2) \in \mathbb{C}^T | (t_1 \in R_{\mathcal{V}}(c) \vee t_2 \in R_{\mathcal{V}}(q_c)\}$.

Example 24. *Let consider the configuration* $(Q,(0,1,1))$ *of* A_Q*, the relevant variables are then* $(x,0), (y,1)$ *and* $(z,1)$. *Let the following sets of constraints :*

- $\mathbb{C}_1 = \{((w,1) = (z,0))\}$
- $\mathbb{C}_2 = \{((w,0) = R), ((w,1) = (z,0))\}$
- $\mathbb{C}_3 = \{((w,3) = R), ((w,4) = (z,1))\}$

The relevant constraints are respectively :

- $\emptyset$
- $\emptyset$
- $\{((w,4) = (z,1))\}$

We dispose now of all the tools required to define *pcover* in the scope of pattern containment problems.

Definition 22 (epcover w.r.t a set of constraints)**.** *Let* A_Q *and* A_P *be two* $\mathcal{EL}_{\mathcal{RV}}$*-description automata and let* $pc = (q,p)$ *and* $pc' = (q',p')$ *be two epconfs. Let* $\mathbb{C}$ *be a set of constraints.*

We say that pc' is covered by pc w.r.t. $\mathbb{C}$, and we note $pc' \lhd_{\mathbb{C}} pc$, if and only if the following conditions hold:

1. $q_p = q_{p'}$ *and* $\mathbb{C}_{|p} \equiv \mathbb{C}_{|p'}$
2. $q_q = q_{q'}$ *and* $\mathbb{C}_{|q} \equiv \mathbb{C}_{|q'}$

We are now ready to present our algorithm *Check_Simu* (Algorithm 2). Given two $\mathcal{EL}_{\mathcal{RV}}$-description automata A_Q and A_P, the algorithm tests whether $A_Q \ll_\forall A_P$. To achieve this task, *Check_Simu* explores a search space made of *epconf*s where each *epconf* describes a specific state of a synchronous execution of A_Q and A_P. When exploring an *epconf* (q, p), the algorithm makes non deterministic choices to map outgoing transitions of q to outgoing transitions of p while generating new constraints which are recorded in a global set of constraints. The algorithm starts with the initial *epconf* $pc_0 = ((q_0, \vec{0}), (p_0, \vec{0}))$ made of the initial configuration $(q_0, \vec{0})$ of A_Q and the initial configuration $(p_0, \vec{0})$ of A_P. The initial global set of constraints $\mathbb{C}$ is empty. The idea is to explore the synchronous product and to try constructing two generic substitutions θ_G and σ_G by guessing at each *epconf* $pc = ((q_q, \vec{I}), (q_p, \vec{J}))$ the appropriate mappings in order to preserve a homomorphism from $T(A_Q, \theta_G)$ into $T(A_P, \sigma_G)$. More precisely, it aims to construct all mappings from the outgoing transitions of $(q_q, \vec{I})$ into outgoing transitions of $(q_p, \vec{J})$ (Algorithm 2 Line 3). Let $\mathcal{M}_{q_q \mapsto q_p}$ be such a set of mappings. Each mappings is a set of pairs of the form $(((q_q, \vec{I}), t_q, (q'_q, \vec{I'})), ((q_p, \vec{J}), t_p, (q'_p, \vec{J'})))$. The algorithm then guesses a mapping $M \in \mathcal{M}_{q_q \mapsto q_p}$ (Algorithm 2 Line 7) and constructs a set of constraints $\mathbb{C}_M$ made of constraints derived from the mapping M: for each pair $(((q_q, \vec{I}), t_q, (q'_q, \vec{I'})), ((q_p, \vec{J}), t_p, (q'_p, \vec{J'})))$ in M, a new constraint $(t_q = t_p)$ is added to $\mathbb{C}_M$. The global set of constraints $\mathbb{C}$ is then augmented with the new set $\mathbb{C}_M$ (Algorithm 2 Line 8). If the global set of constraints is not satisfiable (i.e., the associated quantified constraint formula QF($\mathbb{C}$) is not satisfiable), the algorithm returns $false$ (Algorithm 2 Line 9).

If the global set of constraints $\mathbb{C}$ is satisfiable the algorithm creates a new *epconf* $pc'_i = ((q'_q, \vec{I'}), (q'_p, \vec{J'}))$ for each element $m_i = (((q_q, \vec{I}), t_q, (q'_q, \vec{I'})), ((q_p, \vec{J}), t_p, (q'_p, \vec{J'})))$ of M (Algorithm 2 Line 14). The algorithm then recursively calls Check_Simu for each generated pc'_i. Thus the processing of the mapping M succeeds if all such calls succeed (Algorithm 2 Line 16).

The algorithm stops and returns true if a leaf of A_Q is reached or if a cover condition is fulfilled (Algorithm 2 Line 1).

In order to show correctness of the algorithm we need to define the execution tree of *Check_Simu*. The next section will explain step by step how it is obtained before giving its formal definition.

Algorithm 2 *Check_Simu*

Input : A_Q, A_P ; Epconf : pc ; Epconf's historic : Hist
Output : True if $A_Q \ll_\forall A_P$, False otherwise

if q is not a leaf and $Check\_Cover(A_Q, A_P, pc, Hist) == false$ **then**
  Hist $\leftarrow$ Hist $\cup$ {pc}
  Compute the mappings $\mathcal{M}_{q \mapsto p}$
  if $\mathcal{M}_{q \mapsto p} = \emptyset$ **then**
    return False
  else
    Guess a mapping M from $\mathcal{M}_{q \mapsto p}$
    $\mathbb{C} \leftarrow \mathbb{C} \cup \mathbb{C}_M$
    if $\mathbb{C}$ is not satisfiable **then**
      return False
    else
      $n \leftarrow |M|$
      for $m_i = (((q, \vec{I}), t_q, (q', \vec{I'})), ((p, \vec{J}), t_p, (p', \vec{J'}))) \in M$ **do**
        Compute $pc'_i = ((q', \vec{I'}), (p', \vec{J'}))$
      end for
      return $\bigwedge_{i \in [1,n]} Check\_Simu(A_Q, A_P, pc'_i, Hist)$
    end if
  end if
end if
return $True$

Algorithm 3 Check_Cover

Input : A_Q, A_P ; Epconf : pc ; Epconf's historic : Hist
Output : Boolean

if $\exists$ pc' $\in$ Hist such that pc' $\lhd_{\mathbb{C}}$ pc **then**
  return True
else
  return False
end if

5.3 Product Execution Tree of Check_Simu

The algorithm *Check_Simu* starts with the initial $epconf_0 = ((Q, \vec{0}), (P_1, \vec{0}))$. Possible mappings are generated and stored in the set $\mathcal{M}_{Q \mapsto P_1}$ (Algorithm 2 Line 3). One mapping is selected non-determinstically and the satisfiability of the corresponding constraints is verified (Algorithm 2 Line 9).

Example 25. *In Figure 5.2, the algorithm starts at the initial product configuration* $epconf_0 = ((Q, (0,0,0)), (P_1, (0)))$ *with an empty set of constraints. There are three outgoing transitions from the configuration* $(Q, (0,0,0))$*:*

$$\begin{array}{rcl} t_1 & = & ((Q, (0,0,0)), (x, 0), (N_{A_1}, (0,0,0))) \\ t_2 & = & ((Q, (0,0,0)), (y, 0), (N_{A_2}, (0,0,0))) \\ t_3 & = & ((Q, (0,0,0)), (z, 0), (Q, (0, , 1, 1))) \end{array}$$

There are two outgoing transitions from $(P_1, (0))$*:*

$$\begin{array}{rcl} t'_1 & = & ((P_1, (0)), R, (P_2, (1))) \\ t'_2 & = & ((P_1, (0)), R, (P_3, (0))) \end{array}$$

Consequently, the set of mappings $\mathcal{M}_{Q \mapsto P_1}$ *includes all the possible mappings from* $\{t_1, t_2, t_3\}$ *into* $\{t'_1, t'_2\}$*. Among them, the figure illustrates two mappings,* $M_1^\varepsilon = \{(t_1, t'_1), (t_2, t'_1)(t_3, t'_2)\}$ *and* $M_2^\epsilon = \{(t_1, t'_2), (t_2, t'_2), (t_3, t'_1)\}$*. In the figure, choices made by the algorithm are symbolized by the* $\vee$ *nodes. The left side corresponds to* M_1^ε *and the right side to* M_2^ϵ*. In both cases, the set of constraints generated is the same, i.e.* $\{(x,0) = R, (y,0) = R, (z,0) = R\}$*. The corresponding quantified constraint formula,* $\exists (x,0), (y,0), (z,0), (((x,0) = R) \wedge ((y,0) = R) \wedge (z, 0 = R))$ *is satisfiable.*

Each choice leads to consider a different product execution tree. Nonetheless, the algorithm then constructs children pc'_i issued of the chosen mapping. (Algorithm 2 Line 14). Check_Simu then makes a recursive call using the resulting children (Algorithm 2 Line 16). There are three possibilities for a child :

- It concerns a leaf or satisfies cover criteria.
- It fails to produce a satisfiable mapping.
- It chooses non-deterministically a mapping.

