Università degli studi di Torino

Tesi di Laurea Magistrale

DysToPic: a Distributed Theorem Prover for non–monotonic Description Logics

Candidato:

Luca Violanti

Abstract

In this work we present a distributed theorem prover for a non-monotonic Description Logic called $\mathcal{ALC} + \mathbf{T}$. The logic is an extension of the Description Logic ALC, adding a typicality operator $\mathbf{T}$, which allows to define concepts such as $\mathbf{T}(C)$, i.e. selecting the "most typical" or "most normal" instances of the concept C. This allows the knowledge bases to contain subsumption relations (e.g. "$\mathbf{T}(C) \sqsubseteq D$" - the typical members of C are instances of the concept D). Moreover, the use of a "minimal model" semantics (i.e. models that maximise typical instances of concepts), allows new forms of non-monotonic reasoning on prototypical properties and defeasible inheritance. Expanding a pre–existing sequential software, the application is based on a SICStus Prolog implementation of the tableaux calculi $\mathcal{TAB}_{min}^{\mathcal{ALC}+\mathbf{T}}$, wrapped by a Java interface which relies on the Java RMI APIs for the distribution. The system is designed for scalability and based on a "worker/employer" paradigm: the computational burden for the "employer" can be spread among an arbitrarily high number of "workers" which operate in complete autonomy, so that they can be either deployed on a single machine or on a computer grid.

1

Contents

1. Introduction

In this thesis we present the design and implementation of a distributed theorem prover for the non–monotonic description logic $\mathcal{ALC} + \mathbf{T}_{min}$.

Description Logics (DL) are a family of formal knowledge representation languages. They are a decidable fragment of the first-order logic formalism used to provide semantics to representation structures. DLs are used in artificial intelligence for formal reasoning on the concepts of an application domain. It is of particular importance in providing a logical formalism for ontologies and the Semantic Web. The most notable applications outside information science are in the field of bioinformatics and in the codification of medical knowledge. A DL knowledge base (KB) comprises two components: the TBox, containing the definition of concepts (and possibly roles) and a specification of inclusion relations among them, and the ABox containing instances of concepts and roles. Since the very objective of the TBox is to build a taxonomy of concepts, the need of representing prototypical properties and of reasoning about defeasible inheritance of such properties naturally arise. The traditional approach is to handle defeasible inheritance by integrating some kind of non–monotonic extension of DLs.

In chapter 2 we introduce the Description Logics and briefly describe some proposals for non–monotonic extensions, while in chapter 3 we describe the non–monotonic logic *alctmin*.

Our implementation is based on the tableaux proof method that is a mechanism that allows to demonstrate the validity of a formula in a given logical system. The tableaux calculus allows us to conclude that a formula is valid (or not) just by applying the tableaux computation rules, i.e., without appealing to the semantics. Proof methods can be implemented in many different programming languages, but logic programming languages offer a better solution since in this paradigm we make use of mathematical logic for computer programming. Logic programming in the first and wider sense gave rise to a large number of implementations. Foster and Elcock's Absys (1969), was probably the first language to be explicitly developed as an assertional programming language. Logic programming in the narrower sense can be traced back to debates in the late 1960s and early 1970s about declarative versus procedural representations of knowledge in Artificial Intelligence.

In chapter 4 we introduce the tableaux calculus for $\mathcal{ALC} + \mathbf{T}_{min}$, which is at the base of our theorem prover7.

Since our work is an expansion of a previous theorem prover called PreDeLo, chapter 6 is dedicated to its description. In chapter 5 we

In chapter

7

2. Description Logics

Description Logics are decidable fragments of the first-order logic formalism used to provide semantics to representation structures like frames and networks. Each fragment has different features, leading to computational problems of variable complexity: the use of a more simple language (low computational complexity) rather than a more expressive one (high computational complexity) depends on how complex we want the world description to be.
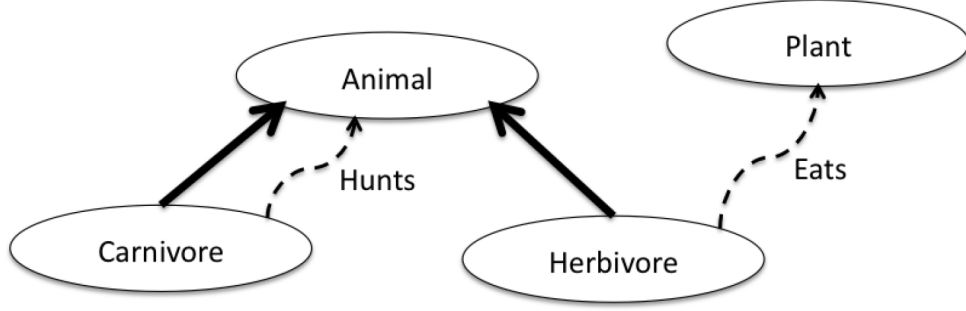

Figure 1: An example of network, showing relationships between concepts.

Typically, nodes are used to characterize concepts, i.e., sets of individuals, and links are used to characterize relationships among them. Concepts can have simple properties, which are typically attached to the corresponding nodes. It is rather simple to make a correspondence between network structures and Description Logics because the latter are equipped with unary predicates representing sets of individuals, binary predicates representing relationships among them and a mechanism of inclusion statements to express concept properties. Indeed, we can represent concept properties with the inclusion statement $Carnivore \sqsubseteq Animal$ and we can express the following facts about an individual $lion$:

- $Carnivore(lion)$

- $Hunts(lion, zebra)$

- $Herbivore(elephant)$

- $Eats(elephant, grass)$

It is also possible to use intersection concepts with the syntax $Carnivore \sqcap Plant$ to select individuals which belong to both $Carnivore$ and $Plant$.

4

2.1. Application domains

Description Logics are used in the implementation of many systems that demonstrate their practical effectiveness. Some of these systems have found their way into production use, despite the fact that there was no real commercial platform that could be used for developing them.

Software engineering. Software Engineering was one of the first application domains for Description Logics. The basic idea was to use a Description Logic to implement a Software Information System, i.e. a system that would support the software developer by helping him in finding out information about a large software system.

Configuration management. One very successful domain for knowledge-based applications built using Description Logics is configuration management, which includes applications that support the design of complex systems created by combining multiple components. The main goal of this discipline is finding a proper set of components that can be suitably connected in order to implement a system that meets a given specification. For example, choosing computer components in order to build a home PC is a relatively simple configuration task. When the number, the type, and the connectivity of the components grow, the configuration task can become rather complex. In particular, computer configuration has been among the application fields of the first Expert Systems and can thus be viewed as a standard application domain for knowledgebased systems. Configuration tasks arise in many industrial domains, such as telecommunications, the automotive industry, building construction, etc. DL-based knowledge representation systems meet the requirements for the development of configuration applications. In particular, they enable the object-oriented modelling of system components, which combines powerfully with the ability to reason from incomplete specifications and to automatically detect inconsistencies. Using Description Logics one can exploit the ability to classify the components and organize them within a taxonomy.

Medicine. Medicine is also a domain where Expert Systems have been developed since the 1980's, in particular in the field of decision support for medical diagnosis. One focus has been on the construction and maintenance of very large ontologies of medical knowledge, the subject of some large government initiatives. In order to cope with the scalability of the knowledge base, the DL language adopted in these applications is often limited to a few basic constructs and the knowledge base turns out to be rather shallow, that is to say the taxonomy does not have very many levels of sub-concepts below the top concepts.

Data mining. Description Logics have also been used in data mining applications, where their inferences can help the process of analysing large amounts of data. In this kind of application, DL structures can represent views, and DL systems can be used to store and classify such views. The classification mechanism can help in discovering interesting classes of items in the data [**?**].

2.2. Basic Description Logics

As the name Description Logics indicates, one of the characteristics of these languages is that they are equipped with a formal, logic-based semantics. Another distinguished feature is the emphasis on reasoning as a central service: reasoning allows one to infer implicitly represented knowledge from the knowledge that is explicitly contained in the knowledge base. Description Logics support inference patterns that occur in many applications of intelligent information processing systems, and which are also used by humans to structure and understand the world: classification of concepts and individuals. Classification of concepts determines subconcept/superconcept relationships (called subsumption relationships in DL) between the concepts of a given terminology, and thus allows one to structure the terminology in the form of a subsumption hierarchy. Classification of individuals (or objects) determines whether a given individual is always an instance of a certain concept (i.e., whether this instance relationship is implied by the description of the individual and the definition of the concept). It thus provides useful information on the properties of an individual. Moreover, instance relationships may trigger the application of rules that insert additional facts into the knowledge base. The implementation of reasoning services requires building procedures that should always terminate, both for positive and for negative answers, but also complexity needs to be taken into consideration. Decidability and complexity of the inference problems depend on the expressive power of the DL at hand. On the one hand, very expressive DLs are likely to have inference problems of high complexity. On the other hand, very weak DLs (with efficient reasoning procedures) may not be sufficiently expressive to represent the important concepts of a given application. Investigating this trade-off between the expressivity of DLs and the complexity of their reasoning problems is one of the most important issues in DL research [**?**].

