Weak subsumption in the EL-description logic with refreshing variables

Théo Ducros, Marinette Bouet, Farouk Toumani

Weak subsumption in the $\mathcal{EL}$-description logic with refreshing variables

Théo Ducros, Marinette Bouet, and Farouk Toumani

LIMOS, 1 rue de la Chébarde, Aubière 63178 France firstname.name@limos.fr

Abstract. 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*, in presence of terminological cycles. Weak subsumption can be viewed as a generalization of matching and unification in presence of refreshing variables. We show that weak subsumption w.r.t. greatest fix-point semantic is EXPTIME-complete. Our main technical results are derived by establishing a correspondence between this logic and a specific form of variable automata.

Keywords: Weak subsumption · cyclic TBox · refreshing variables · description automaton · existential simulation.

1 Introduction

Concepts with variables (also called patterns) have been introduced in description logics since the mid-nineties [18, 1] and led to a highly interesting research stream on the so-called non-standard reasoning, specifically matching [8, 5] and unification [4]. As an example, consider the following pattern P_1 defined as a $Person$ with a certain relationship with a $University$.

$$P_1 \equiv Person \sqcap \exists x.University$$

Here, the variable x takes its values from a set of possible atomic role names. Note that, we restrict our attention on role variables while the techniques proposed in this paper can be extended to handle concept variables. The concept description

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

matches the pattern P_1. Indeed, if we replace the variable x by the role $worksIn$, the pattern P_1 becomes equivalent to the concept $Academic$. 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. Matching in $\mathcal{FL}_0$ has been shown polynomial [1] while considering general TBox induces a blow up in complexity leading to EXPTIME [5]. On the other hand, matching in $\mathcal{EL}$ in presence or not of TBoxes is NP-complete in both cases. Unification is NP-complete for $\mathcal{EL}$ in presence of cycle restricted TBoxes while the general case remains open [4].

Consider now the case of a cyclic description of a pattern $SAcademic$:

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

Such a pattern describes $SAcademic$ as persons that have (unknown) relationships with universities and with $SAcademics$. Consider now the following concept descriptions:

$$Doctor = Person \sqcap \exists PhDfrom.University \sqcap \exists formerly.PhDStudent$$

$$PhDStudent = Person \sqcap \exists studyIn.University \sqcap \exists supervisedBy.Doctor$$

Using standard semantics of variable substitution, $Doctor$ does not match the pattern $SAcademic$ (i.e., the corresponding matching problem is unsolvable). However, the situation changes if variables x and y obtain the ability to have a local scope which allows to bound them to potentially different values each time $SAcademic$ is unfolded. In other words, each unfolding would lead to specific independent instances of variables x and y as illustrated below:

$$SAcademic \equiv Person \sqcap \exists x_1.University \sqcap \exists y_1.(Person \sqcap \exists x_2.University \sqcap \exists y_2.(\ldots))$$

This feature leads to a new semantics of variable substitutions which enables to *refresh* the values of the variables x and y in each description of the pattern $SAcademic$. Hence, by alternating variable substitutions between $\{x \mapsto PhDfrom, y \mapsto formerly\}$ and $\{x \mapsto studyIn, y \mapsto supervisedBy\}$, it is now possible to compute a matcher that makes the pattern $SAcademic$ equivalent to the concept $Doctor$. Such variables, called refreshing variables, are inspired from the framework of variable automata [11].

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: *(i)* our definition of concept patterns uses role variables while the literature focuses on concept variables, *(ii)* we support cyclic patterns 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 specific form of variable automata.

The paper is organized as follows. Section 2 deals with related works before presenting technical notions and notations needed in this paper in Section 3. Section 4 is devoted to the presentation of the description logic $\mathcal{EL}_V$, an extension of $\mathcal{EL}$ to handle refreshing variables. Section 5 presents our main technical results regarding the problem of testing weak subsumption between $\mathcal{EL}_V$ concepts. These results are obtained by a reduction of the weak subsumption problem to a simulation problem between $\mathcal{EL}_V$-description automata. We conclude and draw future research direction in section 6.

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 [19]. The first approach led to interesting results on how close description logics can be to automata [2]. Indeed, in $\mathcal{FL}_0$ subsumption can be reduced in language inclusion of finite automata [6] while $\mathcal{EL}$ subsumption will be equivalent to simulation [3].

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 [1]. 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 [1] 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[8]. 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 [10]. 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 [4]. 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 [8].

So far matching and unification have proven to be useful in order to filter out important aspects of large concepts in classic [13]. Baader et al. [10] 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 [12].

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.

