Università degli studi di Torino

Tesi di Laurea Magistrale

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

Candidato: Luca Violanti

Relatore: Gian Luca Pozzato

Controrelatore: Alberto Martelli

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 $\mathcal{ALC}$, 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 $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{min}$, 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.

Contents

Chapter 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

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, and are 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 more detailed description can be found in chapter 2.

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.

  For instance, we can state that penguins are birds

  $$Penguin \sqsubseteq Bird$$

  which means that "every individual which is a member of the concept *Penguin*, is also a member of the concept *Bird*".

- the ABox containing *facts* of the domain of interest, such as memberships of individuals to concepts (e.g. $Bird(tweety)$), and relationships between individuals (e.g. $Hunts(sylvester, tweety)$).

This allows us to draw the conclusion that if an individual named Tux is a penguin, it is also a bird.

$$TBox \sqcap ABox \sqcap \{Penguin(tux)\} \models Bird(tux)$$

The "traditional" DLs are however expressively limited since they do not allow the representation of typical properties of concepts, nor to make an efficient use of defeasible inheritance. 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 arises. For instance, since one of the main practical applications of the DLs is the definition and use of medical taxonomies, let us consider a case of defeasible inheritance. In a "normal" patient, the heart is situated more in the left side of the thorax than in the right.

$$Heart \sqsubseteq \exists Position.Left$$

The *exception* to this is constituted by patients born with the condition known as *situs inversus*, who have their major visceral organs reversed or mirrored from their normal positions. In case of exceptions, such as this, with the 'traditional" DLs, the KB becomes trivial.

Let us introduce another limit with an example:

$$Penguin \sqsubseteq Bird$$
$$Bird \sqsubseteq FlyingAnimal$$
$$Penguin \sqsubseteq \neg FlyingAnimal$$

This KB is consistent only if there are no individuals which are penguins.

Non–monotonic reasoning and typicality

The traditional approach to overcome these limits is to handle defeasible inheritance by integrating DLs with some kind of non–monotonic reasoning machinery. This is presented extensively in section 2.5.

Chapter 3 describes another approach, based on a typicality operator ($\mathbf{T}$), to build non–monotonic logics. We focus on $\mathcal{ALC} + \mathbf{T}$ (presented in section 3.1) and $\mathcal{ALC} + \mathbf{T}_{min}$ (section 3.2), which are the base for our work.
In brief, $\mathcal{ALC} + \mathbf{T}_{min}$ allows us to consistently represent the following TBox:

$$\mathbf{T}(Student) \sqsubseteq \neg IncomeTaxPayer$$
$$\mathbf{T}(Student \sqcap Worker) \sqsubseteq IncomeTaxPayer$$
$$\mathbf{T}(Student \sqcap Worker \sqcap \exists HasAChild.\top) \sqsubseteq \neg IncomeTaxPayer$$

These axioms model the following assertions: the typical students do not have to pay income taxes, whereas typically a student which is also a worker does.
Moreover, we suppose the existence of a tax exemption for the typical students which have a job and also are fathers.

Note that such a KB, without the typicality operator $\mathbf{T}$, would be consistent only in case no individual was both a student and a worker.

Let us now consider the following ABox:

$$Student(mario)$$
$$\exists HasAChild.Student(franco)$$

In $\mathcal{ALC} + \mathbf{T}_{min}$, from the $KB = ABox, TBox$ we can infer that

$$\neg IncomeTaxPayer(mario)$$

while

$$\exists HasAChild.IncomeTaxPayer(franco)$$

This shows us that an individual has the properties of the *most specific* concept to which it belongs. Unlike what happens with other languages, in $\mathcal{ALC} + \mathbf{T}_{min}$ the inferences are correctly applied both to the individuals explicitly nominated in the ABox (e.g. Mario) and those introduced by the use of quantifiers (e.g. Franco's son).

$\mathcal{ALC} + \mathbf{T}_{min}$ is also non–monotonic: a formula F which can be inferred from a KB is not necessarily inferable from another $KB' \supset KB$.
Following the example, if we added to the ABox the information

$$Worker(mario)$$

the previous conclusion ($\neg IncomeTaxPayer(mario)$ would not be inferable anymore. On the other hand we could infer

$$IncomeTaxPayer(mario)$$

Furthermore, $\mathcal{ALC} + \mathbf{T}_{min}$ can deal with a form of *irrelevance*.
From the previous KB we are able to infer also that

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

since the characteristic of being tall is *irrelevant* to being or not being a taxpayer.
Even if we added to the ABox

$$Tall(mario)$$

all the previous inferences involving Mario and him being a taxpayer would still be valid.