2.3. The basic description language $\mathcal{AL}$

A knowledge representation system based on Description Logics provides facilities to set up knowledge bases, to reason about their content, and to manipulate them. A knowledge base (KB) comprises two components, the TBox and the ABox. The TBox introduces the terminology, i.e., the vocabulary of an application domain, while the ABox contains assertions about named individuals in terms of this vocabulary. The vocabulary consists of concepts, which denote sets of individuals, and roles, which denote binary relationships between individuals. Elementary descriptions are atomic concepts and atomic roles. Complex descriptions can be built from them inductively with concept constructors. In abstract notation, we use the letters A and B for atomic concepts, the letter R for atomic roles, and the letters C and D for concept descriptions. In the sequel we present the language $\mathcal{AL}$ that has been introduced in [?] as a minimal language that is of practical interest. The other languages of this family are extensions of $\mathcal{AL}$.

Concept descriptions in $\mathcal{AL}$ are formed according to the following syntax rule:

$C, D \rightarrow$

$A \mid$ (atomic concept)

$\top \mid$ (top concept - most general)

$\bot \mid$ (bottom concept - most specific)

$\neg A \mid$ (atomic negation)

$C \sqcap D \mid$ (intersection)

$\forall R.C \mid$ (value restriction)

$\exists R.\top$ (limited existential quantification)

In order to define a formal semantics of $\mathcal{AL}$-concepts we consider interpretations $\mathcal{I}$ that consist of a non-empty set Δ (the domain of the interpretation) and an interpretation function, which assigns to every atomic concept A a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to every atomic role R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The interpretation function is extended to concept descriptions by the following inductive definitions:

$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$

$(\neg A)^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash A^{\mathcal{I}}$

$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \sqcap D^{\mathcal{I}}$

$(\forall R.C)^{\mathcal{I}} = \left\{ x \in \Delta \mid \forall y.(x, y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}} \right\}$

$(\exists R.\top)^{\mathcal{I}} = \left\{ x \in \Delta \mid \exists y.(x, y) \in R^{\mathcal{I}} \right\}$

We say that two concepts C, D are equivalent, and write $C = D$, if $C^{\mathcal{I}} = D^{\mathcal{I}}$ for all interpretations $\mathcal{I}$ [?].

2.4. Complex concept negation: $\mathcal{ALC}$

A basic extension of $\mathcal{AL}$ is the negation of an arbitrary concept:

$$\neg C \qquad\qquad \text{(concept negation)}$$

whose semantics is:

$$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$$

[**?**]

2.5. Non–monotonic reasoning and the operator **T**

Since the very objective of the TBox is to build a taxonomy of concepts, the need of representing prototypical properties and of reasoning about defeasible inheritance of such properties easily arises. The traditional approach is to handle defeasible inheritance by integrating some kind of non–monotonic reasoning mechanism. This has led to study non–monotonic extensions of DLs [**?** **?** **?** **?** **?** **?** **?** **?**]. However, finding a suitable non–monotonic extension for inheritance reasoning with exceptions is far from obvious. To give a brief account, [**?**] proposes the extension of DL with Reiter's default logic. However, the same authors have pointed out that this integration may lead to both semantical and computational difficulties. Indeed, the unsatisfactory treatment of open defaults via Skolemization may lead to an undecidable default consequence relation. For this reason, [**?**] proposes a restricted semantics for open default theories, in which default rules are only applied to individuals explicitly mentioned in the ABox. Furthermore, Reiter's default logic does not provide a direct way of modeling inheritance with exceptions. This has motivated the study of extensions of DLs with prioritized defaults [**?** **?**]. A more general approach is undertaken in [**?**], where it is proposed an extension of DL with two epistemic operators. This extension allows to encode Reiter's default logic as well as to express epistemic concepts and procedural rules. In [**?**] the authors propose an extension of DL with circumscription. One of the motivating applications of circumscription is indeed to express prototypical properties with exceptions, and this is done by introducing "abnormality" predicates, whose extension is minimized. The authors provide decidability and complexity results based on theoretical analysis. A tableau calculus for circumscriptive $\mathcal{ALCO}$ is presented in [**?**].

In [**?** **?**] a non–monotonic extension of $\mathcal{ALC}$ based on the application of Lehmann and Magidor's *rational closure* [**?**] to $\mathcal{ALC}$ is proposed. The approach is based on the introduction of a consequence relation $\vdash\!\!\sim$ among concepts and of a consequence relation $\Vdash$ among an unfoldable KB and assertions. The authors show that such consequence relations are *rational*. It is also shown that such relations inherit the same computational complexity of the underlying DL.

Recent works discuss the combination of open and closed world reasoning in DLs. In particular, formalisms have been defined for combining DLs with logic programming rules (see, for instance, [**?**] and [**?**]). A grounded circumscription approach for DLs with local closed world capabilities has been defined in [**?**].

A non–monotonic extension should have the following characteristics:

1. The (non–monotonic) extension must have a clear semantics and should be based on the same semantics as the underlying monotonic DL.
2. The extension should allow to specify prototypical properties in a natural and direct way.
3. The extension must be decidable, if so is the underlying monotonic DL, and standard proof methods for monotonic DL should be generalized to it.

Before entering the technical details, let us sketch how the pre–existent work makes use of the typicality operator $\mathbf{T}$ for inference. The KB is comprised of the standard TBox and ABox, with the addition of a set of assertions of the form $\mathbf{T}(C) \sqsubseteq D$, where D is a concept not mentioning $\mathbf{T}$.

For instance, let the TBox contain:

$\mathbf{T}(Student) \sqsubseteq \neg IncomeTaxPayer$
$\mathbf{T}(Student \sqcap Worker) \sqsubseteq IncomeTaxPayer$
$\mathbf{T}(Student \sqcap Worker \sqcap Erasmus) \sqsubseteq \neg IncomeTaxPayer$

corresponding to the assertions: typical students don't have to pay income taxes, whereas if a student is also a worker, typically he will pay income taxes; if an Erasmus Programme student is also a worker, he does not have to pay income taxes. Suppose further that the ABox contains alternatively one of the following facts:

1. $Student(luca)$
2. $Student(antonio), Worker(antonio)$
3. $Student(marco), Worker(marco), Erasmus(marco)$

From the different combinations of TBox and one of the above ABox assertions (either 1 or 2 or 3), we can infer the expected (defeasible) conclusions. These are, respectively:

1. $\neg IncomeTaxPayer(luca)$
2. $IncomeTaxPayer(antonio)$
3. $\neg IncomeTaxPayer(marco)$

Moreover, we would also like to infer (defeasible) properties of individuals implicitly introduced by existential restrictions, for instance, if the ABox contains

$\exists HasChild.Student(grazia)$

we would like to infer that:

$\exists HasChild.\neg IncomeTaxPayer(grazia)$

Finally, adding irrelevant information should not affect the conclusions. From the TBox above, one should be able to infer as well

$\mathbf{T}(Student \sqcap Tall) \sqsubseteq \neg IncomeTaxPayer$
$\mathbf{T}(Student \sqcap Worker \sqcap Tall) \sqsubseteq IncomeTaxPayer$
$\mathbf{T}(Student \sqcap Worker \sqcap Erasmus \sqcap Tall) \sqsubseteq \neg IncomeTaxPayer$

as *Tall* is irrelevant with respect to paying income taxes or not. For the same reason, the conclusion about *luca* being an instance of $\neg IncomeTaxPayer$ or not should not be influenced by adding $Tall(luca)$ to the ABox.

Hereinafter we introduce a non–monotonic extension for defeasible reasoning in description logics, called $\mathcal{ALC} + \mathbf{T}_{min}$, which is an extension of $\mathcal{ALC} + \mathbf{T}$.

3. Description Logics for typicality

3.1. The logic $\mathcal{ALC} + \mathbf{T}$

In this section, we recall the original $\mathcal{ALC} + \mathbf{T}$, which is an extension of $\mathcal{ALC}$ by a typicality operator $\mathbf{T}$ introduced in [**?**]. Given an alphabet of concept names $\mathcal{C}$, of role names $\mathcal{R}$, and of individual constants $\mathcal{O}$, the language $\mathcal{L}$ of the logic $\mathcal{ALC} + \mathbf{T}$ is defined by distinguishing *concepts* and *extended concepts* as follows:

- (Concepts)
    - $A \in \mathcal{C}$, $\top$ and $\bot$ are *concepts* of $\mathcal{L}$;
    - if $C, D \in \mathcal{L}$ and $R \in \mathcal{R}$, then $C \sqcap D, C \sqcup D, \neg C, \forall R.C, \exists R.C$ are *concepts* of $\mathcal{L}$

- (Extended concepts)
    - if C is a concept of $\mathcal{L}$, then C and $\mathbf{T}(C)$ are *extended concepts* of $\mathcal{L}$
    - boolean combinations of extended concepts are extended concepts of $\mathcal{L}$.

A knowledge base is a pair (TBox, ABox). TBox contains subsumptions $C \sqsubseteq D$, where $C \in \mathcal{L}$ is an extended concept of the form either C' or $\mathbf{T}(C')$, and $C', D \in \mathcal{L}$ are concepts. ABox contains expressions of the form $C(a)$ and aRb where $C \in \mathcal{L}$ is an extended concept, $R \in \mathcal{R}$, and $a, b \in \mathcal{O}$.