Example 26. *In Figure 5.2, each mapping leads to its specific execution trees.*
For the mapping M_1^ε*, the reached children* 0*,* 1 *and* 2 *are respectively labeled by* $((Q, (0,1,1)), (P_3, (0)))$*,* $((N_{A_2}, (0,0,0)), (P_2, (1)))$ *and* $((N_{A_1}, (0,0,0)), (P_2, (1)))$*. For those three nodes there are no mappings possible since variables can not be substituted by element of the set* N_A*. The set of mappings is then empty therefore the algorithm returns* false *(symbolized by the cross in the figure).*
We will now look toward M_2^ε *which produces a tree such that*

$$\lambda(0) = ((Q, (0,1,1)), (P_2, (1)))$$

$$\lambda(1) = ((N_{A_2}, (0,0,0)), (P_3, (0)))$$

$$\lambda(2) = ((N_{A_1}, (0,0,0)), (P_3, (0)))$$

The involved states are not leaves and cover criteria is not satisfied. The the set of mappings which is not empty is computed. Thus the algorithm continues its execution. Note that for the nodes 1 *and* 2*, there is only one successful mapping so we will not discuss this part. Regarding, the node* 0*, Figure 5.2 depicts two possible mappings. There are three outgoing transitions from the configuration* $(Q, (0,1,1))$*:*

$$\begin{array}{rcl} t_1 & = & ((Q,(0,1,1)),(x,0),(N_{A_1},(0,1,1))) \\ t_2 & = & ((Q,(0,1,1)),(y,1),(N_{A_2},(0,1,1))) \\ t_3 & = & ((Q,(0,1,1)),(z,1),(Q,(0,2,2))) \end{array}$$

There are three outgoing transitions from $(P_2, (1))$*:*

$$\begin{array}{rcl} t'_1 & = & ((P_2,(1)),S,(P_1,(1))) \\ t'_2 & = & ((P_2,(1)),R,(P_3,(1))) \\ t'_3 & = & ((P_2,(1)),(w,1),(P_3,(1))) \end{array}$$

Consequently, the set of mappings $\mathcal{M}_{Q \mapsto P_2}$ *includes all the possible mappings from* $\{t_1, t_2, t_3\}$ *into* $\{t'_1, t'_2, t'_3\}$*. On the left side, we depict the mapping* $M_1^0 = \{(t_1, t'_3), (t_2, t'_3), (t_3, t'_1)\}$*. However, the quantified constraint formula,* $\forall(w,1), \exists(x,0),(y,0),(y,1),(z,0),(z,1)(((x,0) = R) \wedge ((y,0) = R) \wedge (z,0 = R) \wedge ((y,1) = (w,1) \wedge ((x,0) = (w,1)))$ *is not satisfiable. Indeed, it contains* $(((x,0) = R) \wedge ((x,0) = (w,1)))$ *which infers* $\forall(w,1), ((w,1) = R)$*.*
On the other hand, the mapping $M_2^0 = \{(t_1, t'_2), (t_2, t'_3), (t_3, t'_1)\}$ *is successful and leads to new children which will makes a recursive call until the cover criteria is verified, a leaf is reached or it fails. In the figure, this criteria is verified for* 0 *and* 000 *since we have* $\lambda(0) \lhd_{\mathbb{C}} \lambda(000)$ *which is reached after one more step.*

The notion of product execution tree of *Check_Simu* will then be formally defined with regard to a mapping that generates a set of constraints : $\mathbb{C}_M$. This mapping is a union of mappings corresponding to the recursive choices the algorithm makes.

Definition 23 (Product execution tree of *Check_Simu* w.r.t to M)**.** *Let* $A_Q = (\mathcal{Q}_Q, \mathcal{L}_Q, \mathcal{V}ar_Q, q_0, \delta_Q, q_f, \kappa_Q)$ *and* $A_P = (\mathcal{Q}_P, \mathcal{L}_P, \mathcal{V}ar_P, p_0, \delta_P, p_f, \kappa_P)$ *be two description automata. Let* $\vec{G^Q} \in \mathbb{N}^{|\mathcal{V}ar_Q|}$ *and* $\vec{G^P} \in \mathbb{N}^{|\mathcal{V}ar_P|}$ *be counters. Let* S *be the set of all possible epconf and let* $\mathcal{L}$ *be* $\mathcal{L}_P \cup \mathcal{V}ar_P \times \mathbb{N}$*. A run of Check_Simu, denoted* $Exec_{A_Q,A_P}$ *and called a product execution tree, is a* $\langle S, \mathcal{L}\rangle$*-labeled tree* (τ, λ, δ) *constructed as follows:*

- $\lambda(\epsilon) = ((q_0, \vec{0}), (p_0, \vec{0}))$*, a root of the tree, and let* $\vec{G^Q} = \vec{G^P} = \vec{0}$*, a global counter initialized to a vector of zero.*

- *Let* $n \in \tau$ *be a node such that* $\lambda(n) = ((q_q, \vec{I}), (q_p, \vec{J}))$*. For each* m_i *in* $(((q_q, \vec{I}), t_q, (q'_q, \vec{I'})), ((q_p, \vec{J}), t_p, (q'_p, \vec{J'}))) \in M$*,a new child* ni *of* n *is generated according to the following sequence:*

  (i) $\delta(n, ni) = t_p$

  (ii) $\forall y \in \mathcal{V}ar_Q$*, if* $q'_q \in \kappa(y)$ *then* $G_y^Q := G_y^Q + 1$*, and*

(iii) Let $\vec{I'} = \begin{cases} I'_y = G^Q_y, & \forall y \in \mathcal{V}ar_Q,\ s.t.\ q'_q \in \kappa(y) \\ I'_y = I_y & otherwise \end{cases}$

(iv) $\forall y \in \mathcal{V}ar_P$, *if* $q'_p \in \kappa(y)$ *then* $G^P_y := G^P_y + 1$, *and*

(v) Let $\vec{J'} = \begin{cases} J'_y = G^P_y, & \forall y \in \mathcal{V}ar_P,\ s.t.\ q'_p \in \kappa(y) \\ J'_y = J_y & otherwise \end{cases}$

(vi) $\lambda(ni) = ((q'_q, \vec{I'}), (q'_p, \vec{J'}))$

The tree of the Figure 5.2 shows execution trees of $A_Q \ll_\forall A_P$. Note that transitions labeled with $(x, i) = t_p$, instead of t_p as stated in the definition, are here to illustrate bounding of variable during the execution. Everything required to prove correctness of the algorithm is now defined. The next section will then detail correctness of *Check_Simu*.

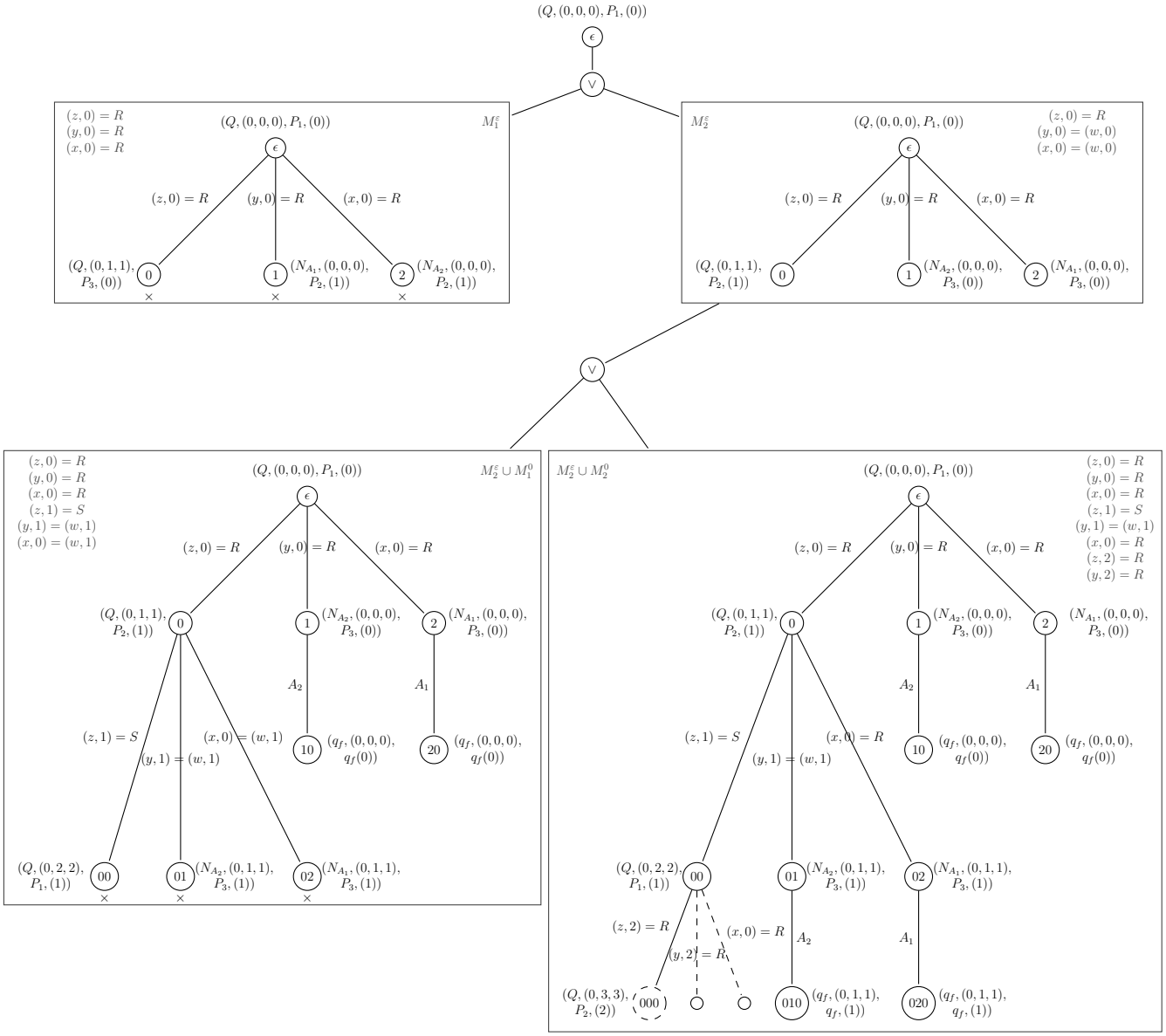


Figure 5.2: Fragment Execution trees of *Check_Simu*

5.4 Correctness of Check_Simu

5.4.1 Termination of Check_Simu

A run of the algorithm with the initial $epconf$ leads to an exploration of its associated execution tree. An execution tree may be infinite, however we aim here to prove that only a finite part is explored leading to the algorithm's halt. The algorithm stops exploring a branch if one of the following occurs:

(i) A leaf has been reached.