3 Preliminaries

3.1 Trees

We use the following definition of a tree [16]: A tree is a set $\tau \subseteq \mathbb{N}^*$ such that if $in \in \tau$, for $i \in \mathbb{N}^*$ and $n \in \mathbb{N}$, then $i \in \tau$ and $im \in \tau$ 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 i, the nodes of the form in, for $n \in \mathbb{N}$, are children of i. 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, $\lambda : \tau \rightarrow S$ is a node labeling function that maps each node of τ to an element in S, and $\delta : \tau \times \tau \rightarrow M$ is an edge labeling function that maps each edge (i, in) of τ to an element in M. Note that, $\delta(i, in) = r$ can be rewritten (i, r, in).

We define now simulation which is a binary relationship between two trees as follows.

Definition 1. ***(simulation between trees)***
Let $(\tau_1, \lambda_1, \delta_1)$ and $(\tau_2, \lambda_2, \delta_2)$ be respectively two trees. 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 $(c_2, r, c_2') \in \delta_2$*

If such a relation exists, it is noted $(\tau_1, \lambda_1, \delta_1) \ll (\tau_2, \lambda_2, \delta_2)$. We extend simulation to nodes and say that a node c_1 is simulated by a node c_2 when $(c_1, c_2) \in Z$.

3.2 Basics of the description logic $\mathcal{EL}$

The description logic $\mathcal{EL}$ is based on three constructors : *top concept* ($\top$), *conjunction* ($\sqcap$) and *existential restriction* ($\exists R.C$). 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 $\mathcal{EL}$ are 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.

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

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 $\mathcal{T}$.

Normal forms play a key role in description logics since they facilitate reasoning. Indeed, comparing two concepts is easier if they bear the same structure. For $\mathcal{EL}$, an $\mathcal{EL}$-concept-description C is in normal form if C is of the form :

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

where $A_i \in N_A$, $r_i \in N_R$ and $B_i \in N_{def}$.

Note that an $\mathcal{EL}$-TBox is said normalized if all its definitions are normalized. The normalization process consists in simply creating concept-definitions and add them to the TBox. By replacing definitions by their name in others definition, the TBox will reach a normalized state.

An interpretation $\mathcal{I}$ is a *model* of $\mathcal{T}$ if and only if for all relationships of $\mathcal{T}$ of the form $A \equiv C$, $A^{\mathcal{I}} \equiv C^{\mathcal{I}}$. Subsumption, the relationship that determines whether a concept is more general than another one, is commonly proposed as an inference task at a terminological level (i.e. w.r.t $\mathcal{T}$). C is subsumed by D ($C \sqsubseteq_{\mathcal{T}} D$) w.r.t $\mathcal{T}$ if and only if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for all model $\mathcal{I}$ of $\mathcal{T}$. Deciding subsumption relationship is proven to be polynomial in $\mathcal{EL}$. Equivalence is a stronger relationship based on subsumption where a pair of concept subsumed each other noted $\{C \equiv D\} \equiv \{C \sqsubseteq D; D \sqsubseteq C\}$.

In the scope of terminologies cyclic concept can be defined. Intuitively, A *cyclic concept* is a concept name A which refers to itself directly (i.e. within its own description) or indirectly (i.e. in the definition of a concept name involved in A). By opposition, an *acyclic concept* refers to a concept that is not cyclic. A *cyclic $\mathcal{EL}$-terminology* is an $\mathcal{EL}$-terminology that contains at least a cycle while an acyclic

$\mathcal{EL}$-terminology don't contain any cycle. This notion is formally presented in [3] where terminological cycles are defined as cycle within an $\mathcal{EL}$-description graph representing an $\mathcal{EL}$-terminology.

4 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 role terms instead of only role names.

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 and variables in different descriptions are different up to renaming.

Example 1. This example presents the normalized TBox that will be used as a reference in this paper. Regarding the variables, z and y are refreshing while x belongs to the set of non-refreshing variables.

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

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., $\sigma_0, \sigma_1, \ldots$) to denote the fact that a substitution σ maps a refreshing variable z to several values. Assume a substitution σ that maps the non-refreshing variable x to a role name R (i.e., $\sigma_i(x) = R, \forall i \in \mathbb{N}$) and it maps the first occurrence of the refreshing variable z to R (i.e., $\sigma_0(z) = R$) while it maps the second occurrence of z to S (i.e., $\sigma_1(z) = S$). This leads to the following $\mathcal{EL}$-description:

$$\sigma_0(P) \equiv \exists R.C \sqcap \exists R.B_1 \sqcap \exists S.A_1 \sqcap \exists R.\underbrace{(\exists S.C \sqcap \exists S.B_1 \sqcap \exists S.A_1 \sqcap \exists R.\sigma_2(P)))}_{\sigma_1(P)}$$

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 2. ***(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 variables have a local scope in the sense 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 z by a new non-refreshing variable. Note that, in the case of an $\mathcal{EL}_V$-pattern P with a cyclic definition that uses refreshing variables, the unfolding of P leads to an $\mathcal{EL}_V$-pattern $u(P)$ with an infinite size description.