The tableaux calculus $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{min}$

Our implementation is based on the tableaux proof method, which is a mechanism that allows to demonstrate the validity of a formula in a given logical system.
In chapter 4 we introduce $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{min}$ (the tableaux calculus for $\mathcal{ALC}+\mathbf{T}_{min}$), which forms the basis of our theorem prover. This tableaux calculus allows us to conclude whether a query (F) is *minimally* entailed by a KB ($KB \models_{min} F$) or not, just by applying the various computation rules, i.e. without appealing to the semantics.
This is performed as a proof by contradiction: $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{min}$ tries to find a *counterexample* (a model of $KB \cup \neg F$) by generating of various candidate models and then verifying if they are *minimal*. The calculus requires two phases:

- a first phase, $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$ (presented extensively in section 4.1), which can generate various branches (corresponding to various candidate models);
- a second phase, $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH2}$ (section 4.2), which verifies the minimality of such candidate branches.

Chapter 5 shows how all these mechanisms have been put into practice by a previous theorem prover called *PreDeLo* (section 5.1), while section 5.2 extensively describes how $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{min}$ was implemented using the logic programming language SICStus Prolog.
PreDeLo is the first implementation of a theorem prover for $\mathcal{ALC}+\mathbf{T}_{min}$: when presented with a query, it runs the first phase of the calculus, and whenever the first phase generates a candidate branch, it interrupts the execution of the first phase to verify whether such branch is a counterexample, using the second phase. Only when the branch has been checked, the execution of the first phase can be resumed. This constitutes the main limitation of *PreDeLo*: the first phase is in fact independent from the second, and the two phases could be run in parallel.

DysToPic

Our work aims to improve *PreDeLo* by tackling this limitation: to do so we developed a new theorem prover named *DysToPic* (illustrated in chapter 6), which is a distributed software, capable of exploiting multiple computing units in parallel to perform its inferences.
The approach is described first at a higher level in section 6.4, and then detailed more techically in section 6.5. An example of execution is also presented in 6.6.

Preliminary tests

Chapter 7 is dedicated to the description of a series of preliminary performance tests on our software. We expect our solution to outperform the previous, at least in some cases, as explained in section 7.2. For example if, in a particular proof, even a single branch generated by the first phase required a long verification by the second phase, while a sequential software would be forced to wait such verification to end, our parallel approach could simultaneously perform a certain number of other verifications on other branches. This is especially useful in case of queries which have "NO" as an answer, i.e. a case in which the query does not entail from the KB (see "case 2" in 7.2): when the execution of the second phase on a branch answers `"false"`, it means the whole entailment does not hold. Verifying such a branch, therefore leads to an immediate result ("NO"), exempting the prover from verifying other branches. We expect *DysToPic* to take great advantage of its parallelism, for instance, in this case.

Chapter 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 simpler language (which comes with lower computational complexity) rather than a more expressive one (high computational complexity) depends on how complex we want our world's description to be.

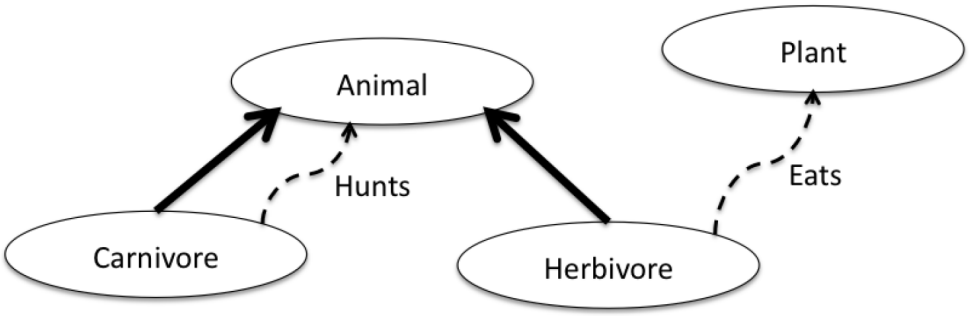

Figure 2.1: An example of network, showing relationships between concepts.