(ii) The chosen mapping $\mathcal{M}_{q\mapsto p}$ of a visited $epconf$ failed.

(iii) An $epconf$ is covered by a previous one.

The two first cases will not be discussed since by definition they terminate. It remains to discuss potential infinite branches. In our context, it means that there exists an infinite sequence of $epconfs$. The following property ensures that using $epcover$ prevents from an infinite branch.

Property 4. *Let $Exec_{A_Q,A_P}$ be a product execution tree of two description automata A_Q and A_P and let $\mathbb{C}$ be its associated set of global constraints. For any infinite sequence $pc_1.pc_2.pc_3...pc_n$ of $epconfs$ of $Exec_{A_Q,A_P}$, there exists $i < j$ such that $pc_i \lhd_{\mathbb{C}} pc_j$.*

Proof. Let $A_Q = (\mathcal{Q}_Q, \mathcal{L}_Q, \mathcal{V}ar_Q, q_0, q_f, \delta_Q, \kappa_Q)$ and $A_P = (\mathcal{Q}_P, \mathcal{L}_P, \mathcal{V}ar_P, p_0, p_f, \delta_P, \kappa_P)$. Let consider an infinite sequence $pc_1.pc_2.pc_3...pc_n$ of $epconfs$ of $Exec_{A_Q,A_P}$. $\mathcal{Q}_P$ and $\mathcal{Q}_Q$ are finite sets. A a direct consequence, there exists $q_q \in \mathcal{Q}_Q$ and $q_p \in \mathcal{Q}_P$ such that there are an infinite number of pc_k of the form $(q_k = (q_q, I_k), p_k = (q_p, J_k))$ within this sequence.

If we take pc_k as defined above then there exists $k < k'$ such that $pc_{k'} = (q_{k'} = (q_q, I'_{k'}), q_{p'} = (q_p, J'_{k'}))$ and the following can be observed :

1. $\mathbb{C}_{|q_k} \equiv \mathbb{C}_{|q_{k'}}$

2. $\mathbb{C}_{|p_k} \equiv \mathbb{C}_{|p_{k'}}$

Indeed, an instance of a variable of A_Q has a total of $(\mathcal{L}_P + \mathcal{V}ar_P + 1)$ possible assignments which is a finite number. It is explained by the fact that if an existential variable is mapped to two different universal variables, the corresponding quantified formula is not satisfiable which limits $\mathcal{V}ar_P \times \mathbb{N}$ to $\mathcal{V}ar_P$. Since $epcover$ compares each relevant variable bounds, there are only a finite number of different possible mappings.

Consequently, pc_k and pc'_k are such that $pc_k \lhd_{\mathbb{C}} pc'_k$ and $k < k'$ which concludes the proof. □

As a direct consequence $Check\_Simu$ has the following property.

Property 5. *The algorithm Check_Simu terminates.*

5.4.2 Soundness of Check_Simu

A run of the algorithm with the initial *epconf* leads to an exploration of its associated execution tree. We will consider here, $Exec_{A_Q,A_P} = (\tau, \lambda, \delta)$ corresponding to a successful run of Check_Simu.

From $Exec_{A_Q,A_P}$, we are able to extract a partial configuration tree $PT(A_Q, \theta_G)$ of A_Q and a partial configuration tree $PT(A_P, \sigma_G)$. The algorithm ensures the following properties :

(i) There is a homomorphism from $PT(A_Q, \theta_G)$ into the partial-tree of $PT(A_P, \sigma_G)$ of A_P.

(ii) $PT(A_Q, \theta_G)$ and $PT(A_P, \sigma_G)$ can be extended to configuration trees $T(A_Q, \theta_G)$ and $T(A_P, \sigma_G)$ such that there is a homomorphism between $T(A_Q, \theta_G)$ into $T(A_P, \sigma_G)$ (Lemma 8).

Based on $Exec_{A_Q,A_P} = (\tau, \lambda, \delta)$, we define the partial trees $PT(A_Q, \sigma_G) = (\tau_Q, \lambda_Q, \delta_Q)$ and $PT(A_P, \theta_G) = (\tau_P, \lambda_P, \delta_P)$ as follows :

- $\forall n \in \tau$, $\lambda(n) = ((q_q, \vec{I}), (q_p, \vec{J}))$ then $\lambda_Q(n) = (q_q, \vec{I})$ and $\lambda_P(n) = (q_p, \vec{J})$
- $\delta_Q(n, ni) = \delta_P(n, ni) = \theta_G(t_q) = \sigma_G(t_p) = t_p$

Configuration trees are associated to a function, here θ_G is done according to the set of constraints of the execution tree $\mathbb{C}_M$ while σ_G is identity. By construction, it is clear that there is a homomorphism Z_{id} between those trees such that $Z_{id}(i) = i$. The tree labeled with configuration of A_Q is a partial tree of a configuration tree and the one labeled with configuration of A_P is a partial tree $T(A_P, \sigma_G)$.

Then $\forall \sigma$, $\theta = \sigma \circ \theta_G$ is a possible substitution for A_Q such that the homomorphism is preserved.

These trees are partial configuration trees for two reasons :

(i) A branch has been cut by *epcover* preventing from the infinite branch of $T(A_Q, \sigma_G)$ or $T(A_P, \theta_G)$.

(ii) Only outgoing transitions of A_P required to mimic A_Q are explored. Some outgoing transitions of A_P may not be explored by the algorithm.

Figure 5.3 illustrates the resulting partial configuration trees of the running example. The following lemma states that *epcover* perserve the homomorphism for complete trees.

Lemma 8. *Let* $A_Q = (\mathcal{Q}_Q, \mathcal{L}_Q, \mathcal{V}ar_Q, q_0, q_f, \delta_Q, \kappa_Q)$ *and* $A_P = (\mathcal{Q}_P, \mathcal{L}_P, \mathcal{V}ar_P, p_0, p_f, \delta_P, \kappa_P)$ *be two description automata. Let* $\mathbb{C}$ *be a set of constraints. For any epconf* $pc_1 = ((q_1, \vec{I}), (p_1, \vec{J}))$ *and* $pc_2 = ((q_2, \vec{I'}), (p_2, \vec{J'}))$ *such that* $pc_1 \lhd_{\mathbb{C}} pc_2$ *w.r.t* $\mathbb{C}$. *Then* $(q_2, \vec{I'}) \ll_\forall (q_1, \vec{I})$ *and* $(p_1, \vec{J}) \ll_\forall (p_2, \vec{J'})$

Proof. By definition of *epcover* we have :

- $p_1 = p_2$ and $\mathbb{C}_{|p_1} \equiv \mathbb{C}_{|p_2}$
- $q_1 = q_2$ and $\mathbb{C}_{|q_1} \equiv \mathbb{C}_{|q_2}$

Then for any outgoing edge of $(q_2, \vec{I'})$ which is of the form $((q_2, \vec{I'}), x, (q', \vec{K'}))$ there is an outgoing edge $((q_1, \vec{I}), x, (q', \vec{K}))$.

The different cases are :

- x is a constant which is trivial.
- For all unconstrained (x, I'_x) then (x, I_x) is unconstrained then for any $\sigma'(x, I'_x) = t$, we can construct σ such that $\sigma(x, I_x) = t = \sigma(x, I_x)$.
- $\forall((x, I'_x) = t) \in \mathbb{C}_{|q_2}$, then $(x, I_x) = t \in \mathbb{C}_{|q_1}$

The reached configuration $(q', \vec{K})$ and $(q', \vec{K'})$ are such that $(q', \vec{K}) \lhd_{\mathbb{C}} (q', \vec{K'})$. The same reasoning can then be recursively applied. Similarly, $(p_1, \vec{J}) \ll_\forall (p_2, \vec{J'})$ is obtained by exchanging roles of p_1 and p_2 compared to q_1 and q_2.

□

While looking for the generic substitutions θ_G and σ_G, the algorithm checks the *pconf* $pc_1 = ((q_1, \vec{I}), (p_1, \vec{J}))$ in order to have $(q_1, \vec{I}) \ll_\forall (p_1, \vec{J})$. It stops when a leaf is reached which is a trivial or when it finds a cover $pc_2 = ((q_2, \vec{I'}), (p_2, \vec{J'}))$ for this configuration (i.e. $pc_1 \lhd_{\mathbb{C}} pc_2$). In this case, $(q_1, \vec{I}) \ll_\forall (p_1, \vec{J})$ has been verified during the run. Therefore, thanks to Lemma 8 and transitivity of universal simulation, we have $(q_2, \vec{I'}) \ll_\forall (p_2, \vec{J'})$.

