Reasoning in EL-description logic with refreshing variables

Théo Ducros, Marinette Bouet, Farouk Toumani

Reasoning in $\mathcal{EL}$-description logic with refreshing variables

Théo Ducros
Marinette Bouet
Farouk Toumani
theo.ducros@isima.fr
marinette.bouet@uca.fr
farouk.toumani@isima.fr
LIMOS
Aubière, France

ABSTRACT

Description logics have been widely studied and used in several knowledge-based systems. They allow to model knowledge and more importantly to reason over it. Subsumption relationship, a hierarchical relationship between concepts, is one of the most common reasoning task. Matching and unification generalize subsumption to description involving variables. In this paper, we study the problem of reasoning in description logics with variables. More specifically, we consider refreshing semantics for variables in the context of the $\mathcal{EL}$ description logic. We investigate a particular reasoning mechanism, namely *weak subsumption*, which can be viewed as a generalization of matching and unification in presence of refreshing variables. We show that weak-subsumption is EXPTIME-complete. Our main technical results are derived by establishing a correspondence between this logic and variable automata.

ACM Reference Format:
Théo Ducros, Marinette Bouet, and Farouk Toumani. 2021. Reasoning in $\mathcal{EL}$-description logic with refreshing variables. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Description Logics (DLs) are a family of knowledge representation and reasoning formalisms that have been proven useful in many application domains[4]. They 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. Description logics enable to describe a universe of discourse in terms of *concept descriptions*, i.e., expressions built from atomic concepts (unary predicates) and atomic roles (binary predicates) using the constructors of the considered description logic. For example, using the atomic concept $Person$ and the atomic role $haschild$, the concept of $Parent$ can be represented by the concept description

$$Person \sqcap \exists haschild.Person$$

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

While subsumption reasoning, i.e., the ability to determine subconcept–superconcept relationships, is a traditional reasoning in DL-based system additional inference mechanisms, such as matching [9] and unification [8], that go beyond subsumption, have been proposed in the literature. In this latter non-standard forms of reasoning, concept description with not completely specified form (incomplete information) can be specificed using the so-called concept patterns, i.e., concept descriptions containing variables. As an example, consider the following pattern which defines an $Academic$ as a $Person$ with a certain relationship with a $University$.

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

Here, the variable x takes its values from a set of possible atomic role names. The concept description

$$Academic \equiv Person \sqcap \exists worksIn.University$$

matches this pattern. Indeed, if we replace the variable x by the role $worksIn$, the pattern becomes equivalent to the description. Replacing a variable x with a value is called variable substitution. Given a description C and a pattern P, the matching problem asks then whether there is a variable substitution such that C matches P. The unification extends the matching to the case where C is itself a pattern. Consider now the case of a cyclic description:

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

and the following concept descriptions:

$$A \equiv Person \sqcap \exists worksIn.University \sqcap \exists hasAdviser.B$$

$$B \equiv Person \sqcap \exists presidentOf.University \sqcap \exists hasChild.B$$

Using standard semantics of variable substitution, A do not match the pattern academic. However, if we exploit a different semantics that enables to *refresh* the values of the variable x and y in each description of the pattern $Academic$, in this case it becomes possible to compute a matcher that makes the concept A matching the pattern $Academic$ (i.e., the first occurrences of x and y are respectively mapped to $worksIn$ and $hasAdvisers$ while their second occurences are respectively mapped to $prsidentof$ and $hasChild$).

This paper studies the extension of description logics with variables equipped with refreshing semantics. More specifically, we focus on a new description logic, called $\mathcal{EL}_V$, that extends the description logic $\mathcal{EL}$ with refreshing variables. Our definition of $\mathcal{EL}_V$-patterns deviates from the one used in the literature with respect to the following features:

(1) our definition of concept patterns use role variable while the literature focuses on concept variables,
(2) we support cyclic pattern definition and allow two different types of semantics for variables (i.e., refreshing and not refreshing semantics),

We consider in particular a new reasoning mechanism in this context, called *weak-subsumption*, which extends matching and unification to logics with refreshing variables. We show that testing weak-subsumption between $\mathcal{EL}_V$-patterns is EXPTIME-complete. Our main technical results are derived by establishing a correspondence between this logic and a specifc form of variable automata.

The paper is organized as follows. Section 2 presents the related works. Section 3 introduces a motivating example while preliminaries are given at Section 4. Section 5 is devoted to the presentation of the description logic $\mathcal{EL}_V$. Section 6 introduces a modelling of description under the shape of variable automata used in order to solve weak-subsumption. We conclude and draw future research direction at section 7.

2 RELATED WORKS

Description logics are used by knowledge representations systems such as Classic [14] or Loom [17] in order to represent a domain in a structured and formally well understood way. By using formal links defined between descriptions, reasoning tasks of descriptions logics can be used.

There are two levels of reasoning which can be defined: assertional and terminological levels. While assertional level focuses on instances of defined concepts by checking their consistency or satisfiability [15], terminological level is directed toward concept relationships themselves. Indeed, disjointedness, equivalence and notably subsumption are defined. Subsumption is one of the most basic and yet important reasoning mechanism which allows to discover information.

Subsumption focuses on determining whether a concept C subsumes - i.e. C is more general than - a concept D (noted $D \sqsubseteq C$). This mechanism presents obvious advantages to infer knowledge that is not directly expressed by exploiting DL's structural approach of knowledge. Up to now many works have solved subsumption for different logics in presence or not of terminological TBox.

In the case of family based on the description logic $\mathcal{FL}_0$ and $\mathcal{EL}$, we can distinguish two kinds of approaches to solve subsumption: normalize-compare algorithm and a tableau-based algorithm [18]. The first approach led to interesting results on how close description logics can be to automata [1]. Indeed, in $\mathcal{FL}_0$ subsumption can be reduced in language inclusion of finite automata [7] while $\mathcal{EL}$ subsumption will be equivalent to simulation [2].

Recently, description logics have been extending by introducing variables. Variables in description logic introduce pattern in addition to ground concept description. Subsumption between a pattern P and a ground description C ($C \sqsubseteq^? P$ or $P \sqsubseteq^? C$) is called a matching problem [6]. Matching is a new non-standard reasoning task that aims to find a substitution of variables such that $C \sqsubseteq \sigma(P)$ or $\sigma(P) \sqsubseteq C$. It has been studied for concept-description variables since considering role variable is trivial and can be done by enumerating all the possibilities. Matching has been proven to be polynomial in $\mathcal{FL}_0$ without TBox [6] while considering a general TBox blows up the complexity to EXPTIME [5]. Interestingly enough, $\mathcal{EL}$ proposes a more complex matching problem by achieving NP-Completeness without TBox [9] but does not suffer any blow up of complexity when considering a general TBox[10]. The different results were achieved by reducing matching to automata theory after adopting a well-adapted normal form. Only matching w.r.t to general Tbox in $\mathcal{EL}$ differs by proposing a goal-oriented algorithm that uses a non-deterministic rule to transform a given matching problem into a solved form.

The generalisation of matching involving two patterns P,Q ($P \sqsubseteq Q$) is named unification [11]. Unification has also been investigated in $\mathcal{EL}$ and $\mathcal{FL}_0$. Recently, unification in $\mathcal{EL}$ has made a huge step forward by achieving NP-Completeness w.r.t a general TBox that fulfills a restriction on cycles [3]. Unfortunately, the general case is still an open problem since the proposed algorithm is not complete. As for matching in $\mathcal{EL}$, a goal-oriented algorithm has been employed by extending the one proposed in [10].

So far matching and unification have proven to be useful in order to filter out important aspects of large concepts in classic [12]. Baader et al. [11] proposed to use those reasoning tasks as a tool to find and thus prevent redundancies in knowledge base. These reasoning mechanisms can also be used to support integration of knowledge bases by prompting interscheme assertions to the integrator [13].

Variables in description logics allowed to extend subsumption to non-standard reasoning tasks namely matching and unification. These tasks have been well-studied and proven to be useful. However, boundaries of the variables can be pushed up even further by introducing refreshing semantic. Combination of description logics variables and refreshing semantic is an interesting new possibility. By making the potential of variables even greater, new possibilities are offered to reason in description logics.

Next section offers a close-up to advantages of introducing refreshing semantic.

3 MOTIVATING EXAMPLES

This section aims to illustrate the potential of introducing refreshing semantic in description logic using $\mathcal{EL}$ as example. Consequently, we will save technical aspects for later sections and emphasize here on presenting how valuable it can be. Examples are based on the following terminology.