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 $Plant$.

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 knowledge based 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 [26].

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 [4].

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, and contains inclusions or definitions of roles.

As an example of TBox, here we have a fraction of a hierarchy of animals. To represent that the penguins are birds and that both the birds and the cats are animals, we have

$$Penguin \sqsubseteq Bird$$
$$Bird \sqsubseteq Animal$$
$$Cat \sqsubseteq Animal$$

The ABox contains assertions about named individuals in terms of this vocabulary.

For instance, an ABox can contain the following facts

$$Bird(tweety)$$
$$Cat(sylvester)$$
$$Hates(sylvester, tweety)$$

representing that Tweety is a bird, Sylvester is a cat, and that Sylvester hates Tweety, respectively.

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 $\mathcal{AL}$ (Attributive concept Language), that has been introduced in [32] as a minimal language that is of practical interest. The other languages of this family are extensions of $\mathcal{AL}$.

2.3.1 Syntax

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)

2.3.2 Semantics

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 $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ (the set of the individuals of the domain which belong to such concept), and to every atomic role R a binary relation, $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ (which is a couple of elements of the domain).

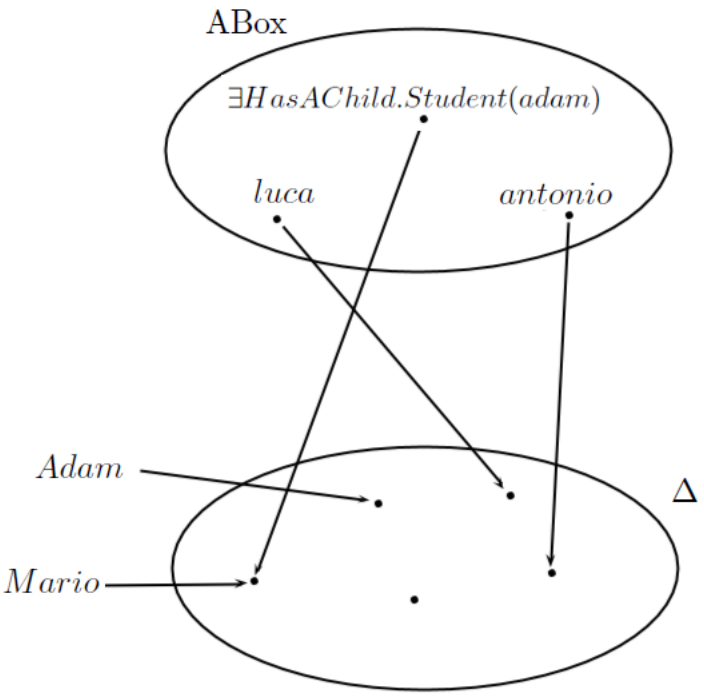

Figure 2.2: The arrows represent the assignments established by $\mathcal{I}$

For instance if we have a domain that contains some individuals, some are birds and some are not:

$$Bird \xrightarrow{\mathcal{I}} Bird^{\mathcal{I}}$$

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 $\mathcal{I}$ [4].

A model is a structure $\mathcal{M} = \langle \Delta, \mathcal{I} \rangle$, where:

- Δ is the domain;
- $\mathcal{I}$ is the extension function that maps each extended concept C to $C^{\mathcal{I}} \subseteq \Delta$, every atomic concept $A \in C$ to a set $C^{\mathcal{I}} \subseteq \Delta$, and each role R to a $R^{\mathcal{I}} \subseteq \Delta \times \Delta$.

The models attribute a meaning to the inclusions: a certain inclusion $C \sqsubseteq D$ holds in a model $\mathcal{M}$ if $C^{\mathcal{I}} \sqsubseteq D^{\mathcal{I}}$.
For instance, the model $\mathcal{M}_1$ confirms the inclusion $Penguin \sqsubseteq Bird$ since the extension of the penguins ($Penguin^{\mathcal{I}}$) is included in the extension of the birds ($Bird^{\mathcal{I}}$).