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 $(\tau_P, \lambda_P, \delta_P)$ which is recursively defined as follows:

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

Example 2. Figure 1 depicts the tree representation of $u(P)$. Each node is labeled by the corresponding concept in the TBox. Leaves are labeled by the keyword *atomic*. Note that, since P is a cyclic pattern, the tree associated with $u(P)$ is infinite since it includes an infinite branch: $P.x.P.x.P.x.P \ldots$ where each occurrence of the nodes labeled by P has outgoing edges labeled by the variables z_i, with $i \in [0, +\infty[$.

Instantiation of $\mathcal{EL}_V$-concepts is defined using the notion of variable *substitution*. 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 $T \in N_C \cup N_R$;

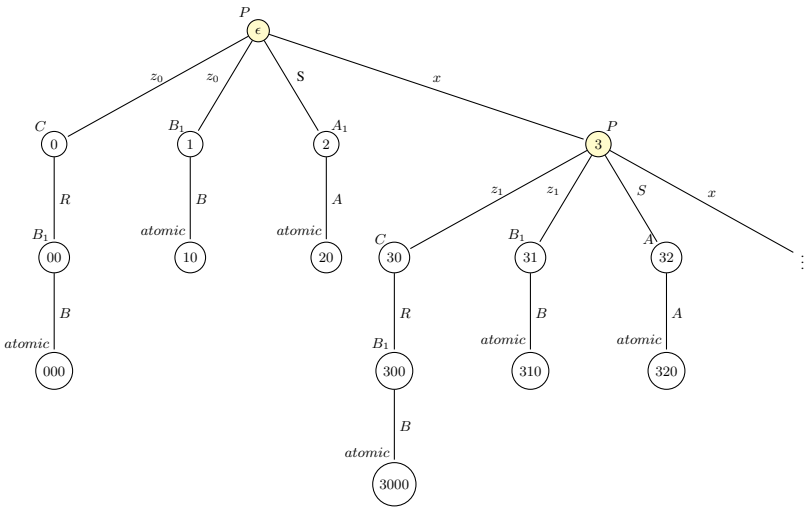


Fig. 1: Tree corresponding to $u(P)$

- $\sigma(C \sqcap D) = \sigma(C) \sqcap \sigma(D)$ 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 P.

Definition 3. ***(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 notations and we write $\sigma(P)$ instead of $\sigma(u(P))$ for a pattern instance of P w.r.t. σ.

Example 3. Continuing with the previous example, by considering a substitution σ that maps the non refreshing variable x to $\sigma(x) = R$ and that maps the refreshing variable z alternatively to R and S we obtain the following instance of the $\mathcal{EL}_V$-concept P:

$$\sigma(P) \equiv \exists \underbrace{R}_{\sigma(z)}.C \sqcap \exists \underbrace{R}_{\sigma(z)}.B_1 \sqcap \exists S.A_1 \sqcap \exists \underbrace{R}_{\sigma(x)}.(\exists \underbrace{S}_{\sigma(z)}.C \sqcap \exists \underbrace{S}_{\sigma(z)}.B_1 \sqcap \exists S.A_1 \sqcap \exists \underbrace{R}_{\sigma(x)}.\sigma(P)))$$

Note that this instance is only achievable because z is a refreshing variable.

It is worth noting that an instance $\sigma(P)$ can be viewed as a tree which is obtained from the tree corresponding the unfolded $\mathcal{EL}_V$ concept $u(P)$ by replacing the variables with their respective values. More precisely, let $(\tau_P, \lambda_P, \delta_P)$ be a $\langle N_{def} \cup \{atomic\}, N_R \cup N_{V_N} \rangle$-labeled tree corresponding to the unfolded $\mathcal{EL}_V$ concept $u(P)$. Then an instance $\sigma(P)$ is a $\langle N_{def} \cup \{atomic\}, N_R \rangle$-labeled tree $(\tau_{\sigma(P)}, \lambda_{\sigma(P)}, \delta_{\sigma(P)})$ defined as follows:

- $\tau_{\sigma(P)} = \tau_P$,
- $\lambda_{\sigma(P)}(i) = \lambda_P(i), \forall i \in \tau_{\sigma(P)}$, and
- $\delta_{\sigma(P)}(i, in) = \sigma(\delta_P(i, j)), \forall i, in \in \tau_{\sigma(P)}$.