In order to provide a semantics to the operator $\mathbf{T}$, we extend the definition of a model used in "standard" terminological logic $\mathcal{ALC}$:

Definition 1 (Semantics of T with selection function). A model is any structure $\langle \Delta, I, f_{\mathbf{T}} \rangle$ where:

- Δ is the domain, whose elements are denoted with $x, y, z, \dots$;

- I is the extension function that maps each extended concept C to $C^I \subseteq \Delta$, and each role R to a $R^I \subseteq \Delta \times \Delta$. I assigns to each atomic concept $A \in \mathcal{C}$ a set $A^I \subseteq \Delta$ and it is extended to arbitrary extended concepts as follows:
    - $\top^I = \Delta$
    - $\bot^I = \emptyset$
    - $(\neg C)^I = \Delta \backslash C^I$
    - $(C \sqcap D)^I = C^I \cap D^I$
    - $(C \sqcup D)^I = C^I \cup D^I$
    - $(\forall R.C)^I = \{x \in \Delta \mid \forall y.(x,y) \in R^I \rightarrow y \in C^I\}$
    - $(\exists R.C)^I = \{x \in \Delta \mid \exists y.(x,y) \in R^I \text{ and } y \in C^I\}$
    - $(\mathbf{T}(C))^I = f_{\mathbf{T}}(C^I)$

- Given $S \subseteq \Delta$, $f_{\mathbf{T}}$ is a function $f_{\mathbf{T}} : Pow(\Delta) \rightarrow Pow(\Delta)$ satisfying the following properties:
    - $(f_{\mathbf{T}} - 1)$ $f_{\mathbf{T}}(S) \subseteq S$;
    - $(f_{\mathbf{T}} - 2)$ if $S \neq \emptyset$, then also $f_{\mathbf{T}}(S) \neq \emptyset$;
    - $(f_{\mathbf{T}} - 3)$ if $f_{\mathbf{T}}(S) \subseteq R$, then $f_{\mathbf{T}}(S) = f_{\mathbf{T}}(S \cap R)$;

$$- (f_\mathbf{T} - 4) \ f_\mathbf{T}(\bigcup S_i) \subseteq \bigcup f_\mathbf{T}(S_i);$$
$$- (f_\mathbf{T} - 5) \ \bigcap f_\mathbf{T}(S_i) \subseteq f_\mathbf{T}(\bigcup S_i).$$

Intuitively, given the extension of some concept C, the selection function $f_\mathbf{T}$ selects the *typical* instances of C. $(f_\mathbf{T} - 1)$ requests that typical elements of S belong to S. $(f_\mathbf{T} - 2)$ requests that if there are elements in S, then there are also *typical* such elements. The following properties constrain the behaviour of $f_\mathbf{T}$ with respect to $\cap$ and $\cup$ in such a way that they do not entail monotonicity. According to $(f_\mathbf{T} - 3)$, if the typical elements of S are in R, then they coincide with the typical elements of $S \cap R$, thus expressing a weak form of monotonicity (namely, *cautious monotonicity*). $(f_\mathbf{T} - 4)$ corresponds to one direction of the equivalence $f_\mathbf{T}(\bigcup S_i) = \bigcup f_\mathbf{T}(S_i)$, so that it does not entail monotonicity. Similar considerations apply to the equation $f_\mathbf{T}(\bigcap S_i) = \bigcap f_\mathbf{T}(S_i)$, of which only the inclusion $\bigcap f_\mathbf{T}(S_i) \subseteq f_\mathbf{T}(\bigcap S_i)$ holds. $(f_\mathbf{T} - 5)$ is a further constraint on the behavior of $f_\mathbf{T}$ with respect to arbitrary unions and intersections; it would be derivable if $f_\mathbf{T}$ were monotonic. In [?], it is shown that one can give an equivalent, alternative semantics for $\mathbf{T}$ based on a *preference relation* semantics rather than on a selection function semantics. The idea is that there is a global, irreflexive and transitive relation among individuals and that the typical members of a concept C (i.e., those selected by $f_\mathbf{T}(C^I)$) are the minimal elements of C with respect to this relation. Observe that this notion is *global*, that is to say, it does not compare individuals with respect to a specific concept. For this reason, we cannot express the fact that y is more typical than x with respect to concept C, whereas x is more typical than y with respect to another concept D. All what we can say is that either x is incomparable with y or x is more typical than y or y is more typical than x. In this framework, an element $x \in \Delta$ is a *typical instance* of some concept C if $x \in C^I$ and there is no C-element in Δ *more typical* than x. The typicality preference relation is partial since it is not always possible to establish given two element which one of the two is more typical. Following KLM, the preference relation also satisfies a *Smoothness Condition*, which is related to the well known *Limit Assumption* in Conditional Logics [?] [1]; this condition ensures that, if the extension C^I of a concept C is not empty, then there is at least one *minimal* element of C^I. This is stated in a rigorous manner in the following definition:

Definition 2. Given an irreflexive and transitive relation $<$ over a domain Δ, called *preference relation*, for all $S \subseteq \Delta$, we define
$Min_<(S) = \{x \in S \mid \nexists y \in S \text{ s.t. } y < x\}$
We say that $<$ satisfies the *Smoothness Condition* if for all $S \subseteq \Delta$, for all $x \in S$, either $x \in Min_<(S)$ or $\exists y \in Min_<(S)$ such that $y < x$.

The following representation theorem is proved in [?]:

Theorem 1 (Theorem 2.1 in [?]). *Given any model $\langle \Delta, I, f_\mathbf{T} \rangle$, $f_\mathbf{T}$ satisfies postulates $(f_\mathbf{T} - 1)$ to $(f_\mathbf{T} - 5)$ above iff there exists an irreflexive and transitive relation $<$ on Δ, satisfying the Smoothness Condition, such that for all $S \subseteq \Delta$, $f_\mathbf{T}(S) = Min_<(S)$.*

Having the above Representation Theorem, from now on, we will refer to the following semantics:

---

[1]More precisely, the Limit Assumption entails the Smoothness Condition (i.e. that there are no infinite $<$ descending chains). Both properties come for free in finite models.

Definition 3 (Semantics of $\mathcal{ALC} + \mathbf{T}$). A model $\mathcal{M}$ of $\mathcal{ALC} + \mathbf{T}$ is any structure

$$\langle \Delta, I, < \rangle$$

where:

- Δ is the domain;

- $<$ is an irreflexive and transitive relation over Δ satisfying the Smoothness Condition (Definition 2)

- I is the extension function that maps each extended concept C to $C^I \subseteq \Delta$, and each role R to a $R^I \subseteq \Delta \times \Delta$. I assigns to each atomic concept $A \in \mathcal{C}$ a set $A^I \subseteq \Delta$. Furthermore, I is extended as in Definition 1 with the exception of $(\mathbf{T}(C))^I$, which is defined as

$$(\mathbf{T}(C))^I = Min_<(C^I).$$

Let us now introduce the notion of satisfiability of an $\mathcal{ALC} + \mathbf{T}$ knowledge base. In order to define the semantics of the assertions of the ABox, we extend the function I to individual constants; we assign to each individual constant $a \in \mathcal{O}$ a *distinct* domain element $a^I \in \Delta$, that is to say we enforce the *unique name assumption*. As usual, the adoption of the unique name assumption greatly simplifies reasoning about prototypical properties of individuals denoted by different individual constants. Considering the example of department staff having lunches, if (in addition to the TBox) the ABox only contains the following facts about Greg and Sara:

$DepartmentMember(greg)$
$DepartmentMember(sara), TemporaryWorker(sara)$

we would like to infer that Greg takes his lunches at the restaurant, whereas Sara does not; but without the unique name hypothesis, we cannot get this conclusion since Greg and Sara might be the same individual. To perform useful reasoning we would need to extend the language with equality and make a case analysis according to possible identities of individuals. While this is technically possible, we prefer to keep the things simple here by adopting the unique name assumption.

Definition 4 (Model satisfying a Knowledge Base). Consider a model $\mathcal{M}$, as defined in Definition 3. We extend I so that it assigns to each individual constant a of $\mathcal{O}$ an element $a^I \in \Delta$, and I satisfies the unique name assumption. Given a KB (TBox,ABox), we say that:

- $\mathcal{M}$ satisfies TBox iff for all inclusions $C \sqsubseteq D$ in TBox, $C^I \subseteq D^I$.

- $\mathcal{M}$ satisfies ABox iff: (i) for all $C(a)$ in ABox, we have that $a^I \in C^I$, (ii) for all aRb in ABox, we have that $(a^I, b^I) \in R^I$.

$\mathcal{M}$ satisfies a knowledge base if it satisfies both its TBox and its ABox. Last, a query F is entailed by KB in $\mathcal{ALC} + \mathbf{T}$ if it holds in all models satisfying KB. In this case we write $KB \models_{\mathcal{ALC}+\mathbf{T}} F$.

Notice that the meaning of $\mathbf{T}$ can be split into two parts: for any x of the domain Δ, $x \in (\mathbf{T}(C))^I$ just in case (i) $x \in C^I$, and (ii) there is no $y \in C^I$ such that $y < x$. As

already mentioned in the Introduction, in order to isolate the second part of the meaning of **T** (for the purpose of the calculus that we will present in Section **??**), we introduce a new modality $\square$. The basic idea is simply to interpret the preference relation $<$ as an accessibility relation. By the Smoothness Condition, it turns out that $\square$ has the properties as in Gödel-Löb modal logic of provability G. The Smoothness Condition ensures that typical elements of C^I exist whenever $C^I \neq \emptyset$, by avoiding infinitely descending chains of elements. This condition therefore corresponds to the finite-chain condition on the accessibility relation (as in G). The interpretation of $\square$ in $\mathcal{M}$ is as follows:

Definition 5. Given a model $\mathcal{M}$ as in Definition 3, we extend the definition of I with the following clause:

$$(\square C)^I = \{x \in \Delta \mid \text{for every } y \in \Delta, \text{ if } y < x \text{ then } y \in C^I\}$$

It is easy to observe that x is a typical instance of C if and only if it is an instance of C and $\square \neg C$, that is to say:

Proposition 1. *Given a model $\mathcal{M}$ as in Definition 3, given a concept C and an element $x \in \Delta$, we have that*
$$x \in (\mathbf{T}(C))^I \text{ iff } x \in (C \sqcap \square \neg C)^I$$

Since we only use $\square$ to capture the meaning of **T**, in the following we will always use the modality $\square$ followed by a negated concept, as in $\square \neg C$.