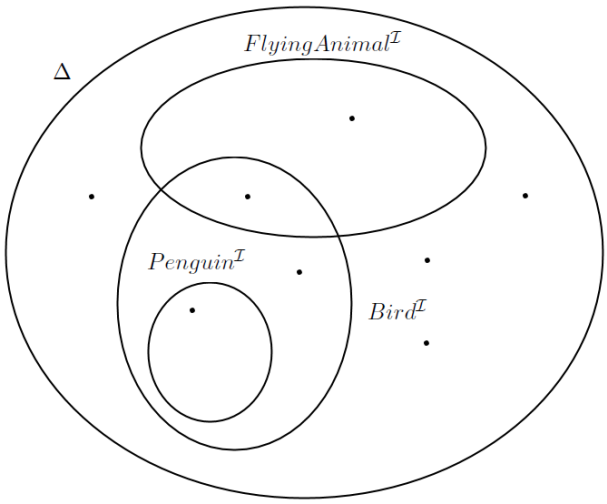

Figure 2.3: The model $\mathcal{M}_1$

If we had another model made as $\mathcal{M}_2$, in which some of the penguins are not birds, the inclusion would not hold.

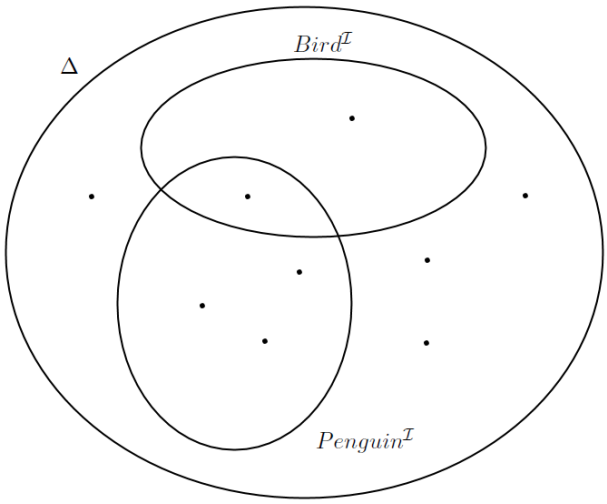

Figure 2.4: The model $\mathcal{M}_2$

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

A basic extension of $\mathcal{AL}$ is $\mathcal{ALC}$ (Attributive concept Language with Complements), which introduces (see [28]) the negation of an arbitrary concept:

$\neg C$ (concept negation)

whose semantics is:

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

This allows us to express notions such as:

$$Penguin \sqsubseteq \neg FlyingAnimal$$

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 [2, 3, 5, 8, 10, 11, 12, 37].
However, finding a suitable non–monotonic extension for inheritance reasoning with exceptions is far from obvious. To give a brief account, [2] 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, [2] 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 [37, 3].
A more general approach is undertaken in [11], 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 [6] 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 [21].
In [8, 9] a non–monotonic extension of $\mathcal{ALC}$ based on the application of Lehmann and Magidor's *rational closure* [23] to $\mathcal{ALC}$ is proposed. The approach is based on the introduction of a consequence relation $\mid\!\sim$ 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, [12] and [25]). A grounded circumscription approach for DLs with local closed world capabilities has been defined in [22].

Another approach, presented in [15] and expanded in [16] [14], [18], [17], [20], and [13], proposes the use of a typicality operator, **T**, for extending $\mathcal{ALC}$ and allow a form of non–monotonic reasoning in Description Logics.

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 present a non–monotonic extension for defeasible reasoning in description logics, called $\mathcal{ALC}+\mathbf{T}_{min}$, that extends the logic $\mathcal{ALC}+\mathbf{T}$ (which introduced the typicality operator $\mathbf{T}$).

Chapter 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 [14].

3.1.1 Syntax

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 $\mathcal{L}$.

As in standard $\mathcal{ALC}$, a knowledge base is a pair (TBox, ABox).
The 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.
The ABox contains expressions of the form $C(a)$ and aRb where $C \in \mathcal{L}$ is an extended concept, $R \in \mathcal{R}$, and $a, b \in \mathcal{O}$.

3.1.2 Semantics

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 $\mathbf{T}$ 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* $x, y, z, \ldots$;
- 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.