The next lemma proposes a characterization of subsumption w.r.t. greatest fix point semantics between $\mathcal{EL}_V$ concept instances using the simulation relation between their corresponding trees.

Lemma 1. *Let P, Q be two $\mathcal{EL}_V$-patterns in an $\mathcal{EL}_V$-TBox $\mathcal{T}$ and let σ, ϕ be two substitutions. Let $(\tau_{\sigma(P)}, \lambda_{\sigma(P)}, \delta_{\sigma(P)})$ and $(\tau_{\phi(Q)}, \lambda_{\phi(Q)}, \delta_{\phi(Q)})$ be the trees corresponding respectively to the instances $\sigma(P)$ and $\phi(Q)$. Then, we have: $\sigma(P) \sqsubseteq_{\mathcal{T},gfp} \phi(Q)$ if and only if $(\tau_{\phi(Q)}, \lambda_{\phi(Q)}, \delta_{\phi(Q)}) \ll (\tau_{\sigma(P)}, \lambda_{\sigma(P)}, \delta_{\sigma(P)})$*

The proof of this lemma is derived from the characterization of subsumption w.r.t. greatest fix point semantics in $\mathcal{EL}$ cyclic TBoxes by means of simulation between the so-called description graphs [3].

Indeed, the framework proposed in [3] extends naturally to infinite $\mathcal{EL}$ concepts. Moreover, simple syntactic transformations enable to turn description graphs of [3] into trees in the form used in this paper while preserving the simulation relationship.

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

Definition 4. ***(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 $P \sqsubseteq\!\!\!\approx_{\mathcal{T},gfp} Q$, iff there exists two substitutions ϕ_1 and ϕ_2 s.t. $\phi_1(P) \sqsubseteq_{\mathcal{T},gfp} \phi_2(Q)$

Note that weak subsumption can be viewed as an extension of respectively matching [8] when either P or Q is a ground $\mathcal{EL}$-concept, and unification [7] when both P and Q are $\mathcal{EL}_V$-patterns, to logic with refreshing variables.

5 $\mathcal{EL}_V$-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 2, we associate to each $\mathcal{EL}_V$-pattern P an $\mathcal{EL}_V$-description automata A_P such that there is a strong correspondence 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.

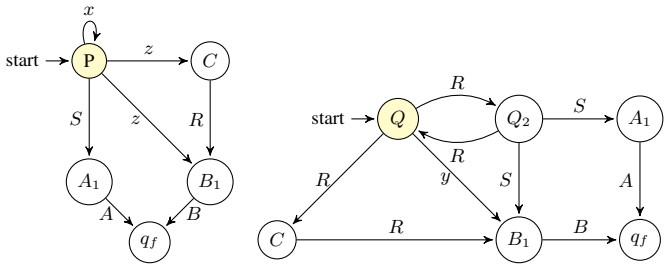
Definition 5. ***($\mathcal{EL}_V$-description automata)***

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 follows:

- $\mathcal{L} \subseteq N_A \cup N_R$ *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}*)*
- $\kappa : \mathcal{V}ar \rightarrow \mathcal{Q}$ *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 5 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 $\mathcal{T}$ an $\mathcal{EL}_V$-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., $\kappa(r_i) = \{P\}$).

Example 4. Figure 2 depicts the description automata A_P and A_Q, respectively, of the $\mathcal{EL}_V$ concepts P and Q of our example. Variable x is non-refreshing while variables z and y are refreshing. Their respecting refreshing states are given by: $\kappa(z) = \{P\}$ (i.e. P is the refreshing state of z in A_P) and $\kappa'(y) = \{Q\}$ (i.e., Q is the refreshing state of y in A_Q).



(a) A_P (b) A_Q

Fig. 2: Description automata for P and Q

An $\mathcal{EL}_V$-description automaton A_P of a given $\mathcal{EL}_V$ concept P is in fact a compacted representation of all the possible instances of P. In other words, the automaton A_P recognizes exactly the trees, hereafter called configuration trees, that are instances of P. We explain below informally the notion configuration trees (see formal definitions in our extended version), then we show that they are *equivalent* to the instances of P.

An $\mathcal{EL}_V$-description automaton A_P runs on *extended variable substitutions* which are defined as follows. Let $\mathcal{V}ar$ be the set of the variables used by an $\mathcal{EL}_V$-description

automaton A_P. An extended substitution $\bar{\sigma}$ is a mapping $\bar{\sigma} : \mathcal{V}ar \times \mathbb{N} \mapsto N_R$ that associates to each pairs $(x, i) \in \mathcal{V}ar \times \mathbb{N}$ a unique value from N_R. A pair (x, i) denotes the copy i of the variable x. Hence, an extended substitution assigns values to (potentially infinite) copies of variables.