EXAMPLE 1.
$Doctor = Person \sqcap \exists getPhDIn.Univ \sqcap \exists formerly.PhDStudent$
$PhDStudent = Person \sqcap \exists studyIn.Univ \sqcap \exists supervisedBy.Doctor$

To this simple TBox, the pattern $Academic \equiv Person \sqcap \exists x.University \sqcap \exists y.Academic$ is added with x, y being role variables. From a state of the art point of view, there are a finite number of substitutions that can be applied to x and y. Unfortunately, none of these substitutions leads to a subsumption relationship with a defined concept of the TBox. In other words, Academic is not subsumed nor subsumes Doctor or PhDStudent. It means that matching problems involving *Academic* and defined concepts of this example are unsolvable. We will now assume that x and y can take new value each time we replace Academic by its own definition. As a consequence,it is possible to alternate between $\{x \mapsto getPhDIn, y \mapsto formerly\}$ and $\{x \mapsto studyIn, y \mapsto supervisedBy\}$ which would conduct to an

equivalent definition of *Doctor* under the greatest fix-point semantic. Therefore there is now a substitution such that a subsumption relationship exists between Doctor and an instance of Academic.

The same statements can be done regarding concept description variables. The next example focuses on explaining how better solution to solvable matching problem can be found in presence of refreshing semantic. This time and to be more complete, we will use concept variable to illustrate it. The following concept will be added to the original TBox of Example 1:

$FrenchDoctor = Person \sqcap \exists getPhDIn.FrenchUniv \sqcap \exists formerly.FrenchPhDStudent$
$FrenchPhDStudent = Person \sqcap \exists studyIn.FrenchUniv \sqcap \exists supervisedBy.Doctor$
$FrenchUniversity = University \sqcap \exists located.France$

We will then consider the pattern Academic2 defined as follows : $Academic2 \equiv Person \sqcap \exists getPhdIn.X \sqcap \exists formerly.(Person \sqcap \exists studyIn.X \sqcap \exists supervisedBy.Academic2)$

We will focus on comparing the pattern to the newly added concept FrenchDoctor. Obviously, taking $X = \top$ or $University$ implies that FrenchDoctor is more specific than Academic2. On the other hand, taking $X = FrenchUniversity$ would reverse the subsumption relationship making Academic2 more specific. However, equivalence can never be achieved. If we consider X as a refreshing variable, we can say that the first time X maps to $FrenchUniversity$ and then it will continuously be mapped to $University$. This solution allows to achieve an equivalent concept to FrenchDoctor. Note that none of the solutions, before changing X's nature from non-refreshing to refreshing variable, could achieve it. This solution is then a better solution in the sense that it is closer to the targeted concept than any other.

4 PRELIMINARIES

We use the following definition of a tree [16]: 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.

Let N_C be a set of concept names and let N_R be a set of role names. We use the letters A, B to range over N_C; R, S to range over N_R; and C, D to range over $\mathcal{EL}$-concept descriptions (or simply, $\mathcal{EL}$-concepts), which are formulas inductively generated by the following rule:

$$\top \mid A \mid C \sqcap D \mid \exists R.D$$

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 Table 1.

A simple *Tbox* $\mathcal{T}$ is 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 that occur on the left-hand side of a definition are called *defined concepts*, and denoted by the set N_{def}, while all the other concepts occurring in $\mathcal{T}$ are called *atomic concepts* and are denoted by the set N_A. We allow for cyclic dependencies between the defined concepts, i.e., a definition of an $\mathcal{EL}$-concept P may directly or indirectly refer to P itself. 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 $\mathcal{T}$.

Name	Syntax	Semantic
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	$D \sqcap E$	$D^{\mathcal{I}} \cap E^{\mathcal{I}}$
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: Interpretation of $\mathcal{EL}$'s constructors

5 THE DESCRIPTION LOGIC $\mathcal{EL}_V$

An $\mathcal{EL}_V$-signature is a pair $\Sigma = (N_C, N_T)$, where N_C is the set of concept names 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 $\mathcal{EL}_V$ extends the logic $\mathcal{EL}$ with role variables. Given a signature $\Sigma = (N_C, N_T)$, $\mathcal{EL}_V$-concept descriptions are built similarly to $\mathcal{EL}$ concepts while using roles terms instead of only role names.

EXAMPLE 2. *Let $x \in N_{V_R}$ and $y, z \in N_{V_N}$, then the following is an $\mathcal{EL}_V$-concept definition:*

$$B_1 \equiv Pers \sqcap \exists z.Univ$$

$$Acad \equiv Pers \sqcap \exists x.Univ \sqcap \exists y.Acad$$

The $\mathcal{EL}_V$-concept B_1 is defined as Person which has a relationship with a University while the cyclic $\mathcal{EL}_V$-concept Academic is defined as Person which has a relationship with a University and another relationship with an Academic.

An $\mathcal{EL}_V$-TBox is a set of $\mathcal{EL}_V$-concept definitions. We present now the notion of normalized $\mathcal{EL}_V$-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 $\mathcal{EL}_V$-signature and let $\mathcal{T}$ be an $\mathcal{EL}_V$-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:

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

for $n, m \geq 0$ and $A_i \in N_A$ and $r_i \in N_T, \forall i \in [0, n]$ and $B_j \in N_{def}, \forall j \in [0, m]$.

In the sequel, we assume that the $\mathcal{EL}_V$-TBoxes are normalized.

EXAMPLE 3. *Back to Example 2, C_1 must be introduced in order to normalize B_1 and Academic since University is not a defined*

concept but an atomic one.

$$C_1 = Univ$$
$$B_1 \equiv Pers \sqcap \exists z.C_1$$
$$Acad \equiv Pers \sqcap \exists x.C_1 \sqcap \exists y.Acad$$

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 $\mathcal{EL}_V$-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* periodically. To illustrate our purpose, we use subscripts (i.e., $\sigma_1, \sigma_2, \ldots$) to denote the fact that a substitution σ maps a refreshing variable x to several values. Assume a substitution σ that maps the non refreshing variable y to a role name $sonOf$ (i.e., $\sigma_i(y) = sonOf, \forall i \in \mathbb{N}$) and it maps the first occurrence of the refreshing variable x to $presidentOf$ (i.e., $\sigma_1(x) = presidentOf$)) while it maps the second occurrence of x to $worksIn$ (i.e., $\sigma_2(x) = worksIn$). This leads to the following $\mathcal{EL}$-TBox:

$$\sigma_1(Acad) \equiv Pers \sqcap \exists \underbrace{presOf}_{\sigma_1(x)} . \underbrace{Univ}_{\sigma(C_1)} \sqcap \exists \underbrace{sonOf}_{\sigma(y)} .(Pers \sqcap \exists \underbrace{workIn}_{\sigma_2(x)} . \underbrace{Univ}_{\sigma(C_1)} \sqcap \exists \underbrace{sonOf}_{\sigma(y)} .\sigma_3(Acad))$$

To define formally the notion of instance of an $\mathcal{EL}_V$-concept in presence of refreshing variables, we first turn $\mathcal{EL}_V$-descriptions with refreshing variables to equivalent infinite $\mathcal{EL}_V$-descriptions with non refreshing variables. This is achieved by the following *unfolding process* which replaces refreshing variables appearing in cyclic definitions of a given terminology by an infinite set of non refreshing variables.

DEFINITION 1. ***(Pattern unfolding)***
Let $\mathcal{T}$ be an $\mathcal{EL}_V$-TBox over a $\mathcal{EL}_V$-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 $\mathcal{EL}_V$-signature $(N_C, N_R \cup N_{V_N})$ such that each $\mathcal{EL}_V$-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 $\mathcal{EL}_V$-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 primitive 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 $\mathcal{EL}_V$-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 of a cyclic definition of an $\mathcal{EL}_V$-pattern P, the unfolding of P leads to an infinite $\mathcal{EL}_V$-pattern $u(P)$.