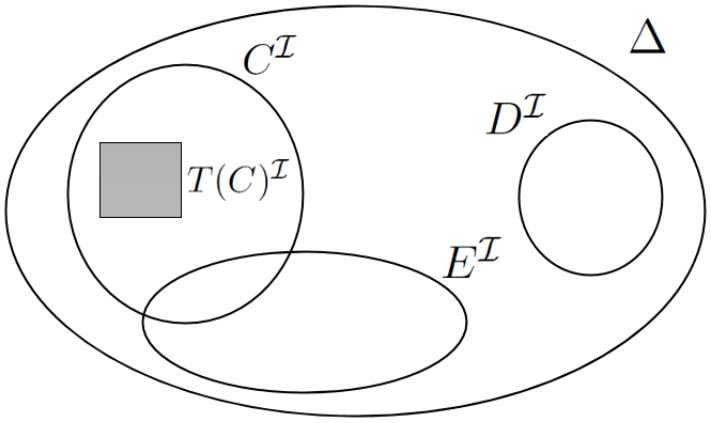

Figure 3.1: f selects the typical elements of $C^{\mathcal{I}}$ (greyed out)

In [14], it is shown that one can give an equivalent, alternative semantics for **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 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 [27] [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.

[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.

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* $y < x$*.*

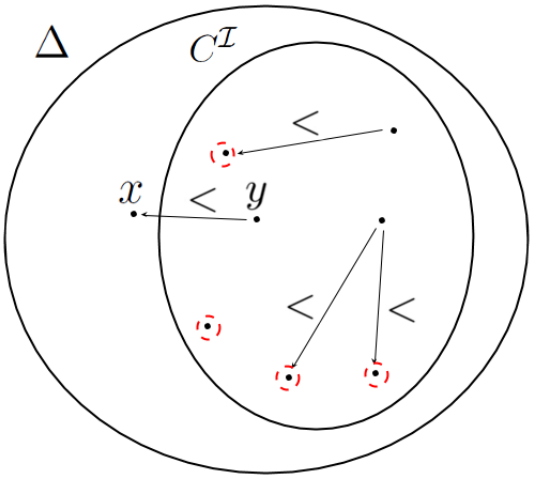

Figure 3.2: The typical individuals of C (i.e. the individuals which are $\mathbf{T}(C)$) are highlighted in the figure. x is preferred to y, but it is not a $\mathbf{T}(C)$ since it is not in $C^{\mathcal{I}}$.

The following representation theorem is proved in [14]:

Theorem 1 (Theorem 2.1 in [14])**.** *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:

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

3.1.3 Satisfiability

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 students and taxpayers, if (in addition to the TBox) the ABox only contains the following facts about Luca and Antonio:

$Student(luca)$
$Student(antonio), Worker(antonio)$

we would like to infer that Antonio pays the taxes, whereas Luca does not; but without the unique name hypothesis, we cannot draw this conclusion since Luca and Antonio 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:

1. $x \in C^I$;
2. 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 $\mathbf{T}$ (for the purpose of the calculus that we will present in Section 4), we introduce a new modality, $\Box$.

The basic idea is simply to interpret the preference relation $<$ as an accessibility relation. By the Smoothness Condition, it turns out that $\Box$ 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 $\Box$ 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:*

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

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

$$x \in (\mathbf{T}(C))^I \text{ iff } x \in (C \sqcap \Box\neg C)^I$$

Since we only use $\Box$ to capture the meaning of $\mathbf{T}$, in the following we will always use the modality $\Box$ followed by a negated concept, as in $\Box\neg C$.
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 $z \in Min_<(C^I)$.*

Proof. See [20]. □

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.*

Proof. See [20]. □

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

As mentioned in the Introduction, the logic $\mathcal{ALC} + \mathbf{T}$ presented in [14] 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:

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

Let us 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 is a 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 $\mathcal{M}^{\Box^-}_{\mathcal{L}_{\mathbf{T}}}$ 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*

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

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

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

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 $\models^{\mathcal{L}_{\mathbf{T}}}_{min} F$.