Property 6. *The algorithm Check_Simu is sound.*

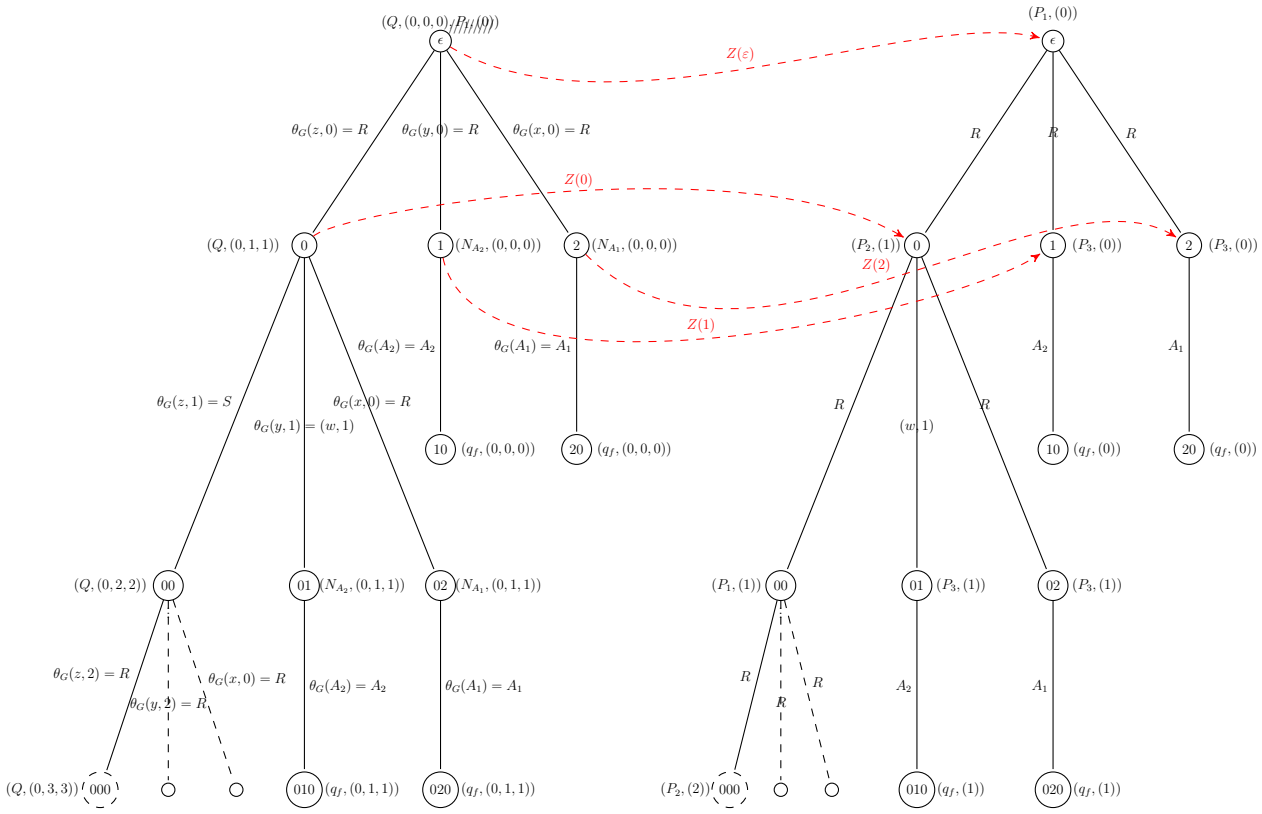


Figure 5.3: Partial Configuration trees extracted from $Exec_{A_Q, A_{P_1}}$

5.4.3 Completeness of Check_Simu

The algorithm makes an exhaustive exploration thanks to computation of all the mappings (Algorithm 2 Line 3). Proof of completeness is then similar to the proof of Check_Match using a given solution one can guide the algorithm. If the algorithm returns false, it demonstrated that the solution was not correct since the computation of mappings is exhaustive. Otherwise soundness ensures the algorithm's answer is good.

Property 7. *The algorithm Check_Simu is complete.*

5.5 Extension to Weak-Subsumption

Pattern containment and weak-subsumption are two problems involving variable on both sides. Fortunately, decidability of weak-subsumption can be achieved similarly with a minor change to handle the specific features of the problem. Indeed, the simulation problem we aim to solve is $A_Q \ll_\exists A_P$ which only requires the existence of a substitution for both automata.

The main difference lies in the definition of the quantified constraint formula required for satisfiability. Intuitively, the quantified constraint formula applied the quantifier "$\forall$" for any variable from A_P and the quantifier "$\exists$" for variables from A_Q accordingly to the simulation definition. However, existential simulation does not carry any difference between variables since both sides are associated to the quantifier "$\exists$". Consequently, the formula generated is a classic constraint formula defined as follows :

$$\bigwedge_{\psi \in \mathbb{C}} \psi \text{ with } \mathbb{C} \text{ a set of constraints.}$$

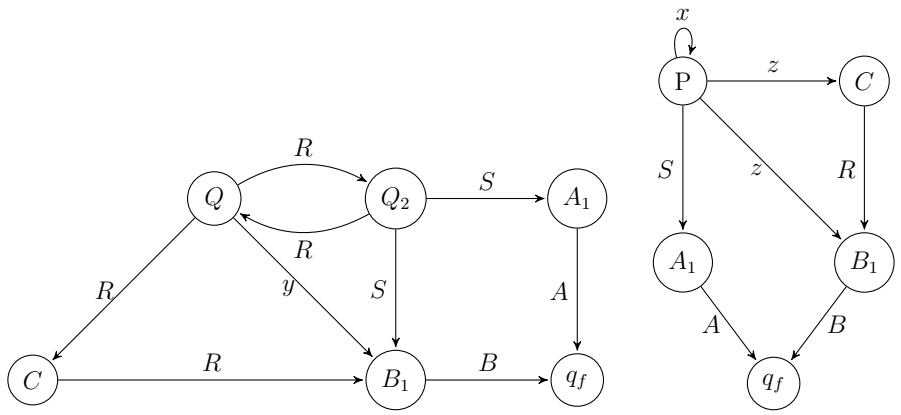
Example 27. *Consider the following $\mathcal{EL}_{\mathcal{RV}}$-terminology where all the variables that occur in $\mathcal{EL}_{\mathcal{RV}}$-descriptions are refreshing variables.*

$$\begin{aligned}
A_1 &\equiv A \\
B_1 &\equiv B \\
C &\equiv \exists R.B_1 \\
P &\equiv \exists x.P \sqcap \exists z.C \sqcap \exists z.B_1 \sqcap \exists S.A_1 \\
Q &\equiv \exists R.Q_2 \sqcap \exists R.C \sqcap \exists y.B_1 \\
Q_2 &\equiv \exists R.Q \sqcap \exists S.B_1 \sqcap \exists S.A_1
\end{aligned}$$

Figures 5.4 depict respectively the $\mathcal{EL}_{\mathcal{RV}}$-description automata of the $\mathcal{EL}_{\mathcal{RV}}$-patterns P and Q. The refreshing state of both x and z is the state P (i.e., $\kappa(x) = \kappa(z) = \{P\}$) while the refreshing state of y is the state Q (i.e., $\kappa(y) = \{Q\}$).

An execution of the algorithm is presented in Figure 5.5. The algorithm will return true.

The proof of termination does not change since the number of configurations remains the same. However, in case of weak-subsumption, the substitution domain can be reduced to constants of A_P and A_Q since we only look for the existence of a solution.



(a) A_Q (b) A_P

Figure 5.4: Automata of Example 27

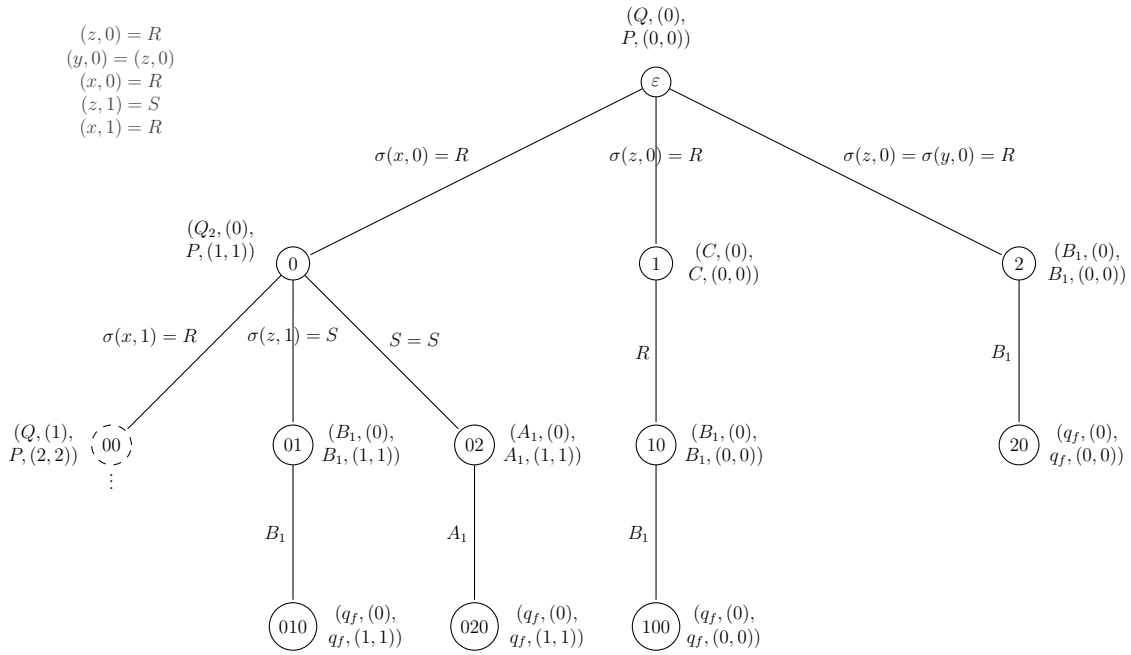


Figure 5.5: Potential Execution Tree to solve $A_Q \ll_\exists A_P$

5.6 Complexity of Reasoning Tasks in $\mathcal{EL}_{\mathcal{RV}}$

The algorithm *Check_Simu* is proven to be correct as is Check_Match. We will now demonstrate that these algorithms are optimal. The lower bound is achieved by a reduction from an alternating turing machine's problem to matching problem. The upper bound will be obtained by estimating the size of the space of search in the worst case. The next section give the details to compute those bounds resulting in a tight upper and lower bounds.