The Smoothness condition, together with the transitivity of $<$, ensures the following Lemma:

Lemma 1. *Given an $\mathcal{ALC} + \mathbf{T}$ model as in Definition 3, an extended concept C, and an element $x \in \Delta$, if there exists $y < x$ such that $y \in C^I$, then either $y \in Min_<(C^I)$ or there is $z < x$ such that $z \in Min_<(C^I)$.*

Last, we state a theorem which will be used in the following:

Theorem 2 (Finite model property of $\mathcal{ALC} + \mathbf{T}$)**.** *The logic $\mathcal{ALC} + \mathbf{T}$ has the finite model property.*

3.2. The logic $\mathcal{ALC} + \mathbf{T}_{\min}$

As mentioned in the Introduction, the logic $\mathcal{ALC} + \mathbf{T}$ presented in [**?**] allows to reason about typicality. As a difference with respect to standard $\mathcal{ALC}$, in $\mathcal{ALC} + \mathbf{T}$ we can consistently express, for instance, the fact that three different concepts, like *Department member*, *Temporary Department Member* and *Temporary Department member having restaurant tickets*, have a different status with respect to *Have lunch at a restaurant*. This can be consistently expressed by including in a knowledge base the three formulas:

$\mathbf{T}(DepartmentMember) \sqsubseteq LunchAtRestaurant$
$\mathbf{T}(DepartmentMember \sqcap TemporaryResearcher) \sqsubseteq \neg LunchAtRestaurant$
$\mathbf{T}(DepartmentMember \sqcap TemporaryResearcher \sqcap \exists Owns.RestaurantTicket) \sqsubseteq$
$LunchAtRestaurant$

Assume that *greg* is an instance of the concept $DepartmentMember \sqcap TemporaryResearcher \sqcap \exists Owns.RestaurantTicket$. What can we conclude about *greg*? We have already mentioned that if the ABox explicitly points out that *greg* is a *typical* instance of the concept, and it contains the assertion that:

(∗) $\mathbf{T}(DepartmentMember \sqcap TemporaryResearcher \sqcap \exists Owns.RestaurantTicket)(greg)$,
then, in $\mathcal{ALC} + \mathbf{T}$, we can conclude that
$LunchAtRestaurant(greg)$.

However, if (*) is replaced by the weaker (**) $(DepartmentMember \sqcap TemporaryResearcher \sqcap \exists Owns.RestaurantTicket)(greg)$, in which there is no information about the typicality of *greg*, in $\mathcal{ALC} + \mathbf{T}$ we can no longer draw this conclusion, and indeed we cannot make any inference about whether *greg* spends its lunch time at a restaurant or not. The limitation here lies in the fact that $\mathcal{ALC} + \mathbf{T}$ is *monotonic*, whereas we would like to make a non–monotonic inference. Indeed, we would like to non–monotonically assume, in the absence of information to the contrary, that *greg* is a typical instance of the concept. In general, we would like to infer that individuals are typical instances of the concepts they belong to, if this is consistent with the KB.

As a difference with respect to $\mathcal{ALC} + \mathbf{T}$, $\mathcal{ALC} + \mathbf{T}_{min}$ is *non–monotonic*, and it allows to make this kind of inference. Indeed, in $\mathcal{ALC} + \mathbf{T}_{min}$ if (**) is all the information about *greg* present in the ABox, we can derive that *greg* is a typical instance of the concept, and from the inclusions above we conclude that $LunchAtRestaurant(greg)$. We have already mentioned that we obtain this non–monotonic behaviour by restricting our attention to the minimal $\mathcal{ALC} + \mathbf{T}$ models. As a difference with respect to $\mathcal{ALC} + \mathbf{T}$, in order to determine what is entailed by a given knowledge base KB, we do not consider *all* models of KB but only the *minimal* ones. These are the models that minimize the number of atypical instances of concepts.

Given a KB, we consider a finite set $\mathcal{L}_{\mathbf{T}}$ of concepts occurring in the KB: these are the concepts for which we want to minimize the atypical instances.

The minimization of the set of atypical instances will apply to individuals explicitly occurring in the ABox as well as to implicit individuals. We assume that the set $\mathcal{L}_{\mathbf{T}}$ contains at least all concepts C such that $\mathbf{T}(C)$ occurs in the KB. Notice that in case $\mathcal{L}_{\mathbf{T}}$ contains more concepts than those occurring in the scope of $\mathbf{T}$ in KB, the atypical instances of these concepts will be minimized but no extra properties will be inferred for the typical instances of the concepts, since the KB does not say anything about these instances.

We have seen that $(\mathbf{T}(C))^I = (C \sqcap \Box \neg C)^I$: x isa typical instance of a concept C ($x \in (\mathbf{T}(C))^I$) when it is an instance of C and there is no other instance of C preferred to x, i.e. $x \in (C \sqcap \Box \neg C)^I$. By contraposition an instance of C is atypical if $x \in (\neg \Box \neg C)^I$ therefore in order to minimize the atypical instances of C, we minimize the instances of $\neg \Box \neg C$. Notice that this is different from maximizing the instances of $\mathbf{T}(C)$. We have adopted this solution since it allows to maximize the set of typical instances of C without affecting the extension C^I of C (whereas maximizing the extension of $\mathbf{T}(C)$ would imply maximizing also the extension of C).

We define the set $\mathcal{M}_{\mathcal{L}_{\mathbf{T}}}^{\Box^-}$ of negated boxed formulas holding in a model, relative to the concepts in $\mathcal{L}_{\mathbf{T}}$:

Definition 6. Given a model $\mathcal{M} = \langle \Delta, I, < \rangle$ and a set of concepts $\mathcal{L}_{\mathbf{T}}$, we define

$$\mathcal{M}_{\mathcal{L}_{\mathbf{T}}}^{\Box^-} = \{(x, \neg \Box \neg C) \mid x \in (\neg \Box \neg C)^I, \text{ with } x \in \Delta, C \in \mathcal{L}_{\mathbf{T}}\}$$

Let KB be a knowledge base and let $\mathcal{L}_{\mathbf{T}}$ be a set of concepts occurring in KB.

Definition 7 (Preferred and minimal models). Given a model $\mathcal{M} = \langle \Delta_{\mathcal{M}}, I_{\mathcal{M}}, <_{\mathcal{M}} \rangle$ of KB and a model $\mathcal{N} = \langle \Delta_{\mathcal{N}}, I_{\mathcal{N}}, <_{\mathcal{N}} \rangle$ of KB, we say that $\mathcal{M}$ is preferred to $\mathcal{N}$ with respect to $\mathcal{L}_{\mathbf{T}}$, and we write $\mathcal{M} <_{\mathcal{L}_{\mathbf{T}}} \mathcal{N}$, if the following conditions hold:

- $\Delta_{\mathcal{M}} = \Delta_{\mathcal{N}}$

- $a^{I_{\mathcal{M}}} = a^{I_{\mathcal{N}}}$ for all individual constants $a \in \mathcal{O}$

- $\mathcal{M}_{\mathcal{L}_{\mathbf{T}}}^{\Box^-} \subset \mathcal{N}_{\mathcal{L}_{\mathbf{T}}}^{\Box^-}$.

A model $\mathcal{M}$ is a *minimal model* for KB (with respect to $\mathcal{L}_{\mathbf{T}}$) if it is a model of KB and there is no a model $\mathcal{M}'$ of KB such that $\mathcal{M}' <_{\mathcal{L}_{\mathbf{T}}} \mathcal{M}$.

Given the notion of preferred and minimal models above, we introduce a notion of *minimal entailment*, that is to say we restrict our consideration to minimal models only. First of all, we introduce the notion of *query*, which can be minimally entailed from a given KB. A query F is a formula of the form $C(a)$ where C is an extended concept and $a \in \mathcal{O}$. We assume that, for all $\mathbf{T}(C')$ occurring in F, $C' \in \mathcal{L}_{\mathbf{T}}$. Given a KB and a model $\mathcal{M} = \langle \Delta, I, < \rangle$ satisfying it, we say that a query $C(a)$ holds in $\mathcal{M}$ if $a^I \in C^I$.
Let us now define minimal entailment of a query in $\mathcal{ALC} + \mathbf{T}_{min}$. In Section **??** we will reduce the other standard reasoning tasks to minimal entailment.

Definition 8 (Minimal Entailment in $\mathcal{ALC} + \mathbf{T}_{min}$). A query F is minimally entailed from a knowledge base KB with respect to $\mathcal{L}_{\mathbf{T}}$ if it holds in all models of KB that are minimal with respect to $\mathcal{L}_{\mathbf{T}}$. We write $KB \models_{min}^{\mathcal{L}_{\mathbf{T}}} F$.