Given an extended substitution $\bar{\sigma}$, a run of an $\mathcal{EL}_V$-description automaton A_P over $\bar{\sigma}$ is a $\langle S, \mathcal{L}\rangle$-labeled tree, noted $T(A_P, \bar{\sigma})$ and called a configuration tree of A_P. S is the set of configurations which is used to label the nodes of $T(A_P, \bar{\sigma})$. Informally, a configuration in $T(A_P, \bar{\sigma})$ fixes the copies of the refreshed variables that are used at a given step 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 (i.e., multiple copies of refreshed 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.

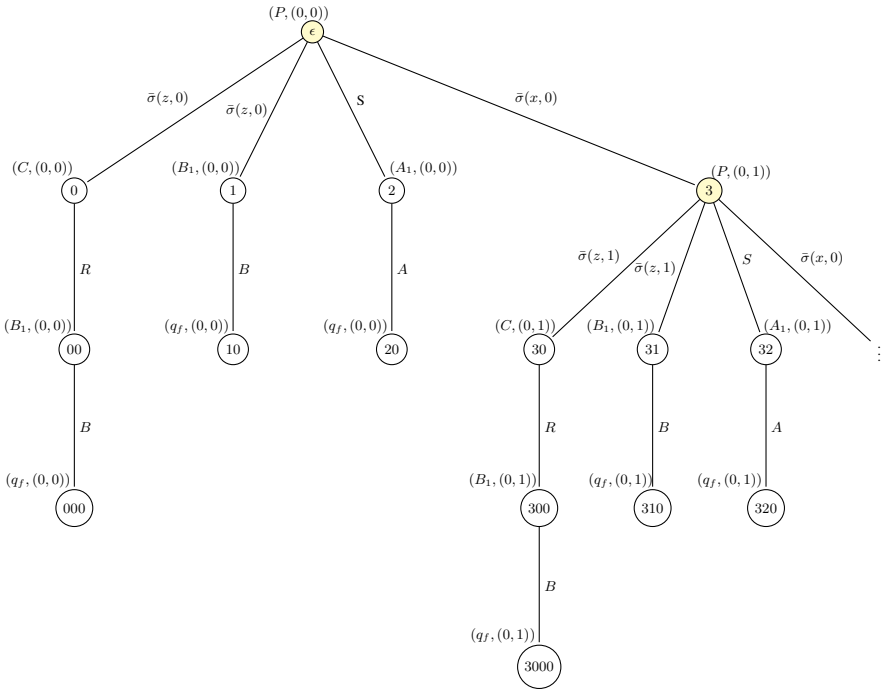


Fig. 3: The configuration tree $T(A_P, \bar{\sigma})$

Definition 6. ***(Configuration and configuration tree)*** *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.*

- *A configuration is a pair (q, I) where $q \in \mathcal{Q}$ is a state of A_P and $I \in \mathbb{N}^{|\mathcal{V}ar|}$ is a vector of integers.*
- *Let $S \subseteq \mathcal{Q} \times \mathbb{N}^{|\mathcal{V}ar|}$. 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.*

Algorithm 1 GenConfTree

Input : $A_P = (\mathcal{Q}, \mathcal{L}, \mathcal{V}ar, q_0, \delta, q_f, \kappa), \sigma$
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

Example 5. Figure 3 shows a configuration tree $T(A_P, \bar{\sigma})$ of the $\mathcal{EL}_V$-description automaton A_P over an extended substitution $\bar{\sigma}$. Nodes of the tree are labeled by configu-

rations. Since the description of P uses two variables (i.e., x and z), configurations of A_P will be made of pairs (q_c, I_c) where q_c is a state of A_P and $I_c = (i1, i2)$ is a vector of integer made of two components $i1$ (respectively, $i2$) used to record which copy of the variable x (respectively, z) is used at a given step of the execution of A_P. The root ϵ of the configuration tree is labeled with the configuration $(P, (0, 0))$ indicating that initially the automaton is at its initial state P and uses the copy 0 of each of the variables x and z. Four possibilities of moves are possible from this initial configuration: a non deterministic move to the configurations $(C, (0, 0))$ or $(B_1, (0, 0))$ on the transition labelled $\bar{\sigma}(z, 0)$ (i.e., the value assigned by $\bar{\sigma}$ to the copy 0 of z) or a move to the configuration $(A_1, (0, 0))$ on the transition labeled S or a move to the configuration $(P, (0, 1))$ using the transition labeled $\bar{\sigma}(x, 0)$. Note that since P is a refreshing state for the variable z, each time the state P is visited during the execution of A_P, the counter associated to the variable z (i.e., the second component of the vector I) is incremented. As explained in our extended version, in addition to local counters associated with configurations, a global counter is used to keep track of the last indices used for each variable. The global counter enables to avoid conflicts when generating new copies for refreshing variables.