5.6.1 Lower Bound : ATM Halting and Matching

The EXPTIME-hardness of checking universal simulation between $\mathcal{EL}_{\mathcal{RV}}$-description automata is obtained by a reduction from the existence of infinite execution of an Alternating Turing Machine M working on a space polynomially bounded by the size of the input. This later problem is known to be EXPTIME-COMPLETE [27].

Definition 24. ***Alternating Turing machine[27]***
An alternating Turing machine M is a tuple $(\mathbb{Q}, q_0, \Gamma, \delta, mode)$ where :

- $\mathbb{Q}$ *is the set of states*
- q_0 *is the initial state*
- Γ *is the set of tape symbols*
- $mode : \mathbb{Q} \mapsto \{\forall, \exists, accept, reject\}$. *is the labelling function of control state*
- $\delta : \mathbb{Q} \times \Gamma \mapsto \mathcal{P}(\mathbb{Q} \times \Gamma \times \{L, R\})$

Figure 5.6 represents an example of an alternating turing machine composed of three states. Among them, there are one universal state (q^0) and two existential states (q^1 and q^2). This machine will be later used to illustrate the reduction of universal simulation.

A configuration C of M is of the form $y_1, ..., qy_j, ..., y_n$ where $q \in \mathbb{Q}$ a state of M, and the head points actually on the j^{th} letter of the tape (i.e., y_i are the letters of the word w on the tape). A transition $qa \mapsto bRq'$ is applicable from a configuration C if the letter pointed by the head is equal to a ($y_j = a$), then the successor C' of C is equal to $y'_1, ...y'_j, q'y'_{j+1}, ..., y'_n$ s.t $y_k = y'_k$ for $k \in [1, n]$ and $k \neq j$ and $y'_j = b$. It can be rewritten over configuration as $C \xrightarrow{qa/bRq'} C'$ or even $(y_1, ..., qy_j, ..., y_n) \xrightarrow{qa/bRq'} (y'_1, ...y'_j, q'y'_{j+1}, ..., y'_n)$. The machine M then starts on $C_0 = q_0 y_1, ..., y_n$, where $y_i = w_i$, the i^{th} letter of the input word w. The definition of acceptance of an alternating Turing machine is recursive :

- If the configuration C is in an accepting control state q, then C is accepting.
- If the configuration C is in a rejecting control state q, then C is rejecting.
- If the configuration C is in a universal control state q, then C is accepting if all the configurations reachable from C in one step are accepting and rejecting if some configurations reachable from C in one step are rejecting.
- If the configuration C is in an existential control state q, then C is accepting if some configurations reachable in one step are accepting and rejecting when all configurations reachable in one step are rejecting (the case of classical non-deterministic Turing machine correspond to an alternating machine where all states are existential).

M is said to accept an input word w if the initial configuration of M is accepting, and to reject w if the initial configuration is rejecting. A configuration reachable in one step from configuration C is called a successor of C and the set of successors of C is denoted successors(C).

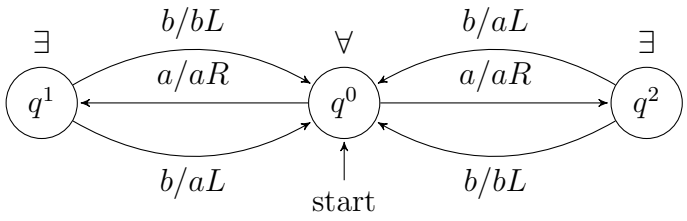


Figure 5.6: Example of Alternating Turing Machine

We consider the problem of the existence of an infinite execution of an Alternating Turing machine M on an input word $w = w_1, ..., w_n$. That is given a word w as input then whatever the transitions of universal states of M the machine must continue the execution. For existential states of M, the,machine must have at least a transition such that the machine continues its execution. Thus, rejecting states are states without outgoing transitions.

The machine works on a space bounded by the size n of the input word w. Hence, if the head points on x_1 the machine is not allowed to move to the left (i.e. execute a transition labelled with L), and if the head points on x_n the machine is not allowed to move to the right (i.e. execute a transition labelled with R).

Given an Alternating Turing Machine M working on a space polynomially bounded by the size of the input word w. We construct two description automata denoted as $A_{control}$ and A_{test} such that M has an infinite execution on the input w if and only if $A_{test} \ll_{\forall} A_{control}$. Note that in the reduction $A_{control}$ will be a ground automata hence considering a matching problem which is a special case of pattern containment. Therefore, we will only aim to prove the **existence** of a substitution for A_{test}.

We consider a function nt: $\mathcal{Q} \mapsto \mathbb{N}$ such that it gives for each state q of M the number of outgoing transitions. For practical purpose, each state q of M will consider a specific order over transitions t_j with $j \in$ [1,nt(q)]. This order will be used later on to build $A_{control}$ and A_{test}.

Construction of $A_{control}$ decomposes transitions of the original machine M. To do so, we will encode each state q of M as a state l_q in $A_{control}$. Moreover, we denote a set of states $\{q_{t_j,i}\}$ such that the j^{th} transition t_j of q can be decomposed. A transition of M first reads the letter in x pointed by the head then the transition replaces value stored in the cell x and moves the head accordingly. Each steps will then corresponds to a specific transition in $A_{control}$. $A_{control}$ focuses on capturing the structure and the behaviour defined in M. In other words it emphasizes on the input word w and transitions of M. Each transition in M is encoded into a sequence of transitions in $A_{control}$. Note that the structure induced differs regarding the mode of the state (i.e. universal or existential).

Definition of $A_{control}$:

Let $A_{control} = (\mathcal{L}_{control}, \mathcal{V}ar_{control}, \mathcal{Q}_{control}, q_{0,control}, q_{f,control}, \delta_{control}, \kappa_{control})$ defined as follows :

- The set of letters $\mathcal{L}_{control}$ is composed of :

  - $\{t_j\}$ with $j \in [1, m]$, the symbols representing the transitions of M and $m = max(\{\text{nt}(q)|q \in \mathcal{Q}\})$ the maximal number of transitions among states of M
  - $\{w_i\}$ the letters of the input w with $i \in [1, |w| = n]$
  - $\Gamma \cup \{L, R\}$, the symbols used in M
  - *reject* a specific symbol only used by the state *univ*

- $\mathcal{V}ar_{control} = \emptyset$
- The set of states $\mathcal{Q}_{control}$ is composed of :
  - $\{l_q|q \in \mathcal{Q}\}$ with $\mathcal{Q}$ the set of states of M
  - $\{q_{t_j,i}\}$ a set of additional states such that $i \in [0, 3]$ and $j \in [1, m]$ with m as defined above
  - $\{q_{init,i}\}$ a set of additional states used to mimic the input w with $i \in [0, n[$
  - *univ*, a state that can loop on itself with any letter of $\mathcal{L}_{control}$
- $q_{0,control} = q_{init,0}$
- $q_{f,control} = \emptyset$
- The set of transitions $\delta_{control}$ is composed of :
  - For each letter of $w = w_1...w_n$: $\{q_{init,i}, w_i, q_{init,i+1}\}$ for $i \in [0, n[$ and $(q_{init,n-1}, w_n, l_{q_0})$
  - For each transition t_j of q such that $q \xrightarrow{a/bd} q'$ where q is an existential state of M , $j \in$ [1,nt(q)] :
    - $(l_q, t_j, q_{t_j,0})$,
    - $(q_{t_j,0}, t_j, q_{t_j,1})$
    - $(q_{t_j,1}, a, q_{t_j,2})$
    - $(q_{t_j,2}, b, q_{t_j,3})$
    - $(q_{t_j,3}, d, l_{q'})$
    - $\{(q_{t_j,0}, t_k, univ)\}$ for $j \neq k$ and $k \in$ [0,nt(q)]
  - For each transition t_j of q such that $q \xrightarrow{a/bd} q'$ where q is a universal state of M, $j \in$ [1,nt(q)]
    - $(l_q, t_j, q_{t_j,1})$
    - $(q_{t_j,1}, a, q_{t_j,2})$
    - $(q_{t_j,2}, b, q_{t_j,3})$
    - $(q_{t_j,3}, d, l_{q'})$
  - $(univ, l, univ)$ for $l \in \mathcal{L}_{control}$
- $\kappa_{control} = \emptyset$

$A_{control}$ alone only encodes the behaviour of the alternating turing machine. On the opposite, A_{test} emphasizes on the tape and the head position. It considers a set of variables where each element x_i corresponds to the i^{th} cell of the tape. For each state of q in M, we then denote q_{x_i} the state encoding q in M with the head on the i^{th} cell.

In A_{test}, the set $\{q_{x,t_j,i}\}$ is used to decompose transitions of M with q the state considered, x representing the variable pointed by the head and t_j the transition. By making a wise use of L and R, we can link the transition toward the state simulating the head position in a deterministic way. As we did for $A_{control}$, we will use a set of additional transitions to capture the behaviour of universal states and existential states.