EXAMPLE 4. *We show below partial unfolding of the $\mathcal{EL}_V$-pattern Academic*

$$u(Acad) \equiv \underbrace{Pers}_{u(Pers)} \sqcap \exists \underbrace{x_1}_{u(x)} . \underbrace{u(Univ)}_{u(C_1)} \sqcap \exists \underbrace{y}_{u(y)} .(Pers \sqcap \exists \underbrace{x2}_{u(x)} . \underbrace{u(Univ)}_{u(C_1)} \sqcap \exists \underbrace{y}_{u(y)} .u(Acad))$$

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

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

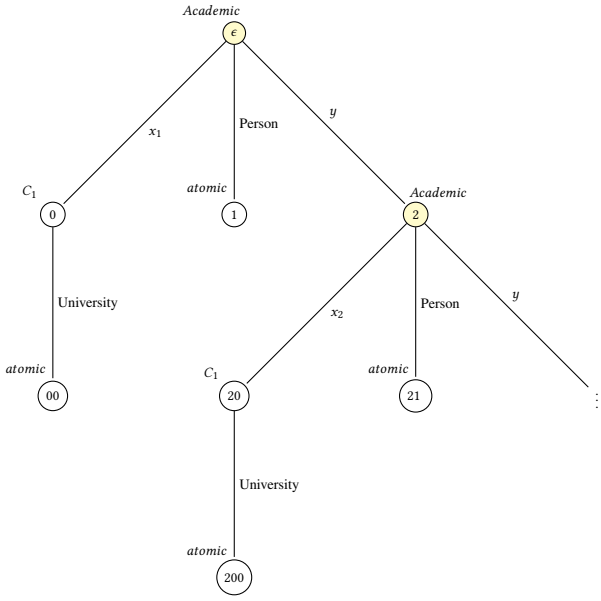

Figure 1: Tree of unfolded academic

EXAMPLE 5. *Figure 1 represents the tree representation of Academic unfolded. Each node is labeled by the corresponding concept in the TBox. If they participate in a cycle, they will appear an infinite number of times. Leaves are only labeled by atomic since atomic concept always leads to an end of unfolding.*

Instanciations of $\mathcal{EL}_V$-concept definitions (respectively, $\mathcal{EL}_V$-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 $\mathcal{EL}_V$-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 $\mathcal{EL}_V$-concepts;
- $\sigma(\exists T.C) = \exists \sigma(T).\sigma(C)$.

In addition, a substitution σ maps each $\mathcal{EL}_V$-TBox $\mathcal{T}$ into an $\mathcal{EL}$-TBox $\sigma(\mathcal{T})$ which is obtained by converting each $\mathcal{EL}_V$-concept definition $P \equiv C$ in $\mathcal{T}$ into an $\mathcal{EL}$-concept definition $\sigma(P) \equiv \sigma(C)$.

In this case, the $\mathcal{EL}$-concept $\sigma(P)$ is called an instance of the $\mathcal{EL}_V$-concept A.

DEFINITION 2. ***(Pattern Instances)***
Let $\mathcal{T}$ be an $\mathcal{EL}_V$-TBox over a $\mathcal{EL}_V$-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 $P = A_0 \sqcap ... \sqcap A_n \sqcap \exists r_0.B_0 \sqcap ... \sqcap \exists r_m.B_m$ be an $\mathcal{EL}_V$-pattern in $\mathcal{T}$. Let $\phi : N_{V_N} \rightarrow N_R$ be a variable substitution. Then $\phi(u(P))$ is an instance of P w.r.t. the variable substitution ϕ.

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.

EXAMPLE 6. *Consider two substitutions σ and θ that respectively map the variable z as follows: $\sigma(z) = workIn$ and $\theta(z) = gradFrom$. Each of these substitutions leads to an instantiation of the $\mathcal{EL}_V$-concept B_1 of example 2 as follows: $\sigma(B_1) \equiv Pers \sqcap \exists workIn.Univ$ and $\theta(B_1) \equiv Pers \sqcap \exists gradFrom.Univ$. On the other hand, if we consider $\sigma(y) = sonOf$ and $\sigma(x)$ such that it alternates between $workIn$ and $presOf$. It leads to the following instantiation of Acad. Note that this instantiation is not achievable by using only non-refreshing variables.*

$$\sigma_1(Acad) \equiv Pers \sqcap \exists \underbrace{presOf}_{\sigma_1(x)}.\underbrace{Univ}_{\sigma(C_1)} \sqcap \exists \underbrace{sonOf}_{\sigma(y)}.(Pers \sqcap \exists \underbrace{workIn}_{\sigma_2(x)}.\underbrace{Univ}_{\sigma(C_1)} \sqcap \exists \underbrace{sonOf}_{\sigma(y)}.\sigma_1(Acad))$$

Various kinds of reasoning could be defined over $\mathcal{EL}_V$-terminologies. We focus on this paper on one specific reasoning mechanism, called hereafter *weak subsumption*.

DEFINITION 3. ***(Weak subsumption)***
Let $\mathcal{T}$ be an $\mathcal{EL}_V$-TBox and let P, Q two $\mathcal{EL}_V$-patterns. Then, P is weakly subsumed by Q w.r.t. $\mathcal{T}$, denoted $P \overset{\sqsubseteq}{\sim}_{\mathcal{T}} Q$, iff exists two substitutions ϕ_1 and ϕ_2 s.t $\phi_1(P) \sqsubseteq_{\mathcal{T}} \phi_2(Q)$

Note that weak subsumption can be viewed as an extension of respectively matching [9, 10] when either P or Q is a ground $\mathcal{EL}$-concept and unification [8] when both P and Q are $\mathcal{EL}_V$-patterns to logic with refreshing variables.

6 $\mathcal{EL}_V$-DESCRIPTION AUTOMATON

Our reasoning procedures over $\mathcal{EL}_V$-terminologies are built on the notions of $\mathcal{EL}_V$-description automata. Such automata recognize *configuration trees* which are nothing other than a syntactic variant of pattern instances. As a main result of this section, stated by lemma 1, we associate with each $\mathcal{EL}_V$-pattern P an $\mathcal{EL}_V$-description automata A_P such that there is a bijection between the configuration trees recognized by A_P and the instances of P. Consequently, an $\mathcal{EL}_V$-description automaton A_P characterizes all the possible instances of its associated $\mathcal{EL}_V$-pattern P.

DEFINITION 4. ***($\mathcal{EL}_V$-description automata)***
Let $\mathcal{T}$ be an $\mathcal{EL}_V$-TBox over the signature $\Sigma = (N_C, N_T)$, with $N_C = N_{def} \cup N_A$, $N_T = N_R \cup \mathcal{V}$ and $\mathcal{V} = N_{V_R} \cup N_{V_N}$ and let $P \equiv A_0 \sqcap ... \sqcap A_n \sqcap \exists r_0.B_0 \sqcap ... \sqcap \exists r_m.B_m$ be a defined concept in $\mathcal{T}$. The $\mathcal{EL}_V$-description automaton associated with P, denoted A_P, is a tuple $A_P = (\mathcal{L}, \mathcal{V}ar, \mathcal{Q}, q_0, q_f, \delta, \kappa)$ recursively build as:

- $\mathcal{L} \subseteq N_A \cup N_R$ *is a finite alphabet,*
- $\mathcal{V}ar \subseteq \mathcal{V}$ *is a finite set of variables,*
- $\mathcal{Q} = N_{def} \cup \{q_f\}$ *is a finite set of states,*
- $q_0 = P$ *is the initial state and q_f is the final state,*
- $\delta \subseteq \mathcal{Q} \times (\mathcal{L} \cup \mathcal{V}ar) \times \mathcal{Q}$ *is a transition relation defined as follows: $\delta = \{(P, A_i, q_f) : for\ i \in [0, n]\} \cup \{(P, r_j, B_j) : for\ j \in [0, m]\}$ (where B_j is the initial state of the automaton A_{B_j})*
- $\kappa : \mathcal{V}ar \rightarrow \in^{\mathcal{Q}}$ *is the refreshing function defined as follows: $\forall r_i \in \mathcal{V}ar$ we have: $\kappa(r_i) = \{P\}$ if $r_i \in N_{V_R}$, or $\kappa(r_i) = \emptyset$ if $r_i \in N_{V_N}$.*

Definition 4 associates to 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 $\mathcal{T}$ and $\mathcal{EL}_V$-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 primitive 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, in this case it is refreshed in the state P and its refreshing state is given by the function κ (i.e., $\kappa(r_i) = \{P\}$).

Figure 2 depicts the description automaton of the pattern Academic introduced previously. The variable x is non-refreshing while the variable y is refreshing with $\kappa(y) = \{Academic\}$ its refreshing state.

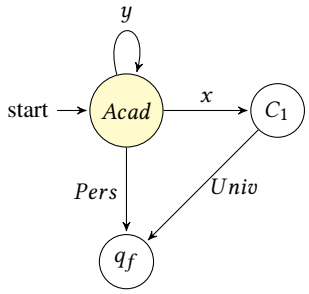

Figure 2: The description automaton $A_{Academic}$.

A description automaton of a given pattern P is in fact a compacted representation of all the possible instantiation of P. To make this statement more precise, we introduce the notions of configurations and configuration trees and we show that these latter ones are *equivalent* to the instances of P. Informally, a configuration of A_P 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 changed at their refreshing states, a configuration includes a vector of integer used to distinguish between multiple value assignations to a given refreshing variables. More precisely, we define a configuration as a pair (q, I) where q is a state of A_P and I is a vector of integers, where the i^{th} component of 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 variables by incrementing its corresponding component in the vector I.