The non–monotonic character of $\mathcal{ALC} + \mathbf{T}_{min}$ also allows to deal with the following examples.

Example 1. Consider the following KB:

$$\text{KB} = \{\mathbf{T}(Athlete) \sqsubseteq Confident, Athlete(john), Finnish(john)\}$$

and $\mathcal{L}_{\mathbf{T}} = \{Athlete, Finnish\}$. We have

$$\text{KB} \models_{min}^{\mathcal{L}_{\mathbf{T}}} Confident(john)$$

Indeed, there is no minimal model of KB that contains a non typical instance of some concept (indeed in all minimal models of KB the relation $<$ is empty). Hence *john* is an instance of $\mathbf{T}(Athlete)$ (it can be easily verified that any model in which *john* is not an instance of $\mathbf{T}(Athlete)$ is not minimal). By KB, in all these models, *john* is an instance of *Confident*. Observe that *Confident(john)* is obtained, in spite of the presence of the irrelevant assertion *Finnish(john)*.

Example 2. Consider now the knowledge base KB' obtained by adding to KB the formula $\mathbf{T}(Athlete \sqcap Finnish) \sqsubseteq \neg Confident$, that is to say: KB' $= \{\mathbf{T}(Athlete) \sqsubseteq Confident, \mathbf{T}(Athlete \sqcap Finnish) \sqsubseteq \neg Confident, Athlete(john), Finnish(john)\}$ and to $\mathcal{L}_{\mathbf{T}}$ concept $Athlete \sqcap Finnish$. From KB', *Confident(john)* is no longer derivable. Instead, we have that KB' $\models_{min}^{\mathcal{L}_{\mathbf{T}}} \neg Confident(john)$. Indeed, by reasoning as above it can be shown that in all the minimal models of KB', *john* is an instance of $\mathbf{T}(Athlete \sqcap Finnish)$, and it is no longer an instance of $\mathbf{T}(Athlete)$. This example shows that, in case of conflict (here, *john* cannot be both a typical instance of *Athlete* and of $Athlete \sqcap Finnish$), typicality in the more specific concept is preferred.

In general, a knowledge base KB may have no minimal model or more than one minimal model, with respect to a given $\mathcal{L}_{\mathbf{T}}$. The following property holds.

Proposition 2. *If* KB *has a model, then* KB *has a minimal model with respect to any* $\mathcal{L}_{\mathbf{T}}$.

The above fact is a consequence of the *finite model property* of the logic $\mathcal{ALC}+\mathbf{T}$ (Theorem 2).

4. A tableaux calculus for $\mathcal{ALC}+\mathbf{T}_{min}$

In this section we present a tableau calculus for deciding whether a query F is minimally entailed by a knowledge base (TBox,ABox). We introduce a labelled tableau calculus called $\mathcal{TAB}_{min}^{\mathcal{ALC}+\mathbf{T}}$, which extends the calculus $T^{\mathcal{ALC}+\mathbf{T}}$ presented in [**?**], and allows to reason about minimal models.

$\mathcal{TAB}_{min}^{\mathcal{ALC}+\mathbf{T}}$ performs a two-phase computation in order to check whether a query F is minimally entailed from the initial KB. In particular, the procedure tries to build an open branch representing a minimal model satisfying KB $\cup \{\neg F\}$.

In the first phase, a tableau calculus, called $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$, simply verifies whether KB $\cup \{\neg F\}$ is satisfiable in an $\mathcal{ALC}+\mathbf{T}$ model, building candidate models. In the second phase another tableau calculus, called $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$, checks whether the candidate models found in the first phase are *minimal* models of KB. To this purpose for each open branch of the first phase, $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ tries to build a "smaller" model of KB, i.e. a model whose individuals satisfy less formulas $\neg\square\neg C$ than the corresponding candidate model. The whole procedure $\mathcal{TAB}_{min}^{\mathcal{ALC}+\mathbf{T}}$ is formally defined at the end of this section (Definition 17). $\mathcal{TAB}_{min}^{\mathcal{ALC}+\mathbf{T}}$ is based on the notion of a *constraint system*. We consider a set of *variables* drawn from a denumerable set $\mathcal{V}$. Variables are used to represent individuals not explicitly mentioned in the ABox, that is to say implicitly expressed by existential as well as universal restrictions.

$\mathcal{TAB}_{min}^{\mathcal{ALC}+\mathbf{T}}$ makes use of labels, which are denoted with $x, y, z, \ldots$. A label represents either a variable or an individual constant occurring in the ABox, that is to say an element of $\mathcal{O} \cup \mathcal{V}$.

Definition 9 (Constraint). A *constraint* (or *labelled* formula) is a syntactic entity of the form either $x \xrightarrow{R} y$ or $y < x$ or $x : C$, where x, y are labels, R is a role and C is either an extended concept or has the form $\square\neg D$ or $\neg\square\neg D$, where D is a concept.

Intuitively, a constraint of the form $x \xrightarrow{R} y$ says that the individual represented by label x is related to the one denoted by y by means of role R; a constraint $y < x$ says that the individual denoted by y is "preferred" to the individual represented by x with respect to the relation $<$; a constraint $x : C$ says that the individual denoted by x is an instance of the concept C, i.e. it belongs to the extension C^I. As we will define in Definition 11, the ABox of a knowledge base can be translated into a set of constraints by replacing every membership assertion $C(a)$ with the constraint $a : C$ and every role aRb with the constraint $a \xrightarrow{R} b$.

Let us now separately analyze the two components of the calculus $\mathcal{TAB}_{min}^{\mathcal{ALC}+\mathbf{T}}$, starting with $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$.

4.1. The tableau calculus $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$

Let us first define the basic notions of a tableau system in $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$:

Definition 10 (Tableau of $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$). A tableau of $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ is a tree whose nodes are constraint systems, i.e., pairs $\langle S \mid U \rangle$, where S is a set of constraints, whereas U contains formulas of the form $C \sqsubseteq D^L$, representing subsumption relations $C \sqsubseteq D$ of the TBox. L is a list of labels [2]. A branch is a sequence of nodes $\langle S_1 \mid U_1 \rangle, \langle S_2 \mid U_2 \rangle, \ldots, \langle S_n \mid$

---

[2]As we will discuss later, this list is used in order to ensure the termination of the tableau calculus.

$U_n\rangle \ldots$, where each node $\langle S_i \mid U_i\rangle$ is obtained from its immediate predecessor $\langle S_{i-1} \mid U_{i-1}\rangle$ by applying a rule of $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ (see Figure 2), having $\langle S_{i-1} \mid U_{i-1}\rangle$ as the premise and $\langle S_i \mid U_i\rangle$ as one of its conclusions. A branch is closed if one of its nodes is an instance of *clash* (either (Clash) or (Clash)$_\top$ or (Clash)$_\perp$), otherwise it is open. A tableau is closed if all its branches are closed.

In order to check the satisfiability of a KB, we build the corresponding constraint system $\langle S \mid U\rangle$, and we check its satisfiability.

Definition 11 (Corresponding constraint system). Given a knowledge base $KB=$(TBox,ABox), we define its *corresponding constraint system* $\langle S \mid U\rangle$ as follows:

- $S = \{a : C \mid C(a) \in \text{ABox}\} \cup \{a \xrightarrow{R} b \mid aRb \in \text{ABox}\}$

- $U = \{C \sqsubseteq D^\emptyset \mid C \sqsubseteq D \in \text{TBox}\}$

Definition 12 (Model satisfying a constraint system). Let $\mathcal{M} = \langle \Delta, I, < \rangle$ be a model as defined in Definition 3. We define a function α which assigns to each variable of $\mathcal{V}$ an element of Δ, and assigns every individual constant $a \in \mathcal{O}$ to $a^I \in \Delta$. $\mathcal{M}$ satisfies a constraint F under α, written $\mathcal{M} \models_\alpha F$, as follows:

- $\mathcal{M} \models_\alpha x : C$ if and only if $\alpha(x) \in C^I$

- $\mathcal{M} \models_\alpha x \xrightarrow{R} y$ if and only if $(\alpha(x), \alpha(y)) \in R^I$

- $\mathcal{M} \models_\alpha y < x$ if and only if $\alpha(y) < \alpha(x)$

A constraint system $\langle S \mid U\rangle$ is satisfiable if there is a model $\mathcal{M}$ and a function α such that $\mathcal{M}$ satisfies every constraint in S under α and that, for all $C \sqsubseteq D^L \in U$ and for all $x \in \Delta$, we have that if $x \in C^I$ then $x \in D^I$.

Let us now show that:

Proposition 3. *KB=(TBox,ABox) is satisfiable in an $\mathcal{ALC} + \mathbf{T}$ model if and only if its corresponding constraint system $\langle S \mid U\rangle$ is satisfiable in the same model.*

To verify the satisfiability of KB $\cup \{\neg F\}$, we use $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ to check the satisfiability of the constraint system $\langle S \mid U\rangle$ obtained by adding the constraint corresponding to $\neg F$ to S', where $\langle S' \mid U\rangle$ is the corresponding constraint system of KB. To this purpose, the rules of the calculus $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ are applied until either a contradiction is generated (*clash*) or a model satisfying $\langle S \mid U\rangle$ can be obtained from the resulting constraint system. As in the calculus proposed in [**?**], given a node $\langle S \mid U\rangle$, for each subsumption $C \sqsubseteq D^L \in U$ and for each label x that appears in the tableau, we add to S the constraint $x : \neg C \sqcup D$: we refer to this mechanism as *subsumption expansion*. As mentioned above, each subsumption $C \sqsubseteq D$ is equipped with a list L of labels in which the subsumption has been expanded in the current branch. This is needed to avoid multiple expansions of the same subsumption by using the same label, generating infinite branches.