Definition of A_{test} :
Let $A_{test} = (\mathcal{L}_{test}, \mathcal{V}ar_{test}, \mathcal{Q}_{test}, q_{0,test}, q_{f,test}, \delta_{test}, \kappa_{test})$ defined as follows :

- The set of letters $\mathcal{L}_{test}$ is composed of :
  - $\{t_j\}$ with $j \in [1, m]$, the symbols representing the transitions of M and $m = max(\{\text{nt}(q) | q \in \mathcal{Q}\})$ the maximal number of transitions for one state in M.
  - $\{w_i\}$ the symbols. corresponding to the letters of the input w with $i \in [1, |w| = n]$.
  - $\Gamma \cup \{L, R\}$, the symbols used in M
  - *reject* a letter to model failing states of M.
- The set of variable $\mathcal{V}ar_{test}$ is composed of :
  - $\{x_i\}$ with $i \in [1, |w| = n]$, refreshing variables corresponding to the n letters of the input word w.
  - *choice* a refreshing variable required for existential state of M
- The set of states $\mathcal{Q}_{test}$ is composed of :
  - $\{q_{x_i} | q \in \mathcal{Q}\}$ with $\mathcal{Q}$ the set of states of M and x_i representing the position of the head on the i^{th} cell in M
  - $\{q_{x,t_j,i}\}$ a set of additional states such that x target the outgoing variable used by the state, $i \in [0, 3]$ and $j \in [1, m]$ with m as defined above.
  - $\{q_{init,i}\}$ a set of additional states to initialize w with $i \in [0, n[$
- $q_{0,test} = q_{init,0}$
- $q_{f,test} = \emptyset$
- The set of transitions δ_{test} is composed of :
  - For each letter of $w = w_1...w_n$: $\{q_{init,i}, x_i, q_{init,i+1}\}$ for $i \in [0, n[$ and $(q_{init,n}, x_n, q_{x_1})$
  - For each transition t_j of q such that $q \xrightarrow{a/bd} q'$ where q is an existential state of M, $j \in [1,\text{nt}(q)]$ and for $i \in [1, n]$:
    - $(q_{x_i}, t_j, q_{x_i,t_j,0})$,

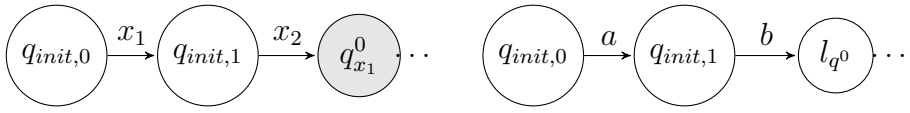
  - $(q_{x_i,t_j,0}, choice, q_{x_i,t_j,1})$,
  - $(q_{x_i,t_j,1}, x_i, q_{x_i,t_j,2})$,
  - $(q_{x_i,t_j,2}, x_i, q_{x_i,t_j,3})$,
  - if $d = R$ and $i \neq n$: $(q_{x_i,t_j,3}, R, q'_{x_{i+1}})\}$
  - if $d = L$ and $i = 1$: $(q_{x_i,t_j,3}, reject, q_{x_i,t_j,3})\}$
  - if $d = R$ and $i \neq n$: $(q_{x_i,t_j,3}, R, q'_{x_{i+1}})\}$
  - if $d = R$ and $i = n$: $(q_{x_i,t_j,3}, reject, q_{x_i,t_j,3})\}$

- For each transition t_j of q such that $q \xrightarrow{a/bd} q'$ where q is a universal state of M, $j \in [1,\mathrm{nt}(q)]$ and for $i \in [1,n]$:
  - $(q_{x_i}, t_j, q_{x_i,t_j,1})$,
  - $(q_{x_i,t_j,1}, x_i, q_{x_i,t_j,2})$,
  - $(q_{x_i,t_j,2}, x_i, q_{x_i,t_j,3})$,
  - if $d = L$ and $i \neq 1$: $(q_{x_i,t_j,3}, L, q'_{x_{i-1}})\}$
  - if $d = L$ and $i = 1$: $(q_{x_i,t_j,3}, reject, q_{x_i,t_j,3})\}$
  - if $d = R$ and $i \neq n$: $(q_{x_i,t_j,3}, R, q'_{x_{i+1}})\}$
  - if $d = R$ and $i = n$: $(q_{x_i,t_j,3}, reject, q_{x_i,t_j,3})\}$
- For each state q such that $\mathrm{nt}(q) = 0$: $(q_{x_i}, reject, q_{x_i})$ for $i \in [1,n]$

- $\kappa_{test}(x_i) = \{(q_{x_i,t_j,2})\}$ for $i \in [1,n]$ and $\kappa(choice) = \{q_{x_i}\}$ for $i \in [1,n]$ and $mode(q) = \exists$.

If the simulation fails because $A_{control}$ can not simulate A_{test}, it means that the move of M can not be done according to the current values of the tape. We will now present how behaviours of initialization, universal states and existential states are captured by universal simulation.

- **Initialization** : A sequence of states/transitions is added such that concatenating labels of the transitions forms w. This sequence ends reaching the state representing in $A_{control}$ the initial state of M. The same is done in A_{test} with the corresponding variables instead of letters of w. Note that this sequence leads to the state representing the initial state of M with the head on the first letter. The only possibility for an homomorphism is to associate to each variable x_i the letter w_i. Figure 5.7 illustrates of both $A_{control}$ and A_{test} initialization on the running example for the word $w = ab$.



(a) A_{test} (b) $A_{control}$

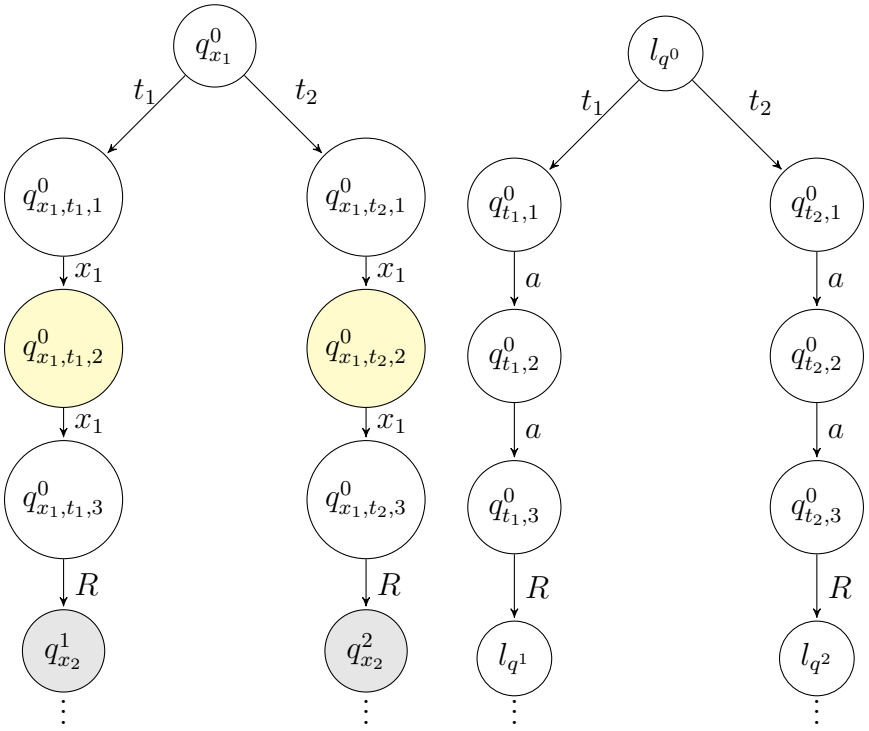
Figure 5.7: Example of Initialization for the Input Word ab

- **Universal states** : The constructions in both automata are identical up to variables. For example, let consider the same transition $(q^0, b/aL, q^1)$ knowing it is the second

transition and that the head is on the first letter. This transition is respectively encoded in A_{test} and $A_{control}$ as :

$$\{(q^0_{x_1}, t_2, q^0_{x_1,t_2,1}), (q^0_{x_1,t_2,1}, x_1, q^0_{x_1,t_2,2}), (q^0_{x_1,t_2,2}, x_1, q^0_{x_1,t_2,3}), (q^0_{x_1,t_2,1}, L, q^1_{x_2})\}$$
$$\{(l_{q^0}, t_2, q^0_{t_2,1}), (q^0_{t_2,1}, b, q^0_{t_2,2}), (q^0_{t_2,2}, a, q^0_{t_2,3}), (q^0_{t_2,3}, L, l_{q^1})\}$$

It is quite intuitive to see that the homomorphism will map $(q^0_{x_3,t_2,1}, x_3, q^0_{x_3,t_2,2})$ into $(q^0_{t_2,1}, b, q^0_{t_2,2})$. This is possible only if the value of the instance of x_3 is equal to b. Then, by definition, x_3 is freed in $q_{x_3,t_2,2}$. This lead to a brand new instance of x_3 which is immediately associated to a (otherwise simulation fails). Figure 5.8 illustrates universal structure on the running example for the state q^0.