Figure 4 represents a configuration tree based on the description automaton of Academic. A configuration c is a pair (q_c, I_c) made of a state q_c and a counter I_c. Using such a pair enables to exploit the potential of variable transitions. Indeed, variables will be replaced during a transition by elements of $\mathcal{L}$. In the tree, the configuration is used as a label.

Each configuration automaton of A_P regarding $\sigma : \mathcal{V}ar \times \mathbb{N} \mapsto N_R$ corresponds to the description tree of $\phi(P)$. For example, the configuration tree of Figure 4 depicted the beginning of $\sigma(Academic)$ presented above.

We will use the following notation. Let $G \in \mathbb{N}^{|\mathcal{V}ar|}$ a tuple of integer. 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 G, noted G_x, which gives the value of the counter of x in G.

DEFINITION 5. ***(Configuration and configuration tree)*** *Let $A_P = (Q, \mathcal{L}, \mathcal{V}ar, q_0, \delta, q_f, \kappa)$ be a description automaton and let σ: $\mathcal{V}ar \times \mathbb{N} \mapsto N_R$ be a variable substitution.*

- *A configuration is a pair (q, I) where $q \in Q$ is a state of A_P and $I \in \mathbb{N}^{|\mathcal{V}ar|}$ is a vector of integers.*
- *Let $S \subseteq Q \times \mathbb{N}^{|\mathcal{V}ar|}$. A run of A_P using a substitution σ, denoted $T(A_P, \sigma)$ and called a configuration tree, is a $\langle S, \mathcal{L}\rangle$-labeled tree (τ, λ, δ) constructed using the Algorithm 1.*

Definition 5 formally defines a configuration tree. As stated above, configurations are of the form (q_c, I_c). The state q_c refers to states of A_P, it allows to know in what states we are and what states are reachable.

On the other hand, I_c is a counter acting as a stamp which keeps track of time and order. Thanks to it, we know what instance of a variable has to be considered. It allows to identify variables and on the top of that, to synchronize the same instances of variables.

The counter is directly impacted by refreshment since refreshing means new instances. Since its role is to keep track of the current instance, it then updates values if a refreshing state is encountered. This update takes the shape of an increasing of the variables' associated counters. However, this increasing is not done randomly, since any path leading to a refreshing states must lead to a brand new instance, a global counter is used to ensure it. Not doing so could imply that different instances of a variable would be synchronized for no reason. To sum up a configuration c is a pair (q_c, I_c) where $q_c \in Q_P$ and $I_c \in \mathbb{N}^{|\mathcal{V}ar|}$ which acts as a local stamp.

By defining relation over configuration with regards to a function $\mathcal{V}ar \times \mathbb{N} \mapsto N_R$, it makes possible to give a specific value to a specific instance of a variable.

Lemma 1 enlightens the link between valuation in description logic and configuration automaton in automata theory. It says that for each substitution of a pattern P in description logic there exists an isomorphic automaton among configuration trees recognized by A_P. Figure 3 proposes a graphical and easier to understand representation of those relationships.

DEFINITION 6. ***(Equivalence between variable's substitution)*** *Let $\mathcal{T}$ be a Tbox with the pattern P and $A_P = (Q, \mathcal{L}, \mathcal{V}ar, q_0, \delta, q_f, \kappa)$ its automaton.*

Algorithm 1 GenConfTree

Input : $A_P = (Q, \mathcal{L}, \mathcal{V}ar, q_0, \delta, q_f, \kappa)$, σ
Output : τ ▷ The configuration tree $T(A_P, \sigma)$

$G \leftarrow \vec{0}$ ▷ A global counter initialized to a null vector
$I_0 \leftarrow \vec{0}$ ▷ A local counter of the initial configuration
$n \leftarrow \epsilon$
$\tau \leftarrow \{n\}$
$\lambda(\epsilon) \leftarrow (q_0, I_0)$
$q \leftarrow q_0$
create a queue Qstate
create a queue Qtrans
enqueue (q_0, I_0, n) onto Qstate
while Qstate is not empty **do**
  $(q, I, n) \leftarrow Qstate.dequeeue()$
  for all $(q, x, q') \in \delta$ **do**
    enqueue (x, q') onto Qtrans
    sort Qtrans in lexicographical order
  end for
  for i $\leftarrow 0$ to $|Qtrans|$ **do**
    $\tau \leftarrow \tau \cup \{ni\}$ ▷ ni is obtained by concatenating n and i
    $(x, q') \leftarrow QTrans.dequeue()$
    if $x \in \mathcal{V}ar$ **then**
      $\delta(n, ni) \leftarrow \sigma(x, I_x)$
    else
      $\delta(n, ni) \leftarrow x$
    end if
    for all $y \in \mathcal{V}ar$ **do**
      if $q' \in \kappa(y)$ **then**
        $G_y \leftarrow G_y + 1$ and $J_y \leftarrow G_y$
      else
        $J_y \leftarrow I_y$
      end if
    end for
    $\lambda(ni) \leftarrow (q', J)$
    enqueue (q', J, ni) onto Qstate
  end for
end while
return τ

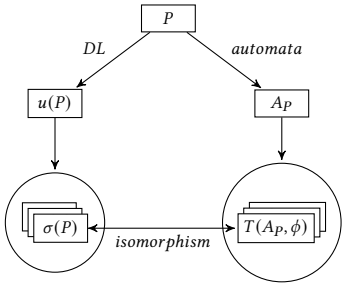

Figure 3: Schematical relationships between representations

Let $\phi : N_{V_N} \rightarrow N_R$ and $\sigma : \mathcal{V}ar \times \mathbb{N} \rightarrow N_R$
We say that ϕ is equivalent to σ, denoted $\phi \equiv \sigma$, if and only if there exists a bijective function $f : N_{V_N} \rightarrow \mathcal{V}ar \times \mathbb{N}$ such that $\phi(x) = \sigma(f(x))$

LEMMA 1. *Let P be an $\mathcal{EL}_V$-pattern of a terminology $\mathcal{T}$ and let A_P be its corresponding description automaton. Then, we have:*

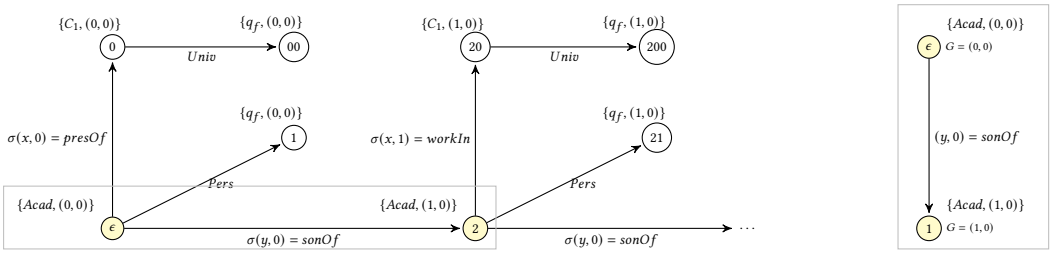

Figure 4: Example of configuration tree for "Academic"

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

PROOF. The general idea is to demonstrate that using equivalence between ϕ and σ, it is possible to have a $T(A_\sigma, P)$ such that it is structurally isomorphic to the tree representation of $\phi(P)$ Let consider $\phi(P) = (\tau_P, \lambda_P, \delta_P)$ and $T(A_\sigma, P) = (\tau, \lambda, \delta)$ created using the same lexical order. By definition, we immediately have $\tau_P = \tau$. Therefore, we will focus on their label and the edges.

If we look at the root ϵ of each tree, we have that $\lambda_P(\epsilon) = \{P\}$ and $\lambda(\epsilon) = \{P, \overrightarrow{0}\}$. Since, they refers to the same concepts, it remains to show that their edges are identical. Constant edges will be automatically present and since the same order has been applied $\delta_P(\epsilon, i) = \delta(\epsilon, i) = r$.

For variable edges, we have $\delta(\epsilon, i) = \phi(x)$ and $\delta(\epsilon, i) = \sigma(x', 0)$. If $x = x'$, it is a non refreshing variable and we can construct f such that $f(x) = (x, 0)$. Otherwise, we are dealing with a refreshing variable. Thanks to the order, we can complete f with f(x')=(x,0). Therefore, we have can construct $\sigma(x, 0) = \sigma(f(x') = \phi(x')$ for each variable transition. f can then be recursively defined since the label of the reach node have the same property as the first one. If a variable is refreshed, the counter will changed accordingly in τ and a new variable will be used in τ_P bounding these two refreshments thanks to f. Consequently, we can say that ϕ and σ are equivalent which complete the proof. The same proof can be used by taking f^{-1} instead of f which is possible since f is bijective. □

Using A_P is valuable because it allows to represent in a finite way, infinite number of possibilities. Thus making possible to reason and solve problems such as simulation while a naive approach would not terminate. The next section presents simulation in the scope of those automaton.