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

Example 1. Consider the following KB

$$\mathbf{T}(\mathit{Athlete}) \sqsubseteq \mathit{Confident}$$
$$\mathit{Athlete}(\mathit{john})$$
$$\mathit{Finnish}(\mathit{john})$$

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

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

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 $\mathbf{T}(\mathit{Athlete})$ (it can be easily verified that any model in which *john* is not an instance of $\mathbf{T}(\mathit{Athlete})$ 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' is

$$\mathbf{T}(Athlete) \sqsubseteq Confident$$
$$\mathbf{T}(Athlete \sqcap Finnish) \sqsubseteq \neg Confident$$
$$Athlete(john)$$
$$Finnish(john)$$

and let us add to $\mathcal{L}_{\mathbf{T}}$ the concept

$$Athlete \sqcap Finnish$$

From KB', *Confident*(*john*) is no longer derivable. Instead, we have that

$$\text{KB}' \models^{\mathcal{L}_{\mathbf{T}}}_{min} \neg Confident(john)$$

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 $\mathcal{ALC} + \mathbf{T}$ (as stated in Theorem 2).

Chapter 4

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

In this chapter we present a tableau calculus for deciding whether a query F is minimally entailed by a knowledge base $KB = (TBox, ABox)$.
This calculus, called $\mathcal{TAB}_{min}^{\mathcal{ALC}+\mathbf{T}}$, extends the calculus $T^{\mathcal{ALC}+\mathbf{T}}$ presented in [14], 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 $\neg\Box\neg C$ 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* $\Box\neg D$ *or* $\neg\Box\neg D$*, 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 analyse 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* [1]*.*
A branch is a sequence of nodes $\langle S_1 \mid U_1 \rangle, \langle S_2 \mid U_2 \rangle, \ldots, \langle S_n \mid U_n \rangle \ldots$*, 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 4.1), 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* $(Clash)_\bot$*), 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:*

- $S = \{a : C \mid C(a) \in ABox\} \cup \{a \xrightarrow{R} b \mid aRb \in ABox\}$
- $U = \{C \sqsubseteq D^{\emptyset} \mid C \sqsubseteq D \in TBox\}$

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

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.*

Proof. See [20]. □

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 [14], 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 [7], 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 $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$.

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 $S^M_{x \to y}$, that will be used in the rule $(\Box^-)$ when $y < x$, in order to introduce $y : \neg C$ and $y : \Box\neg C$ for each $x : \Box\neg C$ in the current branch:

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

$$S^M_{x \to y} = \{y : \neg C, y : \Box\neg C \mid x : \Box\neg C \in S\}.$$

4.1.1 Rules

The rules of $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ are presented in Figure 4.1.