Lemma 2 establishes a strong correspondence between instances of an $\mathcal{EL}_V$ concept P and the configuration trees of its corresponding $\mathcal{EL}_V$-description automaton A_P.

Lemma 2. *Let P be an $\mathcal{EL}_V$ concept of a terminology $\mathcal{T}$ and let A_P be its corresponding description automaton. Then, we have:*

- *$\forall \phi$ such that $\phi(P) = (\tau_{\phi(P)}, \lambda_{\phi(P)}, \delta_{\phi(P)})$ is an instance of $P \Rightarrow \exists \bar{\sigma}$ such that $T(A_P, \bar{\sigma}) = (\tau', \lambda', \delta')$ is a configuration tree of A_P with $\tau_{\phi(P)} = \tau'$ and $\delta_{\phi(P)} = \delta'$*
- *$\forall \sigma$ such that $T(A_P, \sigma) = (\tau', \lambda', \delta')$ is a configuration tree of $A_P \Rightarrow \exists \phi$ such that $\phi(P) = (\tau_{\phi(P)}, \lambda_{\phi(P)}, \delta_{\phi(P)})$ is an instance of P with $\tau_{\phi(P)} = \tau'$ and $\delta_{\phi(P)} = \delta'$*

Proof. The general idea is to demonstrate that since those two trees are constructed with the same lexicographic order they have the same structure. Regarding δ and δ', we will explain how to construct σ by using a bijection f such that $\phi(x) = \sigma(f(x))$. Let consider $\phi(P) = (\tau, \lambda, \delta)$ and $T(A_\sigma, P) = (\tau', \lambda', \delta')$ created using the same lexical order.

Let's consider a pair (ϵ, ϵ') where ϵ is the root in τ and ϵ' in τ'. By definition of the root, we have $\epsilon = \epsilon'$. Regarding their label we have that $\lambda(\epsilon) = \{P\}$ and $\lambda'(\epsilon') = \{P, \vec{0}\}$. $\lambda(\epsilon)$ refers to the concept P represented by the state P of A_P involved in $\lambda'(\epsilon') = \{P, \vec{0}\}$. By construction, they share the same structure. It remains to show that their edges are identically labeled. Consequently, we will consider the pairs (i, i') with such that $i = i'$ and $i \in \tau, i' \in \tau'$. We will now explain how to construct σ such that their edge labels are identical.

Constant edges will find automatically a counterpart since the same order has been applied $\delta(\epsilon, i) = \delta'(\epsilon', i') = r$.

Regarding variable edges, we have $\delta(\epsilon, i) = \phi(x)$ and $\delta'(\epsilon', i') = \sigma(x', 0)$. The lexicographic order combined to P assure $\sigma(x', 0)$ is necessarily associated to $\phi(x)$

that We can construct f such that $f(x) = (x', I_{x'_P} = 0)$. Note that if $x = x'$, it is a non-refreshing variable which will conserve the same value as it should since its counter will not increase.

Therefore, we have can construct $\sigma(x', 0) = \sigma(f(x) = \phi(x)$ for each variable transition.

The last step consists in showing that f can then be recursively defined even if variables are refreshed. In description logic a variable is refreshed if its associated pattern Q is unfolded. In the tree shape, it is traduced by $\lambda(i) = Q$. For description automaton, variables are refreshed if they belong to $\kappa^{-1}(Q)$ in other words if, Q appears in $\lambda(i')$ which is verified here. Consequently each pair (i, i'), will refresh the same variable. Consequently the same procedure can be used to complete f and thus σ. The same reasoning can be recursively applied on each pair for edge which will conduct to $\tau = \tau'$ and $\delta = \delta'$ and conclude the proof.

The same proof can be used by taking f^{-1} instead of f which is possible since f is bijective.

As explained in the next section, based on this lemma, we are now able to reduce the problem of testing weak subsumption between $\mathcal{EL}_V$ concepts into a simulation test over their corresponding $\mathcal{EL}_V$-description automata. Complexity is achieved by solving the simulation test thanks to a constructure algorithm.

5.1 Characterizing weak subsumption using description automaton

We first introduce the notion of existential simulation between $\mathcal{EL}_V$-description automata. This definition of simulation expresses on whether there exist substitutions such that resulting trees comply a successful simulation relationship.

Definition 7. ***(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 first 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 automata.

Theorem 1. *Let P and Q be two $\mathcal{EL}_V$ concepts of a terminology $\mathcal{T}$. Then, we have: $P \sqsubseteq\!\!\!\sim_{\mathcal{T},gfp} Q$ iff $A_Q \ll_\exists A_P$*

Proof. 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 2 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 $P \sqsubseteq\!\!\!\sim Q$.