6.1 Characterizing weak subsumption using description automaton

We extend the notion of simulation relation, used in [2] to characterize subsumption between $\mathcal{EL}$-descriptions, to configuration trees.

DEFINITION 7. ***(simulation between configuration trees)*** *Let $T(A_Q, \phi) = (\tau_1, \lambda_1, \delta_1)$ and $T(A_P, \sigma) = (\tau_2, \lambda_2, \delta_2)$ be respectively two configuration trees of two description automata A_Q and A_P A binary relation $Z \subseteq \tau_1 \times \tau_2$ is a simulation relation iff*

(1) $(\epsilon, \epsilon) \in Z$, and

(2) if $(c_1, c_2) \in Z$ then $\forall (c_1, r, c_1') \in \delta_1$, $\exists c_2' \in \tau_2$ such that $(c_1', c_2') \in Z$ and $(c_2, r, c_2') \in \delta_2$

In this case we say that $T(A_Q, \phi)$ is simulated by $T(A_P, \sigma)$. If $(c_1, c_2) \in Z$, we say that c_1 is simulated by c_2. We extend simulation to configurations and we say $\lambda(c_1)$ is simulated by $\lambda(c_2)$ if $(c_1, c_2) \in Z$.

We use definition 7 to introduce a notion of existential simulation between $\mathcal{EL}_V$-description automata.

DEFINITION 8. ***(Existential simulation)*** *Let A_P and A_Q be two description automata. There is an existential simulation from A_P to A_Q, denoted $A_Q \ll_\exists A_P$, iff $\exists \sigma, \phi$ such that $T(A_Q, \sigma) \ll T(A_P, \phi)$.*

We give now our main technical result consisting in the characterization of weak simulation between $\mathcal{EL}_V$-patterns in terms of existential simulation between $\mathcal{EL}_V$-description graphs.

THEOREM 1. *Let P and Q two patterns of $\mathcal{EL}_{RV}$ under normal form. Let A_Q be the automata for Q while A_P represents P. A weak-subsumption problem $P \sqsubseteq\!\!\!\sim Q$ has a solution if and only if $A_C \ll_\exists A_P$.*

By definition of simulation, if $A_Q \ll_\exists A_P$, it means that $\exists \phi, \sigma$ such that $T(A_Q, \phi) \ll T(A_P, \sigma)$. Lemma 1 states that for any $T(A_P, \sigma)$ there exists an equivalent concept description $\sigma'(P)$. Naturally, the same goes for $T(A_Q, \phi)$ and thus there exists $\phi'(Q)$ Consequently we have, $T(A_Q, \phi) \ll T(A_P, \sigma)$ which implies $\sigma'(P) \sqsubseteq \phi'(Q)$ which is the definition of $P \sqsubseteq\!\!\!\sim Q$.

The next section investigates the problem of testing existential simulation between $\mathcal{EL}_V$-description automota.

6.2 Existential simulation between $\mathcal{EL}_V$-description automata

We shall use the following example through this section.

EXAMPLE 7. *Consider the following $\mathcal{EL}_V$-terminology where all the variables that occur in $\mathcal{EL}_V$-descriptions are refreshing variables.*

$$A_1 = A$$
$$B_1 = B$$
$$C = \exists R.B_1$$
$$P = \exists x.P \sqcap \exists z.C \sqcap \exists z.B_1 \sqcap \exists S.A_1$$
$$Q = \exists R.Q_2 \sqcap \exists R.C \sqcap \exists y.B_1$$
$$Q_2 = \exists R.Q \sqcap \exists S.B_1 \sqcap \exists S.A_1$$

Figures 5 and 6 depict respectively the $\mathcal{EL}_V$-description automata of the $\mathcal{EL}_V$-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., $\kappa(y) = \{Q\}$).

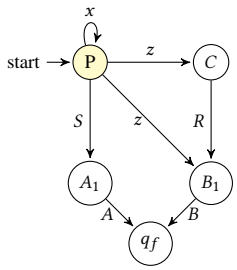

Figure 5: Description automata of P

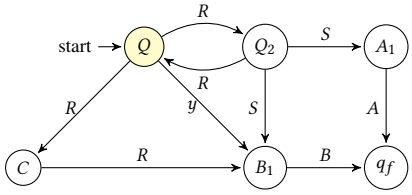

Figure 6: Description automata of Q

In this section, we propose an algorithm, called Check_Simu, to test existential simulation between $\mathcal{EL}_V$-description graphs, we prove its correctness and show that it is EXPTIME-complete. The Check_Simu algorithm is based on a synchronized product of automata. Given two $\mathcal{EL}_V$-description automata A_Q and A_P, the main idea of Check_Simu is to run synchronously A_Q and A_P, trying at each step to guess appropriate values assignments to variables in order to construct two variable substitutions ϕ and σ such that $T(A_Q, \sigma) \ll T(A_P, \phi)$. A given state in such a synchronized product is called a *pconf* (for product configuration). A *pconf* includes a set of *constraints* used to keep track of the value assignments made by the synchronized product. By exploring the possible synchronized executions of A_Q and A_P, the algorithm Check_Simu tries to construct two variable substitutions ϕ and σ that satisfy the constraints of each explored *pconf*. When the algorithm succeed in constructing ϕ and σ, this ensures that $T(A_Q, \sigma) \ll T(A_P, \phi)$ (soundness of the algorithm Check_Simu). In case, the algorithm Check_Simu fails to exhibit such substitutions it ensures that no such substitutions exists (completeness of the algorithm Check_Simu). The completeness is due to an exhaustive exploration of the possibilities of synchronization between A_Q and A_P. Moreover, since a synchronized execution of A_Q and A_P leads to an infinite space (i.e., the set of *pconf* to explore is infinite), we rest on a specific property, hereafter called *pcover* (for cover between *pconf*) , in order to ensure that the algorithm Check_Simu terminates. In the rest of this section, we define formally the notions of *constraints*, *pconf* and *pcover* before presenting the algorithm Check_Simu.

DEFINITION 9. ***(Constraint)***
A constraint is a statement of one of the following forms:

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

where $x, y \in \mathcal{V}$, $i, j \in \mathbb{N}$ and a,b are constants. A constraint of the form $a = b$ where the constant a syntactically differs from b is said inconsistent.

We extend the notion of inconsistency to sets of constraints as follows. Let $\mathbb{C}$ be a set of constraints. We denote by $\mathbb{C}^T$ the transitive closure of $\mathbb{C}$ w.r.t. to the equality relationship. We say that a set of constraints $\mathbb{C}$ is inconsistent iff its transitive closure $\mathbb{C}^T$ contains at least one inconsistent constraint.

We formally define now the notion of *pconf*.

DEFINITION 10. ***(pconf)***
Let A_Q, A_P two description automata. A pconf is a triple $(S_Q, p, \mathbb{C})$ where S_Q is a set of configurations of A_Q, p is a configuration of A_P and $\mathbb{C}$ a set of constraints.

Let $pc = (S_Q, p, \mathbb{C})$ be a *pconf*. Since several configurations appear in an pc, all the constraints of $\mathbb{C}$ are not necessarily relevant for each configuration in pc. For example, let consider a *pconf* $pc = (\{(q_c, (3))\}, (P, (1, 1)), \{(x, 2) = 1, (x, 3) = 5, (y, 0) = (x, 2)\})$. The configuration $(q_c, (3))$ has only one counter, whose current value is 3 and we assume that this counter is associated with the variable x. Therefore, we know that for the configuration $(q_c, (3))$ the only instance of x to consider is $(x, 3)$. Hence, if a constraint does not deal with $(x, 3)$, it doesn't carry any information for configuration $(q_c, (3))$. In our example, the constraints $(x, 2) = 1$ and $(y, 0) = (x, 2)$ are then not relevant for the configuration $(q_c, (3))$. Next definition formalizes this notion.

DEFINITION 11. ***(Relevant constraints w.r.t. a configuration)***
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 $R_{\mathcal{V}}(q_c) = \{x, (I_{c_x})\}$.*
- *The relevant constraints of $\mathbb{C}$ w.r.t. a configuration c is defined as: $\mathbb{C}_{|c} = \{(x = y) \in \mathbb{C}^T | (x \in R_{\mathcal{V}}(q_c) \vee y \in R_{\mathcal{V}}(q_c)\}$.*

We define now a notion of inclusion between sets of constraints.