(a) A_{test} (b) $A_{control}$

Figure 5.8: Example of Universal Construction Corresponding to q_0

- **Existential states** The construction focuses on choosing which transitions must be checked out. Since the decomposition of the transition is similar than for universal states once the choice is done, we will focus on the two first transitions. For example, let consider the transitions of q^1 with the head on the second letter. We have t_1 for $(q^1, b/aL, q^0)$ and t_2 for $(q^1, b/bL, q^0)$ then they will be respectively in A_{test} and $A_{control}$ encoded as :

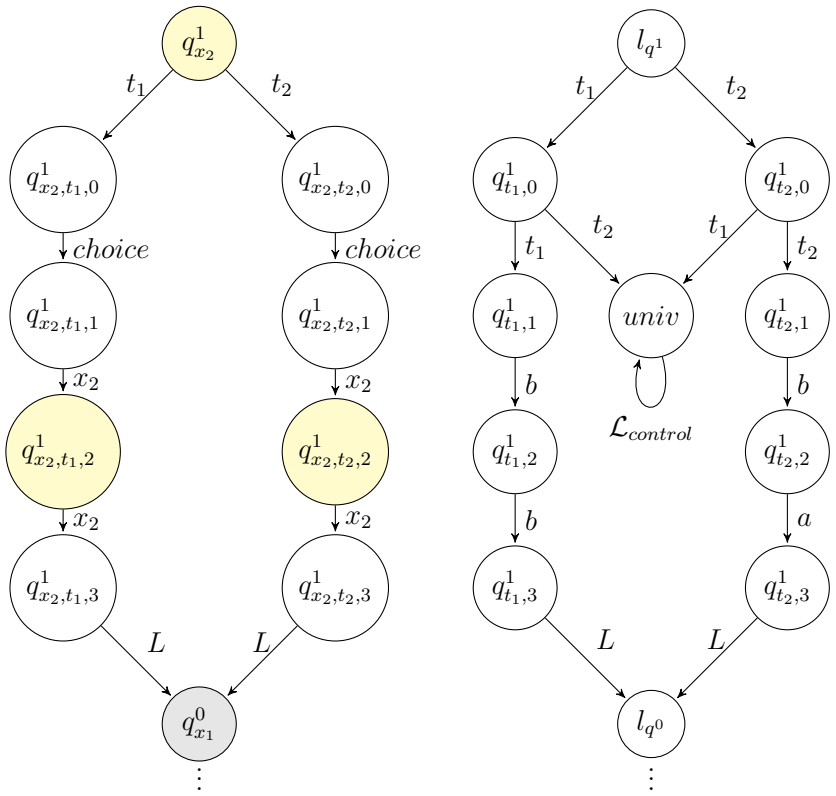
$$\{(l^1_q, t_2, q^1_{t_2,0}), (q^1_{t_2,0}, t_2, q^1_{t_2,1}), (q^1_{t_2,0}, t_1, univ)...\}$$

$$\{(q^1_{x_2}, t_2, q^1_{x_2,t_2,0}), (q^1_{x_2,t_2,0}, choice, q^1_{x_2,t_2,1})...\}$$

and

$$\{(l_q^1, t_2, q_{t_1,0}^1), (q_{t_1,0}^1, t_1, q_{t_1,1}^1), (q_{t_1,0}^1, t_2, univ)...\}$$
$$\{(q_{x_2}^1, t_2, q_{x_2,t_1,0}^1), (q_{x_2,t_1,0}^1, choice, q_{x_2,t_1,1}^1)...\}$$

Since *choice* is refreshed by $(q_{x_2}^1)$, any configuration involving this state will always deal with a brand new (free) instance. Any outgoing transitions will then synchronize *choice* value. It means that a choice can be made but it will be synchronized in both paths. Let assume $choice = t_1$, then if there is an homomorphism, it will map $q_{x_2,t_1,1}^1$ into $q_{t_1,1}^1$ and map $q_{x_2,t_2,1}^1$ into *univ*. Figure 5.9 illustrates universal structure on the running example for the state q^1



(a) A_{test} (b) $A_{control}$

Figure 5.9: Example of Existential Construction Corresponding to q_1

Given an alternating turing machine M an input word w, we construct $A_{control}$ and A_{test} as explained previously denoted as the description automata associated to M and w. The next lemma shows the connection between the existence of infinite execution of the machine M over the word w and the test of simulation between A_{test} and $A_{control}$.

Theorem 3. *Let M be an alternating Turing machine working in space bounded by the size of an input word w, and let $A_{control}$ and A_{test} the description automata associated to M and w. Then, M has an infinite computation on w if and only if $A_{test} \ll_\forall A_{control}$.*

Proof. The initializing phase will enforce each letter w_i on the corresponding variable x_i leading to the matching *pconf* $((q_{x_1}^0, \vec{0}), l_{q^0}, \{(x, i) = w_i \forall i \in [1, n]\})$.

If l_{q^0} is a universal state then there is only one possibility that maps the edge labeled t_j of A_{test} into the edge labeled t_j of $A_{control}$. If it returns false it means that there is no homomorphism for at least one edge which corresponds to M halting.

If l_{q^0} is an existential state then *choice* is free. Resulting in *choice* mapped into t_j with $j \in [1, nt(l_{q^0})]$. Then any transitions that does not correspond to choice will fulfils simulation requirements thanks to the universal state. The remaining transition will then be checked by processing the same transition in $A_{control}$. If none of *choice* possibilities works, then all transitions failed which corresponds to M halting.

Finally, states without outgoing edges in M automatically produces a fail since it has an outgoing edge in A_{test} but not in $A_{control}$ which also corresponds to M halting.

Consequently, if $A_{test} \ll_\forall A_{control}$ it means that M as an infinite run on the input word w. □

Constructing $A_{control}$ requires only polynomial transformation M. The number of states induced is :

$$|\mathcal{Q}| + 3 * \sum_{\{q \in \mathcal{Q} | mode(q) = \forall\}} nt(q) + 4 * \sum_{\{q \in \mathcal{Q} | mode(q) = \exists\}} nt(q) + 1 + n$$

The same can be done for transitions which gives :

$$(4 * \sum_{\{q \in \mathcal{Q} | mode(q) = \forall\}} nt(q) + \sum_{\{q \in \mathcal{Q} | mode(q) = \exists\}} nt(q)(4 + nt(q))) + n + |\mathcal{L}|$$

It is immediate to see that the construction of $A_{control}$ is polynomial in the size of M and w.

The same can be done for A_{test} which bears the same difference for universal and existential since the structure are similar. The number of states induced is :

$$n * (|\mathcal{Q}| + 3 * \sum_{\{q \in \mathcal{Q} | mode(q) = \forall\}} nt(q) + 4 * \sum_{\{q \in \mathcal{Q} | mode(q) = \exists\}} nt(q)) + n$$

Regarding the number of transition, we have :

$$n * (\sum_{\{q \in \mathcal{Q} | nt(q) = 0\}} 1 + 4 * \sum_{\{q \in \mathcal{Q} | mode(q) = \forall\}} nt(q) + 5 * \sum_{\{q \in \mathcal{Q} | mode(q) = \exists\}} nt(q)) + n$$

It is clear to see that the construction of A_{test} is also polynomial in the size of M and w.

Corollary 3. *Deciding if a matching problem in $\mathcal{EL}_{\mathcal{RV}}$ has a solution is* EXPTIME-HARD.

5.6.2 Upper Bound : Size of the Search Space

The EXPTIME upper bound lies in the number of different *pconf* explored. In other words, by the size of the space of search considered by the Algorithm *Check_Simu*. In order to demonstrate its exponentially, we define three factors :

- n : the number of states (i.e. the maximum in both automata)

- v : the number of variables
- D : the domain of valuation of variables.

The algorithm explores a space made of $pconfs$ which concerns two states forming a basis of $n * n$ $pconfs$. Two $pconfs$ sharing the same states are seen different by the algorithm up to the current value assignments. Luckily, we only need to consider a finite number of possibilities thanks to relevant variables. It amounts to D^v possibilities for variables of A_Q. there is no additional requirements for variables of A_P since valuation for these variables are a consequence of edge mappings from A_Q to A_P. We then have a total of $n^2 * D^V$ different $pconfs$. It can be rewritten in $e^{2\ln(n)+v\ln(D)}$ giving exponentially.

Lemma 9. *Deciding if a pattern containment problem in $\mathcal{EL}_{\mathcal{RV}}$ has a solution is in* EXPTIME.

This result coupled with Corollary 3 immediately implies the following theorem :

Theorem 4. *Deciding if a pattern containment problem in $\mathcal{EL}_{\mathcal{RV}}$ has a solution is* EXPTIME-COMPLETE.

As matching is a special case of pattern containment, it also gives a tight upper bound for this problem.

Corollary 4. *Deciding if a matching problem in $\mathcal{EL}_{\mathcal{RV}}$ has a solution is* EXPTIME-COMPLETE.

5.7 Conclusion of Chapter 5

This chapter presents the algorithm *Check_Simu* which demonstrates that pattern containment in $\mathcal{EL}_{\mathcal{RV}}$ is decidable. Its main idea is to extend Check_Match by running synchronously the two automata A_Q and A_P. It leads to consider extended product configuration where both sides may contain variables transitions. As a consequence, we introduce the notion of constraints and set of constraints. The run of the algorithm is done regarding a global set of constraints to ensure synchronism.

For an epconf only specific variables and thus constraints are relevant regarding the counters. The extended cover, *epcover*, exploits this information to define a relationship which is independent of the counter. This extended notion led to demonstrate that the algorithm is correct. Changes required to adapt the algorithm to handle weak-subsumption are presented.

This chapter also emphasizes on the complexity analysis of the different reasoning tasks of $\mathcal{EL}_{\mathcal{RV}}$. The upper bound is given by the space of search which is exponential w.r.t the entries. Moreover, halting problem working on a space polynomially bounded by the size of the input of alternating turing machine can be reduced to matching problem. Alternating turing machines have three key points, initialization, exponential states and universal states where each of them corresponds to a specific structure with description automata. Since this problem is known to be EXPTIME-COMPLETE, we can infer, with the upper bound, that matching is EXPTIME-COMPLETE. Moreover, matching is a subproblem of both, weak-subsumption and pattern containment leading to both of them being EXPTIME-COMPLETE.

Conclusion

Our work explores reasoning with variables in description logics by introducing refreshing semantics. This semantics is characterized by the possibility to give a new assignment after releasing a variable's bound where the classic semantics does not offer any way to release a bound. The first contribution consists in introducing this semantics for $\mathcal{EL}$ which resulted in the definition of $\mathcal{EL}_{\mathcal{RV}}$, a logic coping with refreshing role variables using $\mathcal{EL}$-constructors. Pattern instances augment reasoning possibilities of $\mathcal{EL}_{\mathcal{RV}}$. These instances have been demonstrated to be either regular or irregular. In the case, two regular instances have a subsumption relationship in $\mathcal{EL}_{\mathcal{RV}}$, it implies a subsumption relation w.r.t to the greatest fix-points semantics in $\mathcal{EL}$. Thanks to this newly possible substitutions, reasoning in $\mathcal{EL}_{\mathcal{RV}}$ can go further than reasoning in $\mathcal{EL}$. Indeed, problems without solution may find a meaningful solution thanks to the refreshing semantics. We respectively extend matching and unification into matching and weak-subsumption in $\mathcal{EL}_{\mathcal{RV}}$. In addition, we introduce a third reasoning task, namely pattern containment, which allow to compare two queries whatever the considered terminology.