This theorem reduces weak subsumption test between two $\mathcal{EL}_V$ concepts P and Q to an existential simulation test between the corresponding $\mathcal{EL}_V$-description automata A_P and A_Q.

5.2 Solving existential simulation between description automata

We propose an algorithm, called Check_Simu, to test existential simulation between $\mathcal{EL}_V$-description automata, 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 8. ***(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 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 now formally define the notion of $pconf$ which are used in the solving process by the algorithm.

Definition 9. ***($pconf$)***
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 10. ***(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 11. ***(Constraint set inclusion)***
Let $\mathbb{C}_1$ and $\mathbb{C}_2$ be two sets of constraints. The set $\mathbb{C}_1$ is included in $\mathbb{C}_2$, noted $\mathbb{C}_1 \subseteq \mathbb{C}_2$ if and only if

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 6. 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. A 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 12. ***(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 pconfs. We say that pc' is covered by pc, and we note $pc' \lhd pc$, if and only if the following 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 12, 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 that 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 $S_Q \ll_\exists p$ -i.e. checking if a same substitution verifies the simulation relationship between each element of S_Q and 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

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 $\leftarrow$ hist $\cup$ {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 $\neg$ 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 $\exists$ pc' $\in$ hist such that pc' $\lhd$ pc **then**
  return true
else
  return false
end if

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.

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 4."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 4 dashed $pconf$ and the initial one verifies such a condition.

Figure 4 presents the global shape of a partial execution of the algorithm. In order to do not overload the figure, only two possibilities are represented. Moreover, simulation problem concerning the same state in the two automata are not detailed since they are obviously successful.

Each white box represents a $pgenerator$ with its elements identified thanks to the labels of the trees instead of elements of τ and τ'. Labels have been used in order to make it easier to understand since the link between with automata are clearer this way. The partial tree obtained can easily be completed and transformed into a configuration tree as defined above.

($\vee$) nodes represents the choice among $pgenerators$. In fact, all $pgenerator$ accessible from the current one, are computed and linked to this node. They will then be sequentially analyzed until one is successful or all failed. The corresponding choices regarding variable assignments are stored into a shaded box. Note that, if a simulation path is not valid - i.e. an edge S simulated by an edge R - it is immediately denoted as failed. Reached children inherit of the different assignments unless a variable is not useful for any remaining configuration. All of the children must be valid which is symbolized by the ($\wedge$) nodes.

Since, we look for existential simulation, less constraints means more $pconfs$ and at least the ones of the covering one. In Figure 4, the dashed $pconf$ problem is covered by the initial one. Indeed, we face equivalent configurations under the same constraints.

Dealing with different instances of a same variable while aiming for local choices can be dangerous. In order to have a sound algorithm, the simulation tested out by the algorithm is such that a single configuration of A_P must simulate one or more configurations of A_Q. Which means that some transition must be fused in order to synchronize local choices. Indeed if we look at the right branch of Figure 4. Not doing so might duplicate choice possibilities for a variable ending in a false successful run. If we focus on the second $pgenerator$ of Figure 4, 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 z therefore constraints of z_0 can be removed. Then, if we don't fuse, we have two independent unconstrained $pconf$ $pc_1 = ((Q_2, (0)), (P, (0, 1)))$ and $pc_2 = ((C, (0)), (P, (0, 1)))$. The problem lies in the fact that there exists a simulation for the two of them separately by taking $S = z_1$ for pc_1 and $R = z_1$ for pc_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 $(P, (0, 1))$ can simulate the two states by considering a unique choice for a same variable instance which is not possible here.