Before introducing the rules of $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ we need some more definitions. First, as in [**?**], we define an ordering relation $\prec$ to keep track of the temporal ordering of insertion of labels in the tableau, that is to say if y is introduced in the tableau, then $x \prec y$ for all labels x that are already in the tableau. Moreover, we need to define the *equivalence* between two labels: intuitively, two labels x and y are equivalent if they label the same set of extended concepts. This notion is stated in the following definition, and it is used in order to apply the blocking machinery described in the following, based on the fact that equivalent labels represent the same element in the model built by $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$.

Figure 2: The calculus $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$. To save space, we omit the rules $(\forall^-)$ and $(\exists^-)$, dual to $(\exists^+)$ and $(\forall^+)$, respectively.

Definition 13. Given a tableau node $\langle S \mid U \rangle$ and a label x, we define $\sigma(\langle S \mid U \rangle, x) = \{C \mid x : C \in S\}$.
Furthermore, we say that two labels x and y are *S-equivalent*, written $x \equiv_S y$, if they label the same set of concepts, i.e.
$$\sigma(\langle S \mid U \rangle, x) = \sigma(\langle S \mid U \rangle, y).$$

Last, we define the set of formulas $S_{x \to y}^M$, that will be used in the rule $(\square^-)$ when $y < x$, in order to introduce $y : \neg C$ and $y : \square \neg C$ for each $x : \square \neg C$ in the current branch:

Definition 14. Given a tableau node $\langle S \mid U \rangle$ and two labels x and y, we define

$$S_{x \to y}^M = \{ y : \neg C, y : \square \neg C \mid x : \square \neg C \in S \}.$$

The rules of $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ are presented in Figure 2. Rules $(\exists^+)$ and $(\square^-)$ are called *dynamic* since they introduce a new variable in their conclusions. The other rules are called *static*. A brief explanation of the rules follows:

- (Clash), $(\text{Clash})_\top$ and $(\text{Clash})_\perp$ are used to detect *clashes*, i.e. unsatisfiable constraint systems;

- rules for $\sqcup$, $\sqcap$, $\neg$, and $\forall$ are similar to the corresponding ones in the tableau calculus for standard $\mathcal{ALC}$ [**?**]: as an example, the rule $(\sqcup^+)$ is applied to a constraint system of the form $\langle S, x : C \sqcup D \mid U \rangle$ in order to deal with the constraint $x : C \sqcup D$

18

introducing two branches in the tableau construction, to check the two conclusions obtained by adding the constraints $x : C$ and $x : D$, respectively. The side condition of the rules are the usual conditions needed to avoid multiple applications on the same principal formula: concerning the example of $(\sqcup^+)$, it can be applied only if $x : C \notin S$ and $x : D \notin S$;

- the rules $(\mathbf{T}^+)$ and $(\mathbf{T}^-)$ are used to "translate" formulas of the form $\mathbf{T}(C)$ in the corresponding modal interpretation: for $(\mathbf{T}^+)$, this corresponds to introduce $x : C \sqcap \Box \neg C$ to a constraint system containing $x : \mathbf{T}(C)$, whereas for $(\mathbf{T}^-)$ a branching is introduced to add either $x : \neg C$ or $x : \neg \Box \neg C$ in case $x : \neg \mathbf{T}(C)$ belongs to the constraint system;

- the rule $(\sqsubseteq)$ is used in order to check whether, for all x belonging to a branch, the inclusion relations of the TBox are satisfied: given a label x and an inclusion $C \sqsubseteq D^L \in U$, the branching introduced by the rule ensures that either $x : \neg C$ holds or that $x : D$ holds;

- the rule $(\Box^-)$, applied to a principal formula $x : \neg \Box \neg C$ (x is not a typical instance of the concept C, i.e. there exists an element z which is a typical instance of C and is more normal than x), introduces the constraints $z < x$, $z : C$ and $z : \Box \neg C$. A branching on the choice of the label z to use is introduced, since it can be either a "new" label y, not occurring in the branch, or one of the labels $v_1, v_2, \ldots, v_n$ already belonging to the branch. We do not need any extra rule for the positive occurrences of the $\Box$ operator, since these are taken into account by the computation of $S_{x \to y}^M$ of $(\Box^-)$. $(\exists^+)$ deals with constraints of the form $x : \exists R.C$ in a similar way. The additional side conditions on $(\exists^+)$ and $(\Box^-)$ are introduced in order to ensure a terminating proof search, by implementing the standard *blocking* technique described below. Intuitively, they are applied to constraints $x : \exists R.C$ and $x : \neg \Box \neg C$, respectively, only if x *is not blocked*, i.e. if there is no label (*witness*) z, labelling the same concepts of x, such that the rule has been already applied to $z : \exists R.C$ (resp. $z : \neg \Box \neg C$). This is formally stated in Definition 15 below;

- the (*cut*) rule ensures that, given any concept $C \in \mathcal{L}_\mathbf{T}$, an open branch built by $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ contains either $x : \Box \neg C$ or $x : \neg \Box \neg C$ for each label x: this is needed in order to allow $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ to check the minimality of the model corresponding to the open branch, as we will discuss later.

All the rules of the calculus copy their principal formulas, i.e. the formulas to which the rules are applied, in all their conclusions. As we will discuss later, for the rules $(\exists^+)$, $(\forall^-)$ and $(\Box^-)$ this is used in order to apply the blocking technique, whereas for the rules $(\exists^-)$, $(\forall^+)$, $(\sqsubseteq)$, and (*cut*) this is needed in order to have a complete calculus. Rules for $\sqcap$, $\sqcup$, $\neg$, and $\mathbf{T}$ also copy their principal formulas in their conclusions for uniformity's sake.

In order to ensure the completeness of the calculus, the rules of $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ are applied with the following *standard strategy*:

1. apply a rule to a label x only if no rule is applicable to a label y such that $y \prec x$;
2. apply dynamic rules only if no static rule is applicable.

The calculus so obtained is sound and complete with respect to the semantics in Definition 12.

Definition 15 (Witness and Blocked label). Given a constraint system $\langle S \mid U \rangle$ and two labels x and y occurring in S, we say that x is a witness of y if the following conditions hold:

1. $x \equiv_S y$;
2. $x \prec y$;
3. there is no label z s.t. $z \prec x$ and z satisfies conditions 1. and 2., i.e., x is the least label satisfying conditions 1. and 2. w.r.t. $\prec$.

We say that y is *blocked* by x in $\langle S \mid U \rangle$ if y has witness x.

By the strategy on the application of the rules described above and by Definition 15, we can prove the following Lemma:

Lemma 2. *In any constraint system $\langle S \mid U \rangle$, if x is blocked, then it has exactly one witness.*

As mentioned above, we apply a standard *blocking* technique to control the application of the rules $(\exists^+)$ and $(\Box^-)$, in order to ensure the termination of the calculus. Intuitively, we can apply $(\exists^+)$ to a constraint system of the form $\langle S, x : \exists R.C \mid U \rangle$ only if x *is not blocked*, i.e. it does not have any witness: indeed, in case x has a witness z, by the strategy on the application of the rules described above the rule $(\exists^+)$ has already been applied to some $z : \exists R.C$, and we do not need a further application to $x : \exists R.C$. This is ensured by the side condition on the application of $(\exists^+)$, namely if $\nexists z \prec x$ such that $z \equiv_{S, x : \exists R.C} x$. The same blocking machinery is used to control the application of $(\Box^-)$, which can be applied only if $\nexists z \prec x$ such that $z \equiv_{S, x : \neg\Box\neg C} x$.

4.2. The tableau calculus $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$

Let us now introduce the calculus $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ which, for each open branch $\mathbf{B}$ built by $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$, verifies if $\mathcal{M}^{\mathbf{B}}$ is a minimal model of the KB.

Definition 16. Given an open branch $\mathbf{B}$ of a tableau built by $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$, we define:

- $\mathcal{D}(\mathbf{B})$ as the set of labels occurring on $\mathbf{B}$;

- $\mathbf{B}^{\Box^-} = \{x : \neg\Box\neg C \mid x : \neg\Box\neg C \text{ occurs in } \mathbf{B}\}$.