DEFINITION 12. ***(Constraint set inclusion)***
Let $\mathbb{C}_1$ and $\mathbb{C}_2$ be two sets of constraint. The set $\mathbb{C}_1$ is included in $\mathbb{C}_2$, noted $\mathbb{C}_1 \subseteq \mathbb{C}_2$

(1) $\forall (x, i) = (y, j)) \in \mathbb{C}_1$, $\exists (x, k) = (y, l) \in \mathbb{C}_2$ and
(2) $\forall (x, i) = a) \in \mathbb{C}_1$, $\exists (x, k) = a) \in \mathbb{C}_1$

EXAMPLE 8. *Let consider A_P and A_Q defined above. Let $pc = (\{(Q_2, (0))\}, (P, (1, 1)), y_0 = R)$ be one of their pconf. For each of configuration of pc, their associated set of relevant variables are : $R_{\mathcal{V}}((Q_2, 0)) = \{y_0\}$, $R_{\mathcal{V}}((P, (1, 1)) = \{x_1, z_1\}$. Consequently, the different sets of constraints associated are : $\mathbb{C}_{|(Q_2,0)} = \{y_0 = R\}$ and $\mathbb{C}_{|(P,(1,1)} = \emptyset$*

Since *pconfs* inherit constraints of their predecessors, clean up procedure is required to remove useless constraints. An useless constraint contains at least a variable that is no longer relevant for any of its configurations. Erasing them directly could create a loss of information therefore, we compute the transitivity closure and suppress any unnecessary constraint afterward.

We introduce below the notion of *pcover* used to prune the search space, thereby ensuring termination of the algorithm Check_Simu. Informally speaking, if a *pconf* pc' is covered by a *pconf* pc it means that when computing the synchronized product, the space explored starting from pc' is a subset of the space explored starting from pc. Hence, we can prune the *pconf* pc' if pc has already been explored.

DEFINITION 13. ***(pcover)***
Let A_Q, A_P two $\mathcal{EL}_V$-description automata and let $pc = (S_Q, p, \mathbb{C})$ and $pc' = (S'_Q, p', \mathbb{C}')$ be two pconf. We say that pc' is covered by pc, and we note $pc' \lhd pc$, if and only if the followinf conditions hold:

(1) $q_p = q_{p'}$ *and* $\mathbb{C}_{|p} \subseteq \mathbb{C}'_{|p'}$
(2) there exists $Z \subseteq S'_Q \times S_Q$ *s.t*
  (a) $\forall q \in S_Q, \exists (q', q) \in Z$
  (b) $\forall (q', q) \in Z, q_{q'} = q_q$ *and* $\mathbb{C}_{|q} \subseteq \mathbb{C}'_{|q'}$
(3) $\mathbb{C}_{var} \cup \mathbb{C}_{free} \cup \mathbb{C}_{cte} \cup \mathbb{C}$ *is consistent where*
  $\mathbb{C}_{var} = \bigcup_{(q',q)\in Z} \{((x, I_{q_x}) = (y, I_{p_y}) | ((x, I_{q'_x}) = (y, I_{p'_y})) \in \mathbb{C}'_{|q'} \setminus \mathbb{C}_{|q}\}$,
  $\mathbb{C}_{free} = \bigcup_{(q',q)\in Z} \{((x, I_{q_x}) = free) | \forall (x, I_{q'_x})\ unconstrained\}$
  and
  $\mathbb{C}_{cte} = \bigcup_{(q',q)\in (Z\cup\{(p',p)\})} \{((x, I_{q_x}) = a) | ((x, I_{q'_x}) = a) \in \mathbb{C}'_{|q'} \setminus \mathbb{C}_{|q}\}$

Let $pconf\ pc = (S_Q, p, \mathbb{C})$ and $pc' = (S'_Q, p', \mathbb{C}')$ be two $pconf$. According to definition 13, pc' is covered by pc ($pc' \lhd pc$) if :

(1) States q_p and $q_{p'}$ of respectively p and p' represent the same states from A_P. By adopting such a criteria, a direct consequence is that any outgoing edge of p' has an equivalent one in p up to variables bounds.
  It remains to check information about bounds stored in constraint sets. It is required that $\mathbb{C}'$ must be more restrictive than $\mathbb{C}$. Indeed, if $x = a$ in pc' and x is free in pc then it is possible to create such a bound in pc. However, if x if free in pc' and $x = a$ in pc then pc may not mimic the path where $x = b$ in pc'.
(2) S_Q and S'_Q are sets of configuration and therefore they involve sets of states from A_Q.
  (a) An important condition is any the set of states involved in S_Q must be involved in S'_Q.
  (b) Each couple sharing the same state must then check the same condition as p and p'.
(3) Previous points focus on finding an equivalent configuration. It remains to check consistency of the pairings. Indeed, two distinct instances of a variables can be mapped into the same instance. It is then necessary that such constraints are consistent.

We are now ready to present our algorithm Check_Simu (c.f. algorithm 2). The Algorithm takes as input two $\mathcal{EL}_V$-description automata A_Q and A_P and returns true if $A_Q \ll_\exists A_P$ or false otherwise. The algorithm starts with the initial $pconf\ pc_0 = (\{(q_0, \vec{0})\}, (p_0, \vec{0}), \emptyset)$ made of the initial configuration $(q_0, \vec{0})$ of A_Q and the initial configuration $(p_0, \vec{0})$ of A_P and an empty set of constraints. Then it recursively explores a tree of generated $pconf$. For each $pc = (S_Q, p, \mathbb{C})$, the algorithm tries to check whether $S_Q \ll_\exists p$ under the constraints $\mathbb{C}$. To achieve this task, the algorithm will generate and explore the new $pconf$ that are the children of the $pconf\ pc$. We first consider each mapping from the outgoing transitions of the configurations in S_Q into the outgoing transitions of p. Let $\mathcal{M}_{S_Q \mapsto p}$ be the set of such mappings.

Each mapping $m \in \mathcal{M}_{S_Q \mapsto p}$ is made of a set of pairs $((c, x, c'), (p, y, s'))$ where (c, x, c') is an outgoing transition of $c \in S_Q$ mapped to (p, y, s') an outgoing transition of p. In this case, the transition (c, x, c') is called the source while the transition (p, y, s') is called the target. Let x be either a variable or a constant. We use the following notation: $t_x = x$ if x is a constant and $t_x = (x, I_{c_x})$ if x is a variable. Then, each element $((c, x, c'), (p, y, s'))$ in a mapping m is associated with a constraint $t_x = t_y$. We note by $\mathbb{C}_m$ the set of constraints associated with the elements of m.

Algorithm 2 *Check_Simu*

Input : A_Q, A_P ; Pconf : pc ; Pconf's historic : hist
Output : Result

Create a queue pGen
$Result \leftarrow true$
if $Check\_Cover(pc, hist) == false$ **then**
  $Result \leftarrow false$
  hist ← hist ∪ {pc}
  Compute the mappings $\mathcal{M}_{S_Q \mapsto p}$
  for each mapping $m \in \mathcal{M}_{S_Q \mapsto p}$ **do**
    for each assignment $\mathbb{C}_{Assign}$ w.r.t. m **do**
      if $\mathbb{C} \cup \mathbb{C}_m \cup \mathbb{C}_{Assign}$ is consistent **then**
        enqueue(m,$\mathbb{C}_{Assign}$) onto pGen
      end if
    end for
  end for
  while ¬ Result and $pGen$ is not empty **do**
    $Result \leftarrow true$
    Create a queue children
    $g \leftarrow dequeue(pGen)$
    for each target (p, y, s') of g **do**
      Create a $pconf$ pc'=$(S'_Q, p', \mathbb{C}')$
      $S'_Q \leftarrow \{c' | ((c, x, c'), (p, y, s')) \in g\}$
      $p' \leftarrow s'$
      $\mathbb{C}' \leftarrow \mathbb{C} \cup \mathbb{C}_m \cup \mathbb{C}_{assign}$.
      children.enqueue(pc')
    end for
    while *Result* and children is not empty **do**
      $Result \leftarrow Check\_Simu(A_Q, A_P, children.dequeue(), hist)$
    end while
  end while
end if
return *Result*

Algorithm 3 Check_Cover

Input : Pconf : pc ; Pconf's historic : hist
Output : Boolean

if ∃ pc' ∈ hist such that pc' ⊲ pc **then**
  return true
else
  return false
end if

A mapping $m \in \mathcal{M}_{S_Q \mapsto p}$, augmented with an assignment of values to free variables w.r.t. m and pc (i.e., variables that do not appear in $\mathbb{C} \cup \mathbb{C}_m$) leads to a candidate, called $pgenerator$. A $pgenerator$ is used to generate new $pconf$. An assignment $\mathbb{C}_{Assign}$ is a set of constraints of the form $x = a$, where x is a free variable w.r.t. m and pc. A $pgenerator$ g is a pair $g = (m, \mathbb{C}_{Assign})$ where m is a mapping and $\mathbb{C}_{Assign}$ is an assignment of free variables.