In order to address these problems in $\mathcal{EL}_{\mathcal{RV}}$ and to handle the potentially infinite instances, description automata have been defined. This class of automata can handle refreshing semantics for variables and allow to represent a pattern. Reasoning in $\mathcal{EL}_{\mathcal{RV}}$ can be reduced to variants of simulation between description automata. Pattern containment used the notion of universal simulation while weak-subsumption exploits existential simulation. Matching can be characterized by both of them since it is a sub-problem of both, weak-subsumption and pattern containment. Matching then acts as a lower bound for the three problems allowing to prove that they are optimal. The main results following: matching, weak-subsumption and pattern containment are EXPTIME-COMPLETE.

Works conducted in this thesis offer multiple research perspectives. Firstly, we considered only role variables however it would be interesting to consider concept variables. The main challenge comes from the fact that even for matching, the domain of valuation is infinite. Indeed, a concept variable may stand for any concept description. Current solutions do not handle such possibilities. In order to cope with concept variables, one interesting research direction would be to investigate a model that use refreshing role variables to simulate concept variables.

A result of this thesis puts in evidence the link between subsumption in $\mathcal{EL}_{\mathcal{RV}}$ and subsumption in $\mathcal{EL}$ with regard to the greatest fix-point semantics. It would be interesting to investigate if a link with descriptive semantics could be unraveled.

We presented the wide variety of potential matcher induced by the multiple pattern instances. A very interesting research axis would be to question the quality of a matcher.

How to answer the question "What is the best matcher ?". Usually, one chooses the closest concept to the targeted concept. However, in Section 2.2.4, we presented many matchers that are incomparable w.r.t to the subsumption relationship. Closeness of the concept might be the first property but might not suffice in such cases. Additional properties are required to qualify the best solution. For instance, a solution that requires the less number of definitions to be defined would be an interesting track. Note that such a criteria immediately makes regular solution of better quality than irregular ones. To go further, extending these results to more expressive logics based on $\mathcal{EL}$ such as $\mathcal{ALE}$ would be interesting.

Finally, a more consequent research axis would be to study another logic that offers both low complexity and interesting possibilities : $\mathcal{FL}_0$. This logic has been widely studied in presence of non-refreshing variables. Like $\mathcal{EL}$, $\mathcal{FL}_0$ subsumption can be characterized with automata theory. However, it is reduced to language inclusion instead of simulation. Description automata can be extended to $\mathcal{FL}_0$ but it remains to question language inclusion in order to exploit this link. Once again, it would be valuable to question regular and irregular matchers as well as decidability of reasoning in this framework.

Bibliography

[1] Baader, F., Küsters, R., Borgida, A., McGuinness, D.: Matching in description logics. Journal of Logic and Computation **9**(3), 411–447 (1999)

[2] Baader, F.: Using automata theory for characterizing the semantics of terminological cycles. Annals of Mathematics and Artificial Intelligence **18**(2), 175–219 (1996)

[3] Baader, F.: Computing the least common subsumer in the description logic $\mathcal{EL}$ w.r.t. terminological cycles with descriptive semantics. pp. 117–130 (07 2003)

[4] Baader, F.: Least common subsumers and most specific concepts in a description logic with existential restrictions and terminological cycles. In: Gottlob, G., Walsh, T. (eds.) IJCAI-03. pp. 319–324. Morgan Kaufmann (2003)

[5] Baader, F.: Terminological cycles in a description logic with existential restrictions. In: Gottlob, G., Walsh, T. (eds.) IJCAI. pp. 325–330. Morgan Kaufmann (2003)

[6] Baader, F., Borgwardt, S., Morawska, B.: Extending unification in el towards general tboxes. In: Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning (2012)

[7] Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope. pp. 364–369 (01 2005)

[8] Baader, F., Calvanese, D., McGuinness, D., Patel-Schneider, P., Nardi, D., et al.: The description logic handbook: Theory, implementation and applications. Cambridge university press (2003)

[9] Baader, F., Gil, O.F., Marantidis, P.: Matching in the description logic $\mathcal{FL}_0$ with respect to general tboxes. In: LPAR. pp. 76–94 (2018)

[10] Baader, F., Gil, O.F., Pensel, M.: Standard and non-standard inferences in the description logic $\mathcal{FL}_0$ using tree automata. In: GCAI. pp. 1–14 (2018)

[11] Baader, F., Gil, O.F., Rostamigiv, M.: Restricted unification in the dl $\mathcal{FL}_0$. In: Konev, B., Reger, G. (eds.) Frontiers of Combining Systems. pp. 81–97. Springer International Publishing, Cham (2021)

[12] Baader, F., Küsters, R.: Matching in description logics with existential restrictions. In: Description Logics (1999)

[13] Baader, F., Küsters, R., Molitor, R.: Computing least common subsumers in description logics with existential restrictions. In: Dean, T. (ed.) Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages. pp. 96–103. Morgan Kaufmann (1999)

[14] Baader, F., Morawska, B.: Unification in the description logic $\mathcal{EL}$. In: International Conference on Rewriting Techniques and Applications. pp. 350–364. Springer (2009)

[15] Baader, F., Morawska, B.: Matching with respect to general concept inclusions in the description logic $\mathcal{EL}$. In: Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz). pp. 135–146. Springer (2014)

[16] Baader, F., Narendran, P.: Unification of concept terms in description logics. Journal of Symbolic Computation **31**(3), 277–305 (2001)

[17] Baader, F., Rostamigiv, M.: Restricted unification in the dl $\mathcal{EL}$. In: Proceedings of the 34th International Workshop on Description Logics, CEUR (2021)

[18] Baget, J., Bienvenu, M., Mugnier, M., Thomazo, M.: Answering conjunctive regular path queries over guarded existential rules. In: Sierra, C. (ed.) Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017. pp. 793–799. ijcai.org (2017)

[19] Belkhir, W., Chevalier, Y., Rusinowitch, M.: Fresh-variable automata: Application to service composition. In: International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. pp. 153–160 (2013)

[20] Belkhir, W., Chevalier, Y., Rusinowitch, M.: Parametrized automata simulation and application to service composition. J. Symb. Comput. **69**, 40–60 (2015)

[21] Bodirsky, M., Chen, H.: Quantified equality constraints. In: LICS (2007)

[22] Borgida, A., Küsters, R.: "what's not in a name?" initial explorations of a structural approach to integrating large concept knowledge-bases (1999)

[23] Borgida, A., Brachman, R.J., McGuinness, D.L., Resnick, L.A.: Classic: A structural data model for objects. ACM Sigmod record **18**(2), 58–67 (1989)

[24] Borgida, A., McGuinness, D.L.: Asking queries about frames. In: Aiello, L.C., Doyle, J., Shapiro, S.C. (eds.) KR'96, Cambridge, Massachusetts, USA, 1996. pp. 340–349. Morgan Kaufmann (1996)

[25] Brandt, S.: Polynomial time reasoning in a description logic with existential restrictions, gci axioms, and - what else? pp. 298–302 (01 2004)

[26] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Data complexity of query answering in description logics. Artificial Intelligence **195**, 335–360 (2013)

[27] Chandra, A.K., Stockmeyer, L.J.: Alternation. In: 17th Annual Symposium on Foundations of Computer Science (sfcs 1976). pp. 98–108 (1976). https://doi.org/10.1109/SFCS.1976.4

[28] Côté, R.A.: Systematized nomenclature of human and veterinary medicine : Snomed international (1993)

[29] Glimm, B., Lutz, C., Horrocks, I., Sattler, U.: Conjunctive query answering for the description logic SHIQ. J. Artif. Intell. Res. **31**, 157–204 (2008)

[30] Henzinger, T., Qadeer, S., Rajamani, S., Tasiran, S.: An assume-guarantee rule for checking simulation. ACM Trans. Program. Lang. Syst. **24**, 51–64 (2002)

[31] Horrocks, I., Patel-Schneider, P., Mcguinness, D., Welty, C.: Owl: a description logic based ontology language for the semantic web (01 2007)

[32] Kazakov, Y., De Nivelle, H.: Subsumption of concepts in dl $\mathcal{FL}_0$ for (cyclic) terminologies with respect to descriptive semantics is pspace-complete (2003)

[33] Morawska, B.: Unification in the description logic $\mathcal{FL}_\bot$. In: Homola, M., Ryzhikov, V., Schmidt, R.A. (eds.) Proceedings of the 34th International Workshop on Description Logics (DL 2021) part of Bratislava Knowledge September (BAKS 2021), Bratislava, Slovakia, September 19th to 22nd, 2021. CEUR Workshop Proceedings, vol. 2954. CEUR-WS.org (2021), http://ceur-ws.org/Vol-2954/paper-24.pdf

[34] Nebel, B.: Terminological cycles. Springer (1990)

[35] Nebel, B.: Terminological reasoning is inherently intractable. Artificial Intelligence **43**(2), 235–249 (1990)

[36] Xiao, G., Calvanese, D., Kontchakov, R., Lembo, D., Poggi, A., Rosati, R., Zakharyaschev, M.: Ontology-based data access: A survey. In: Lang, J. (ed.) Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden. pp. 5511–5519. ijcai.org (2018)

[37] Zarrieß, B., Turhan, A.Y.: Most specific generalizations wrt general $\mathcal{EL}$-tboxes. In: Twenty-Third International Joint Conference on Artificial Intelligence. Citeseer (2013)