A tableau of $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ is a tree whose nodes are triples of the form $\langle S \mid U \mid K \rangle$, where $\langle S \mid U \rangle$ is a constraint system, whereas K contains formulas of the form $x : \neg\Box\neg C$, with $C \in \mathcal{L}_{\mathbf{T}}$.
The basic idea of $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ is as follows. Given an open branch $\mathbf{B}$ built by $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ and corresponding to a model $\mathcal{M}^{\mathbf{B}}$ of KB $\cup \{\neg F\}$, $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ checks whether $\mathcal{M}^{\mathbf{B}}$ is a minimal model of KB by trying to build a model of KB which is preferred to $\mathcal{M}^{\mathbf{B}}$. Starting from $\langle S \mid U \mid \mathbf{B}^{\Box^-} \rangle$ where $\langle S \mid U \rangle$ is the constraint system corresponding to the initial KB $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ tries to build an open branch containing all and only the labels appearing on $\mathbf{B}$, i.e. those in $\mathcal{D}(\mathbf{B})$, and containing less negated boxed formulas than $\mathbf{B}$ does. To this aim, first the dynamic rules use labels in $\mathcal{D}(\mathbf{B})$ instead of introducing new ones in their conclusions. Second the negated boxed formulas used in $\mathbf{B}$ are stored in the additional set K of a tableau node, initialized with $\mathbf{B}^{\Box^-}$. A branch built by $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ closes if it does not represent a model preferred to the candidate model $\mathcal{M}^{\mathbf{B}}$, and this happens if the branch contains a contradiction (Clash) or it contains at least all the negated boxed formulas contained in $\mathbf{B}$ ((Clash)$_{\Box^-}$ and (Clash)$_\emptyset$).
More in detail, the rules of $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ are shown in Figure 3. The rule $(\exists^+)$ is applied to a constraint system containing a formula $x : \exists R.C$; it introduces $x \xrightarrow{R} y$ and $y : C$ where $y \in \mathcal{D}(\mathbf{B})$, instead of y being a new label. The choice of the label y introduces a branching in the tableau construction. The rule $(\sqsubseteq)$ is applied in the same way as in

Figure 3: The calculus $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$. To save space, we omit the rules $(\sqcup^+)$ and $(\sqcup-)$.

$\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ to *all the labels of* $\mathcal{D}(\mathbf{B})$ (and not only to those appearing in the branch). The rule $(\square^-)$ is applied to a node $\langle S, x : \neg\square\neg C \mid U \mid K\rangle$, when $x : \neg\square\neg C \in K$, i.e. when the formula $x : \neg\square\neg C$ also belongs to the open branch $\mathbf{B}$. In this case, the rule introduces a branch on the choice of the individual $v_i \in \mathcal{D}(\mathbf{B})$ which is preferred to x and is such that C and $\square\neg C$ hold in v_i. In case a tableau node has the form $\langle S, x : \neg\square\neg C \mid U \mid K\rangle$, and $x : \neg\square\neg C \notin \mathbf{B}^{\square^-}$, then $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ detects a clash, called $(\text{Clash})_{\square^-}$: this corresponds to the situation in which $x : \neg\square\neg C$ does not belong to $\mathbf{B}$, while $S, x : \neg\square\neg C$ is satisfiable in a model $\mathcal{M}$ only if $\mathcal{M}$ contains $x : \neg\square\neg C$, and hence only if $\mathcal{M}$ is *not* preferred to the model represented by $\mathbf{B}$.

The calculus $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ also contains the clash condition $(\text{Clash})_\emptyset$. Since each application of $(\square^-)$ removes the principal formula $x : \neg\square\neg C$ from the set K, when K is empty all the negated boxed formulas occurring in $\mathbf{B}$ also belong to the current branch. In this case, the model built by $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ satisfies the same set of negated boxed formulas (for all individuals) as $\mathbf{B}$ and, thus, it is not preferred to the one represented by $\mathbf{B}$.

$\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ always terminates. Intuitively, termination is ensured by the fact that dynamic rules make use of labels belonging to $\mathcal{D}(\mathbf{B})$, which is finite, rather than introducing "new" labels in the tableau.

Definition 17. Let KB be a knowledge base whose corresponding constraint system is $\langle S \mid U\rangle$. Let F be a query and let S' be the set of constraints obtained by adding to S the constraint corresponding to $\neg F$. The calculus $\mathcal{TAB}_{min}^{\mathcal{ALC}+\mathbf{T}}$ checks whether a query F can be minimally entailed from a KB by means of the following procedure:

- the calculus $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ is applied to $\langle S' \mid U\rangle$;

- if, for each branch $\mathbf{B}$ built by $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$, either:

  (i) $\mathbf{B}$ is closed or

  (ii) the tableau built by the calculus $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ for $\langle S \mid U \mid \mathbf{B}^{\square^-}\rangle$ is open,

then the procedure says YES

else the procedure says NO

The following theorem shows that the overall procedure is sound and complete.

Theorem 3 (Soundness and completeness of $\mathcal{TAB}_{min}^{\mathcal{ALC}+\mathbf{T}}$). *$\mathcal{TAB}_{min}^{\mathcal{ALC}+\mathbf{T}}$ is a sound and complete decision procedure for verifying if $KB \models_{min}^{\mathcal{L}_{\mathbf{T}}} F$.*

We provide an upper bound on the complexity of the procedure for computing the minimal entailment $KB \models_{min}^{\mathcal{L}_{\mathbf{T}}} F$:

Theorem 4 (Complexity of $\mathcal{TAB}_{min}^{\mathcal{ALC}+\mathbf{T}}$). *The problem of deciding whether $KB \models_{min}^{\mathcal{L}_{\mathbf{T}}} F$ is in $\text{CO-NE}\textsc{xp}^{\text{NP}}$.*

22

5. Background

With the increasing accessibility of multi–core processors and distributed computing networks, it has become imperative to investigate novel ways of using these concurrency and distribution technologies to tackle hard problems and to tap latent distribution and collaborative problem solving opportunities present in various tasks that computers are being used to solve.

Theorem proving, with its vast combinatorially explosive search spaces and increasingly complex and bigger problems, can benefit from these new ways of programming and new technologies[?].

This work investigates the application of a concurrent/collaborative approach to a pre–existing theorem prover called PreDeLo.

6. The previous theorem prover: PreDeLo

PreDeLo is a theorem prover for Preferential Description Logics, non–monotonic extensions of standard DLs based on a typicality operator $\mathbf{T}$, which enjoys a preferential semantics.

The core of the software is a Prolog implementation of labelled tableaux calculi for such extensions, which is able to deal with $\mathcal{ALC} + \mathbf{T}_{min}$ (the preferential extension of the basic DL $\mathcal{ALC}$), as well as with the preferential extension of the lightweight DL $DL - Lite_{core}$. The Prolog implementation is inspired by the "lean" methodology, whose basic idea is that each axiom or rule of the tableaux calculi is implemented by a Prolog clause of the program. Concerning $\mathcal{ALC}$, *PreDeLo* considers two extensions based, respectively, on Kraus, Lehmann and Magidor's preferential and rational entailment.

For a complete description of PreDeLo, the reader should refer to [?]: in the sequel we will sketch its general approach to proving and highlight where we changed the implementation in order to distribute the calculus.

In [?], the tableaux calculus for checking entailment in the rational extension of $\mathcal{ALC}$ is also introduced.

6.1. The tableaux calculi's implementation

6.1.1. Operators

This implementation makes use of different operators that have been defined to represent the connectives allowed in $\mathcal{ALC} + \mathbf{T}_{min}$:

- negation ($\neg$) is represented by **neg**;

- modality ($\Box$) is represented by **box**;

- typicality ($\mathbf{T}$) is represented by **ti**;

- universal quantification ($\forall$) is represented by **fe**;

- existential quantification ($\exists$) is represented by **ex**;

- intersection ($\sqcap$) is represented by **and**;

- disjunction ($\sqcup$) is represented by **or**;

- inclusion ($\sqsubseteq$) is represented by **inc**.

The operators are listed in decreasing priority order. Prolog allows to define operators in prefix, postfix and infix precedence. Since roles are binary operators expressed in the form $\exists r.C$, the direct translation into Prolog code of such a statement should look like

```
r ex C
```

Since this notation is not natural, potentially error prone and for sake of code readability, the operator **in** was introduced. The operator **in** was associated to the highest priority and at the same time the quantifiers were kept as infix operators: it can be seen as an infix operator associated with quantifiers $\forall$ and $\exists$. Using **in**, statements like the one mentioned before look like:

```
ex r in C
```

6.1.2. Labels
Labels are defined as two elements lists, like for example:

```
[adam; 1]
```

where the first element represents the individual, while the second is its "age". Intuitively, if two labels have ages M and N, with $M < N$, it means that the label with age M has been introduced before the one with age N. Furthermore, two labels with the same "name" but different ages are considered as different labels.

Let us make some examples:

- `[adam; 1]` and `[john; 2]` are two different labels where `[adam; 1]` has been introduced before `[john; 2]`, so we say that `[adam; 1]` is older than `[john; 2]`;

- `[adam; 1]` and `[adam; 2]`, despite the same "name", are two different labels.

Resuming, we say that a label is identified by both the name and its age.

The aging mechanism was introduced for two reasons. First, it allows to easily treat the label generation in the dynamic rules of the calculus. Second, it is used in the blocking mechanism to decide whether a rule should be applied or not (e.g., the $\Box^-$ rule). As it should be clear, at the beginning all labels have age 1. Dynamic rules like $\Box^-$, $\forall^-$ and $\exists^+$ generate new labels using the support predicate `generateLabels(...)`.
The theorem prover keeps a list of current labels in memory.
When invoked, the predicate `generateLabels` checks this list and returns a new label using the following logic:

- let N be the highest age value in the labels list, a new age $M = N + 1$ is generated;

- label names are generated using letters x, y, z, i, j, k;

- if all of them are already used, the name (e.g., x) is kept while the age is increased (as we said, labels are identified by both name and age).

So that if we had a list of labels

```
[[x; 1]; [y; 1]; [z; 2]]
```

the next label would be

```
[i; 3]
```

while if we had a list like

```
[[x; 1]; [y; 1]; [z; 2]; [i; 3]; [j; 4]; [k; 5]]
```

the next generated label would be

```
[x, 6]
```

This mechanism was chosen to keep the labels readable to the final user. Generally, if we introduced six new labels, it would mean that the dynamic rules had been called six times and a complex proof tree already exists.