Given a $pgenerator$ $g = (m, \mathbb{C}_{Assign})$, we group together elements of m having the same target (p, y, s') (Algorithm 2, Lign 10). Each of such groups, leads to a set of new $pconf$s of the form: $pc' = (S'_Q, p', \mathbb{C}')$ with:

- $S'_Q = \{c' | ((c, x, c'), (p, y, s'))\}$
- $p' = s'$
- $\mathbb{C}' = \mathbb{C} \cup \mathbb{C}_m \cup \mathbb{C}_{assign}$.

The algorithm makes an exhaustive exploration of possible $pgenerators$. It returns true if at least one $pgenerator$ is evaluated successfully.

Checking whether a simulation exists takes the shape of an or/and tree as shown in Figure 7."Or" steps ($\vee$) focus on finding if one $pgenerator$ is successful (Algorithm 2 Lign 14). In order to be successful, all the children ($\wedge$) obtained by the mapping must be valid (Algorithm 2 Lign 25). This is where the recursive call will occur conducting to a new level for the tree and so on.

In order to not have an infinite run, we make full use of the pcover as a stop criteria. That is why, each $pconf$ will transfer to its children the list of all previous $pconf$ in the branch. We will then stop once, we find a couple such that the cover criteria is checked (Algorithm 3). Figure 7 dashed pconf and the initial one verifies such a condition.

Figure 7 illustrates the shape a run of the algorithm can takes over the running example. We presents it here under the shape of an or/and tree. It starts from the initial configuration to produces $pgenerators$ linked by a $\vee$ nodes. A $pgenerator$ can be validated if all its children are valid symbolized by the $\wedge$ nodes. It then stops when it either meets a leaf or when coverage is found.

Now the global idea has been exposed, we will explain why we consider sets of configuration of A_Q over a single configuration as we do for A_P. The reason is simple and illustrated in the second branch of Figure 7. Not doing so might duplicate choice possibilities ending in a false successful run. Indeed, if we focus on the second alternative of Figure 7, the transition mapping is such that x_0 simulates the two transitions labeled by R. P, the state reached by x_0 is refreshing for x_0 therefore x_0 is no more constrained. Then, if we don't fuse, we have two independent unconstrained couples $cp_1 = ((Q_2, (0)), (P, , (1, 1)))$ and $cp_2 = ((C, (0)), (P, (1, 1)))$. The problem lies in the fact that there exist a simulation for the two of them separately by taking $S = z_1$ for cp_1 and $R = z_1$ for cp_2. However, those two simulations are not consistent since for a same instance of a variable, we associate different values. Fusing them will only lead to check out if $(P, (1, 1))$ can simulate the two states by considering a unique choice for a variable instance.

6.2.1 Terminating. 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. It stops exploring a branch if one of the following occurs:

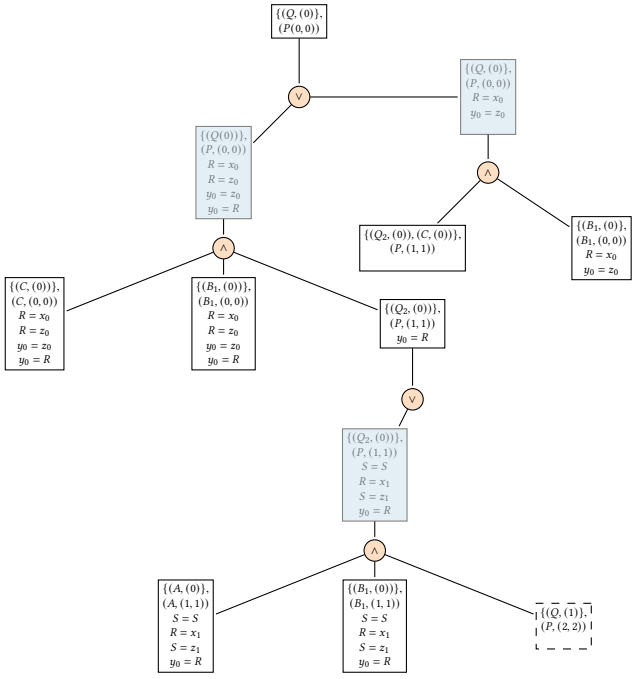

Figure 7: Or/And Tree

(1) A leaf has been reached.
(2) All $pgenerators$ failed to produces a valid simulation.
(3) A pconf is covered by a previous one.

The first case will not be discussed since by definition it will not produce infinite possibilities. Regarding the second case, it produces an infinite branch only if there are infinite possibilities. However, the number of $pgeneratprs$ is limited since we deal with a finite number of transitions for each automata. Regarding the part induces by assignments, it focuses on giving a value to unconstrained variables. Without loss of generality, we will restrict choices to labels of constant transition of A_Q and A_P. This set regroups relevant choices for variables since only a variable can mimic a choice out of this scope then it can be reduced to those constants. Consequently, only a finite number of $pgenerators$ will be produced thus validating that it will not produce an infinite branch.

The last point to check deals with a potential infinite depth. In our context, it means that there exists an infinite sequence of pconfiguration without pcover.

PROPERTY 1. *Let A_Q and A_P be two description automaton. 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. The idea of the proof is to show that there exists a $pconf$ that appears multiple times up to counters values. In an infinite sequence of $pconf$ there necessarily exists a configuration p of A_P associated to a state that appears an infinite number of times. Since, $\mathbb{C}_p$ possibilities can be enumerated. There are an infinite number of $pconf$s such that point 1 of the definition is verified. The same reasoning can be conducted over configurations of S_Q. Since $pconf$ are produced by the product of those two parts, we will have at least

a $pconf$ that appears an infinite number of times differential thanks to counters in the configuration. Point 3 is immediately checked because the proof based on equality over p, S_Q and $\mathbb{C}$ up to counter values. Therefore, the proof is complete since we showed that within any infinite sequence there are at least a configuration that covers another one. □

Therefore, thanks to Property 1 we can conclude that if the algorithm does not fail, it will necessarily halts.

It remains to show that the algorithm is correct, i.e. sound and complete.

6.2.2 Soundness. To show soundness, let consider the simulation tree $Exec_{A_Q,A_P}$ obtained after termination of a non-failing run. This tree is composed of $pconf$s. Note that thanks to the cover criteria we consider here a finite tree. The idea of the proof is to show that pcover induces simulation. As a consequence, $Exec_{A_Q,A_P}$ can be used to create σ, ϕ such that $T(A_Q, \sigma) \ll T(A_P, \phi)$.

In fact, when a $pconf\ pc'$ is covered by pc, any possibility of pc' has a counterpart in pc because pc contains a subsets of states compared to pc' and constraints inclusion allows to have a less constrained $pconf$. Constraints can then be easily transformed into substitution by considering constraints of the form $x = a$. In order to formally define this notion we introduces *universal simulation*. Unlike, existential simulation which looks for simulation w.r.t one substitution, this simulation aims to find such a relationship for any substitution.

DEFINITION 14. ***(Universal simulation)***
Let A_P and A_Q two description automata.
Left universal simulation is such that :

$$A_Q \ll_\forall A_P, \text{ if } \forall\phi, \exists\sigma \text{ such that } T(A_Q, \sigma) \ll T(A_P, \phi)$$

Right universal simulation is such that :

$$A_Q \ {}_\forall\!\ll A_P, \text{ if } \forall\sigma, \exists\phi \text{ such that } T(A_Q, \sigma) \ll T(A_P, \phi)$$

LEMMA 2. *Let A_Q, A_P two description automata. Let $pc = (S_Q, p, \mathbb{C})$ and $pc' = (S'_Q, p', \mathbb{C})$ be two $pconf$. If $pc' \lhd pc$ then $p' \ {}_\forall\!\ll p$ and $S_Q \ll_\forall S'_Q$*

PROOF. The proof will be separated in two steps. First, we will consider to show that states involved in pc are also involved in pc'. Then, we will show how to complete $\mathbb{C}$ in order to reach a state similar to $\mathbb{C}'$.

The first part is immediately verified due to the definition of coverage. By definition of Z, states involved in S_Q are included in S'_Q.On the other hand, p and p' shares the same states. Consequently, if $\mathbb{C}$ can be completed in order to be similar to $\mathbb{C}$. We can conclude that they share the same possibilities.