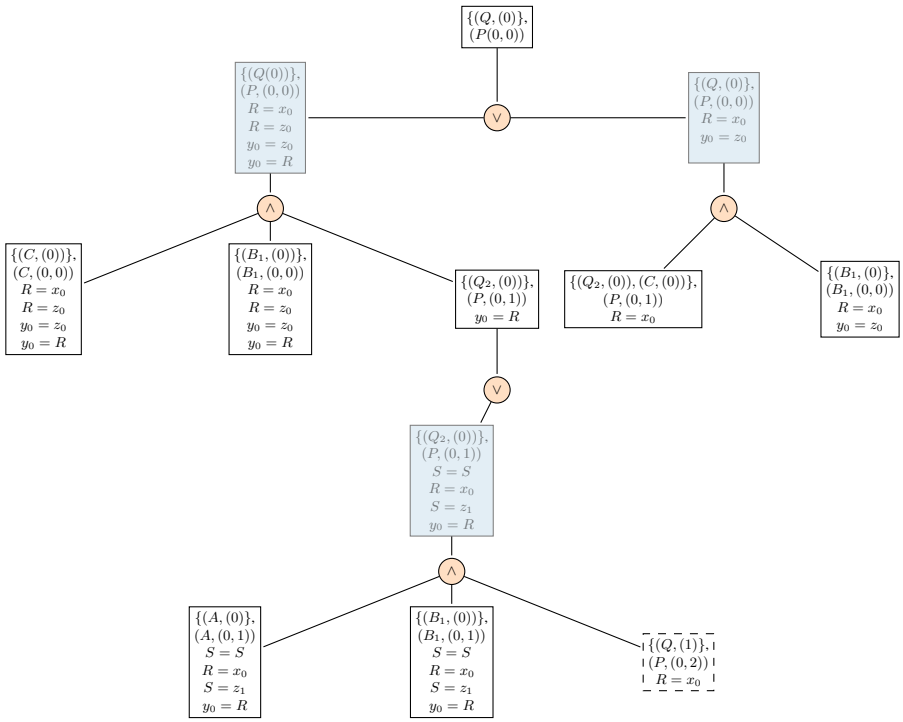


Fig. 4: Testing existential simulation.

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:

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 $pgenerators$ 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 different up 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.

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 13. ***(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 3. *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 is 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 3 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 3, 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.

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) being a 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 substitution fulfilling the simulation relationship.

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

5.3 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.

Proof. 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 either 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 produce 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.

Corollary 1. *Let P,Q be two patterns. Weak subsumption relationship* $P \sqsubseteq\!\!\!\sim\ Q$ *is* EXPTIME*-complete.*

Proof. By theorem 1, since simulation is EXPTIME it proves that deciding weak subsumption is also EXPTIME.

6 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-subsumption, in the context of the description logic $\mathcal{EL}_V$, obtained from an extension of the logic $\mathcal{EL}$ with refreshing variables. Future research works will be devoted to the extension of the approach in three research directions: *(i)* extending our framework to handle concept variables, *(ii)* considering additional reasoning mechanisms in this context that go beyond weak-subsumption (e.g., a form of universal or strong subsumption), and *(iii)* considering other description logics such as the logic $\mathcal{FL}_0$ and $\mathcal{ALN}$.

References

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.: Terminological cycles in a description logic with existential restrictions. In: IJCAI. vol. 3, pp. 325–330 (2003)
4. 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)
5. 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)
6. Baader, F., Küsters, R., Molitor, R.: Structural subsumption considered from an automata-theoretic point of view. In: Proceedings of the 1998 International Workshop on Description Logics (DL'98. Citeseer (1998)
7. Baader, F., Morawska, B.: Unification in the description logic $\mathcal{EL}$. In: International Conference on Rewriting Techniques and Applications. pp. 350–364. Springer (2009)

8. 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)
9. 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)
10. Baader, F., Narendran, P.: Unification of concept terms in description logics. Journal of Symbolic Computation **31**(3), 277–305 (2001)
11. Belkhir, W., Chevalier, Y., Rusinowitch, M.: Fresh-Variable Automata for Service Composition. In: SYNASC 2013 -15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. West University of Timisoara Department of Computer Science, IEEE, Timisoara, Romania (Sep 2013), https://hal.inria.fr/hal-00914778, 28 pages. 4 Figures
12. Borgida, A., Küsters, R.: "what's not in a name?" - initial explorations of a structural approach to integrating large concept knowledge-bases (1999)
13. Borgida, A., Brachman, R., Mcguinness, D., Resnick, L.: Classic: a structural data model for objects. ACM SIGMOD Record **18** (02 1996). https://doi.org/10.1145/67544.66932
14. Brachman, R.J., McGuinness, D.L., Patel-Schneider, P.F., Resnick, L.A., Borgida, A.: Living with classic: When and how to use a kl-one-like language. In: Principles of semantic networks, pp. 401–456. Elsevier (1991)
15. Donini, F., Lenzerini, M., Nardi, D., Schaerf, A.: Reasoning in description logics. Center for the Study of Language and Information (06 1999)
16. Henzinger, T., Qadeer, S., Rajamani, S., Tasiran, S.: An assume-guarantee rule for checking simulation. ACM Trans. Program. Lang. Syst. **24**, 51–64 (2002)
17. MacGregor, R.M.: Inside the loom description classifier. ACM Sigart Bulletin **2**(3), 88–92 (1991)
18. Mcguinness, D.L., Borgida, A.: Explaining Reasoning in Description Logics. Ph.D. thesis, USA (1996)
19. Schmidt-Schauß, M., Smolka, G.: Attributive concept descriptions with complements. Artificial intelligence **48**(1), 1–26 (1991)