6.1.3. Formulas

As presented in the previous chapters, $\mathcal{ALC} + \mathbf{T}_{min}$ formulas represent concepts associated with individuals or concepts relations. More in details, formulas (concepts) with labels (individuals), are the elements that we find in the ABox, while the concepts relations can be found in the TBox. We can distinguish between two types of formulas: the label-concept formulas and the role formulas.

- A label-concept formula $x : F$, where x is the label and F the formula, is represented as a couple (list of two elements):

    ```
    [x; F]
    ```

- A role formula $x \xrightarrow{r} y$, where x and y are labels and r is a role, is represented as a a list of three elements:

    ```
    [x; r; y]
    ```

- A label-order formula $x < y$, where x and y are labels, is represented as a singleton, where labels are separated by the order relation symbol $<$ as in:

    ```
    [x < y]
    ```

Resuming, a label-concept formula as $x : (Student \sqcup Worker) \sqsubseteq TaxPayer$ will be represented as:

```
[[x;N]; Student and Worker inc TaxPayer]
```

where N is a natural number representing the "age" of the formula, which depends on the context. In $\mathcal{ALC} + \mathbf{T}_{min}$ a sequent is composed by two lists:

- list S that initially represents the ABox;

- list U that initially represents the TBox.

The list S is composed of label-concept formulas, as the one presented in the last example. The list U contains formulas describing relations between concepts, e.g. $Worker \sqsubseteq TaxPayer$.

Each element in list U is associated to another list that contains labels. This is because the precondition (see figure 4) for applying rule $\sqsubseteq$ (unfold) in the calculus states that a formula can be unfolded on a label, unless it has already been done for that label.

$$\frac{\langle S, \, x \, : \, C \sqcap D \, | \, U \rangle}{\langle S, \, x \, : \, C \sqcap D, \, \, x \, : \, C, \, x \, : \, D \, | \, U, \, C \sqsubseteq D^{L,x} \rangle} \; (\sqcap^{+})$$

if $\{x : C, \, x : D\} \nsubseteq S$.

Figure 4: $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ rule for $\sqsubseteq$ and its precondition.

In the Prolog implementation, if we had $Worker \sqsubseteq TaxPayer$, and the $\sqsubseteq$ (unfold) rule had already been applied on labels `[adam; 1]` and `[john; 1]`, the corresponding list would be:

$$[...; [Worker \sqsubseteq TaxPayer; [[adam; 1]; [john; 1]]]; ...]$$

6.1.4. Structure

The theorem prover is separated into modules to improve code readability and maintainability. The modules are defined as follows:

- module `operators`, contains the operators definition as described previously;

- module `alct1`, is the main module implementing the $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ calculus.

  This module exposes the top-level predicate `prove(S,U,F,Tree)` that is called to start the execution of the proof method;

- module `alct2` implements $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$. It contains the definition of a top level predicate called `phase2(...)`.

- module `helpers`, contains different support predicates used during proof computation, e.g. example the predicate `generateLabels(...)` described above.

The top-level predicate `prove(S,U,F,Tree)` is declared in the module `alct1`:

- `S` is the list representing the ABox of our knowledge base;

- `U` is the TBox;

- `F` is the query formula;

- `Tree` is a Prolog term representing the proof tree (the proof tree structure and construction will be discussed in the sequel).

The first three parameters must be specified by the user, while `Tree` is an output term.

Before starting the proof computation, `prove(S,U,F,Tree)` negates the query formula `F` and adds it to the `S` list, constructs the list of labels and extracts other useful information. Once these pieces of information are collected in the internal variables, the "non–top–level" `prove(...)` predicate is called.
This predicate has the following structure:

$$\text{prove(S,U,Lt,Labels,ABox,Tree)}$$

where

- `U` is the same list as before.

- `S` is the previous list except for the fact that it now contains the formula $\neg F$.

- `Lt` is the list of concept of which we want to minimize the atypical instances.

- `Labels` is the list of labels presented in the previous section.

- `ABox` is the ABox as it is before starting the computation (we keep it since it is necessary in second phase of the calculus).

- `Tree` is the output term representing the proof tree.

Each rule in the tableaux is associated with a `prove(...)` predicate that computes that rule and calls itself recursively. Let's consider the example of the rule shown in figure 5.

$$\frac{\langle S, x : C \sqcup D \,|\, U \rangle}{\langle S, x : C, x : C \sqcup D \,|\, U \rangle \qquad \langle S, x : D, x : C \sqcup D \,|\, U \rangle} \, (\sqcup^+)$$

$$\text{if } x : C \notin S \text{ and } x : D \notin S.$$

Figure 5: $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$ rule for $\sqcup^+$ and its precondition.

This is how the theorem prover works on such a rule:

- First checks if list S contains a formula $x : C \sqcup D$;

- If no, it skips the rule and try another one.

- If yes, checks the precondition specified in figure 5

- If the precondition is satisfied, it calls prove(...) recursively two times, one for each sub-tree.

This is how the Prolog code looks like:

Listing 1: alct1.pl

```
184  /* or + */
185  prove(S,U,Lt,Labels,F,ABox,NL) :-
186  member([X,C or D],S),
187  n +(member([X,C],S)),
188  n +(member([X,D],S)),
189  (* build the lists for the left and right sub-trees *)
190  prove(S11,U,Lt,Labels,F,ABox,NL1),
191  prove(S22,U,Lt,Labels,F,ABox,NL1).
```

There is a small difference between how static and dynamic rules are implemented. Static rules always introduce at most two branches in the proof tree (like in the previous example). This is not true for dynamic rules where the number of branches varies depending on the number of labels present in the tree. Figure 6 shows an example.

$$\frac{\langle S,\, x : \neg\Box\neg C \,|\, U\rangle}{\langle S, x : \neg\Box\neg C, y < x, y : C, y : \Box\neg C, S^M_{x\to y}|U\rangle \quad \langle S, x : \neg\Box\neg C, v_1 < x, v_1 : C, v_1 : \Box\neg C, S^M_{x\to v_1}|U\rangle \,\dots} \;(\Box^-)$$

<div align="right">

with y fresh,

if $\not\exists z \prec x$ s.t. $z \equiv_{S,x:\neg\Box\neg C} x$ and $\not\exists u$ t.c. $\{u < x, u : C, u : \neg\Box\neg C, S^M_{x\to u}\} \subseteq S$

$\forall v_i$ occurring in S.

</div>

Figure 6: $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$ rule for $\Box^-$ and its precondition.

An additional support predicate for each dynamic rule was introduced to manage the variable number of recursive calls, called `proveRecBox(...)`.

As described in chapter 4, the calculus is composed of two phases. The first phase is implemented in the Prolog module `alct1` and can be started invoking predicate `prove(S,U,F,Tree)`. The Prolog interpreter, then proceeds trying to apply rules of the first phase (corresponding to $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$) until the theorem is proved or in case an unclosed branch in the proof tree is found (so that no other rule in phase one can be applied).

In this case, phase two calculus must be undertaken for the open branch. Implementation $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH2}$ is coded in module `alct2` which exposes the `prove_phase2(...)` predicate.

7. Architectural design

The main goal of our work is to improve the performances of the previously implemented theorem prover. In the previous implementation, especially when using $\mathcal{ALC} + \mathbf{T}_{min}$, the extremely high complexity of the calculus itself [3] could delay the result of several minutes even for a medium length query formula. We aim to build a distributed application for theorem proving so that, in a distributed environment, we can spread the computational complexity of the calculus amongst different machines.

$\mathcal{TAB}_{min}^{\mathcal{ALC}+\mathbf{T}}$ is designed as a two–phase calculus so, intuitively, there is no need for the first phase of the calculus to wait the result of the elaboration of a branch by the second phase before launching another one. Indeed, in order to prove whether a query entails from a KB, we can start computing the first phase on a machine and, in case the branch is not closed (we will call it *parallel branch*), launch the second phase on a different machine. Meanwhile, the main machine, instead of waiting termination (as it happens in a non–distributed environment) can consider the *parallel branch* as "suspended" and continue with the computation of another branch of the first phase. If, during the computation, a *parallel branch* turns out to be open, its status is changed from "suspended" to closed and the computation can continue. If a *parallel branch* turns out to be closed, the formula cannot be satisfied, so the computation process is interrupted. If the main machine reaches the end of its computation, it has to wait the "answers" of the parallel branches:

- if all of them result as open then the theorem prover answers *yes* (which means the query entails from the KB);

- if any of them results as closed, the answer is *no* (which means the query does not entail from the KB).

In order to describe the software architecture that we have developed, we refer to the *worker—employer* metaphor.
The *employer* implements the first phase of the calculus, while the *workers* implement the second phase of the calculus.
Initially, each *worker* registers to the *employer*. When, during the computation of the first phase calculus, the *employer* needs to verify a *parallel branch*, it can delegate the second phase calculus to one of the registered *workers*, and consequently carry on with its calculations. When a *worker* terminates its execution, reports its result back to the *employer*. The *employer*, at the end of the first phase, collects all the results and yields the definitive answer (*yes* or *no*).

7.1. Implementation

7.2. Architecture

8. Conclusion

---

[3] In Theorem 4 we showed that the problem of deciding whether $KB \models_{min}^{\mathcal{L}_{\mathbf{T}}} F$ is in CO-NEXP$^{\text{NP}}$.