Constraint sets inclusion means that they at least share the same base and that, we only need to add constraints to check compatibility of choices. The last point of pcover aims to check if this extension is possible. Indeed, for each couple, we will construct equivalent constraints for each missing constraints. Since a same configuration of S_Q can be bound to more than one configuration of S'_Q. It remains to check that this extension is valid. Three kinds of constraints are added to complete $\mathbb{C}$. Constraints involving two variables ($\mathbb{C}_{var}$), constraints involving a constant and a variable ($\mathbb{C}_{cte}$) and finally special constraints for free variables ($\mathbb{C}_{free}$). The last one avoids case where a free instance and a bound instance are related to the same variable. This case is not valid since we do not know what value the free variable may need to take.

We then have augmented the $\mathbb{C}$ to mimic $\mathbb{C}'$. By definition of pcover, this set if consistent, we can then transform it into σ and $\mathbb{C}'$ into σ' by considering only constraints of the form : $x = a$. As a consequence, for any choice of σ' there exists a similar choice in σ.

Since p and p' shares the same state under the same constraints, we can deduce that $p' \ {}_\forall\!\ll p$ (and $p \ {}_\forall\!\ll p'$). On the other hand, since S'_Q may contain more states than S'_Q, we have $S_Q \ll_\forall S'_Q$ which concludes this proof.

□

Lemma 2 allows to certify soundness. Indeed, based on $Exec_{A_Q,A_P}$ we can create $T(A_Q, \phi)$ and $T(A_P, \sigma)$. By transforming constraints into functions, considering only configurations S_Q for $T(A_Q, \phi)$ and p for $T(A_P, \sigma)$, one can create valid trees. Note that S_Q being a set, in order to have a valid tree it must be split into different nodes.

By construction, we know that the partial tree of A_Q made of $Exec_{A_Q,A_P}$ is simulated by the partial tree of A_P made of $Exec_{A_Q,A_P}$ By Lemma 2, we know that any choices made before can be reproduced. Moreover, the algorithm fails if all the possibilities of pc' conducts to a fail the algorithm will find it and return false anyway. Since we are considering a non-failing run, it means that a run succeed up to cover. The valid possibility can be mimicked in order to complete σ and ϕ. Consequently, we can complete σ and ϕ such that $T(A_Q, \sigma) \ll T(A_Q, \phi)$. Therefore, we have $A_Q \ll_\exists A_P$.

We conclude that the algorithm is sound. We are now left with completeness in order to prove this algorithm's correctness.

6.2.3 Completeness. Regarding completeness, we can use a given solution (σ_0, ϕ_0) to guide the non-deterministic rules in order to generate a non-failing run. Indeed, each time a choice has to be made, we can compare it to the corresponding one made in $T(A_P, \sigma_0)$ for P and $T(A_Q, \phi_0)$ for Q. Since by definition, the simulation exists the algorithm will keep going until the exact same solution is encountered or if a shorter solution starting similarly is found. Failing would mean that there are no valid possibility which contradicts the assumption of (σ_0, ϕ_0) solution.

It is worth to enlighten that the proposed algorithm is constructive in the sense that if the answer is true, the algorithm can be easily modified to exhibit a simulation relationship between its inputs.

THEOREM 2. *Let A_Q and A_P two description automaton, it is decidable whether $A_Q \ll_\exists A_P$*

By the use of theorem 1, the immediate result about weak-subsumption is the following.

COROLLARY 1. *Let P be a pattern and C a ground description. $P \sqsubseteq_{\mathcal{T},gfp} Q$ is decidable.*

6.3 Complexity Analysis

This part will discuss the complexity of weak-subsumption. The upper bound will be achieved by calculating the space explored by the algorithm. Then the equivalence between existential simulation and weak-subsumption will allow to finalize the analysis.

PROPERTY 2. *Let P,Q be two patterns and let A_P, A_Q be their respective $\mathcal{EL}_V$-description automata. Deciding whether $A_P \ll_\exists A_Q$ is* EXPTIME*-complete.*

The proof of the upper bound consists in showing that only an exponential number of pconfigurations are required to decide weak subsumption. We define :

- n: the number of nodes,
- v: the number of variables and
- D: the number of values a variable can take.

In order to estimate the number of pconfigurations, we need to estimate how many different configurations can be visited up to cover. Let's consider one by one each element of the pconfiguration. In any pconfiguration we have configuration which are related to a subset of states of A_Q, a configuration related to one state of A_P and constraints $\mathbb{C}$ on these configurations.

Constraints and configuration issued from p give $n * (D + v)^v$ different possibilities. Indeed, the n states of p are different up to constraints. Constraints where x is bound to one element which is easier a constant (D) or a variable (v).

Regarding, S_Q it is slightly different. In fact, we will always deal with a subset of states of A_Q which induces 2^n. A state and a mapping produces different possibilities. Variables may have multiple instances at the same time during a run. Therefore, a states. can appear with different mapping but at most $(D + v)^v$ different ones. Then we have at most n*$(D + v)^v$ different possibilities since we must consider this for each state. Finally, combining all of this gives the following complexity $2^n * n^2 * (D + v)^{2*v}$. Once simplified we have $\exp^{n*\log(2)+2(\log n)+v\log(D+v))}$

The EXPTIME-hardness of checking existential simulation between $\mathcal{EL}_V$-description automata is obtained by a reduction from the existence of infinite execution of an alternating Turing machine working on a space polynomially bounded by the size of the input.

7 CONCLUSION

This paper investigates the problem of reasoning with description logics augmented with variables. It considers a framework that caters for cyclic terminologies and defines two semantics of variables which differ w.r.t. to the possibility or not to refresh the variables. As preliminary results, the paper investigates a new reasoning mechanism, called weak-subsmption, in the context of the description logic $\mathcal{EL}_V$, obtained from an extension of the logic $\mathcal{EL}$ with refreshing variables. Future research work will be devoted to the extension of the approach in three research directions: *(i)* considering additionnal reasoning mechanism in this context that go beyond weak-subsumption, *(ii)* dealing with concept-description variables, and *(iii)* considering other description logics such as the logic $\mathcal{FL}_0$ and $\mathcal{ALN}$.

REFERENCES

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

[2] Franz Baader. 2003. Terminological cycles in a description logic with existential restrictions. In *IJCAI*, Vol. 3. 325–330.

[3] Franz Baader, Stefan Borgwardt, and Barbara Morawska. 2012. Extending unification in EL towards general TBoxes. In *Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*.

[4] Franz Baader, Diego Calvanese, Deborah McGuinness, Peter Patel-Schneider, Daniele Nardi, et al. 2003. *The description logic handbook: Theory, implementation and applications*. Cambridge university press.

[5] Franz Baader, Oliver Fernández Gil, and Pavlos Marantidis. 2018. Matching in the Description Logic $\mathcal{FL}_0$ with respect to General TBoxes. In *LPAR*. 76–94.

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

[7] Franz Baader, Ralf Küsters, and Ralf Molitor. 1998. Structural subsumption considered from an automata-theoretic point of view. In *Proceedings of the 1998 International Workshop on Description Logics (DL'98*. Citeseer.

[8] Franz Baader and Barbara Morawska. 2009. Unification in the Description Logic $\mathcal{EL}$. In *International Conference on Rewriting Techniques and Applications*. Springer, 350–364.

[9] Franz Baader and Barbara Morawska. 2014. Matching with Respect to General Concept Inclusions in the Description Logic $\mathcal{EL}$. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*. Springer, 135–146.

[10] Franz Baader and Barbara Morawska. 2014. Matching with Respect to General Concept Inclusions in the Description Logic $\mathcal{EL}$. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*. Springer, 135–146.

[11] Franz Baader and Paliath Narendran. 2001. Unification of concept terms in description logics. *Journal of Symbolic Computation* 31, 3 (2001), 277–305.

[12] Alexander Borgida, Ronald Brachman, Deborah Mcguinness, and Lori Resnick. 1996. CLASSIC: a structural data model for objects. *ACM SIGMOD Record* 18 (02 1996). https://doi.org/10.1145/67544.66932

[13] Alex Borgida and Ralf Küsters. 1999. "What's not in a name?" - Initial Explorations of a Structural Approach to Integrating Large Concept Knowledge-Bases.

[14] Ronald J Brachman, Deborah L McGuinness, Peter F Patel-Schneider, Lori Alperin Resnick, and Alexander Borgida. 1991. Living with CLASSIC: When and how to use a KL-ONE-like language. In *Principles of semantic networks*. Elsevier, 401–456.

[15] Francesco Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. 1999. Reasoning in Description Logics. *Center for the Study of Language and Information* (06 1999).

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

[17] Robert M MacGregor. 1991. Inside the LOOM description classifier. *ACM Sigart Bulletin* 2, 3 (1991), 88–92.

[18] Manfred Schmidt-Schauß and Gert Smolka. 1991. Attributive concept descriptions with complements. *Artificial intelligence* 48, 1 (1991), 1–26.