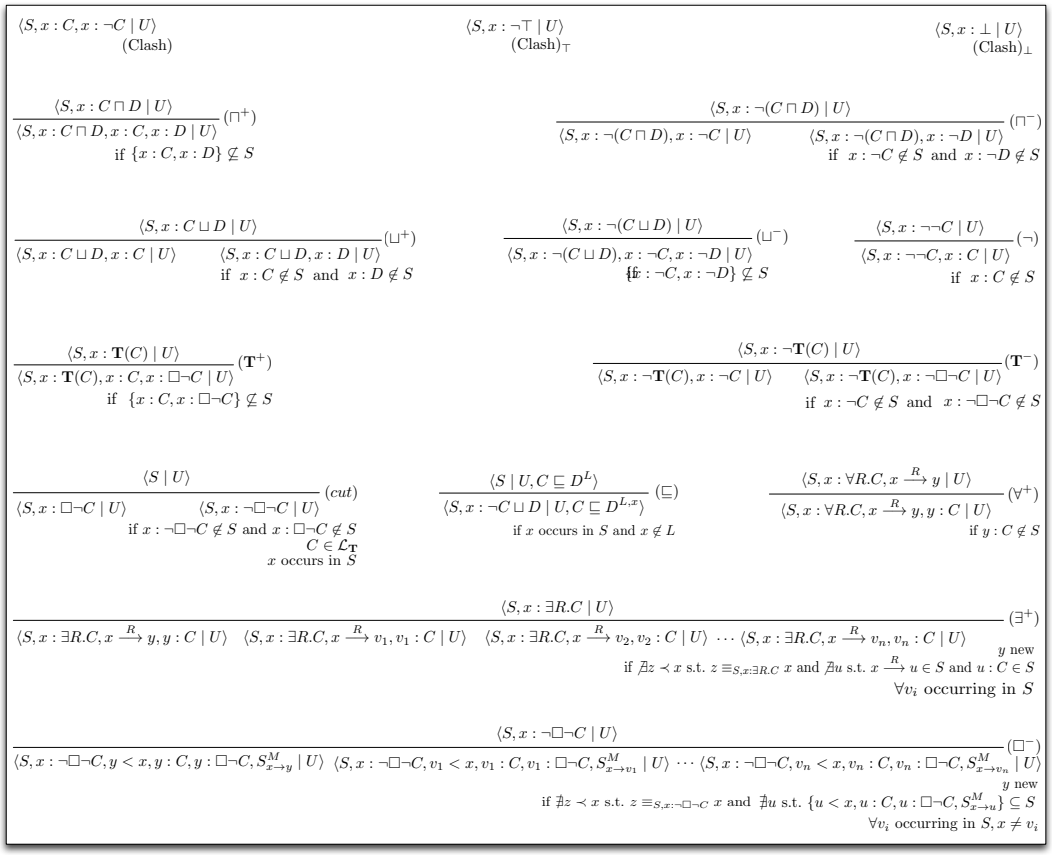

Figure 4.1: The calculus $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$. To save space, we omit the rules $(\forall^-)$ and $(\exists^-)$, dual to $(\exists^+)$ and $(\forall^+)$, respectively.

The rules $(\exists^+)$ and $(\Box^-)$ 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 $(\text{Clash})_\bot$ are used to detect *clashes*, i.e. unsatisfiable constraint systems;

- the rules for $\sqcup$, $\sqcap$, $\neg$, and $\forall$ are similar to the corresponding ones in the tableau calculus for standard $\mathcal{ALC}$ [7]: 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 $x : C \sqcup D$ 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 conditions 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 $S^M_{x\to y}$ 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 $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$ contains either $x : \Box\neg C$ or $x : \neg\Box\neg C$ for each label x: this is needed in order to allow $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH2}$ 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 presented 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.*

Proof. See [20]. □

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 $\mathbf{B}$ ($(\text{Clash})_{\Box^-}$ and $(\text{Clash})_{\emptyset}$).

4.2.1 Rules

More in detail, the rules of $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH2}$ are shown in Figure 4.2.

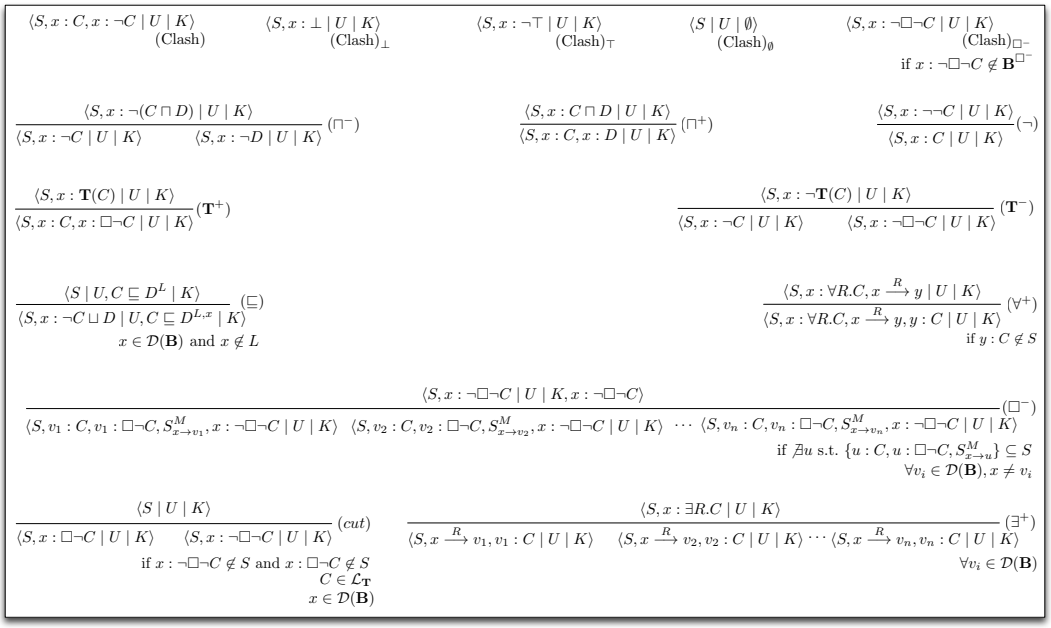

Figure 4.2: The calculus $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH2}$. To save space, we omit the rules $(\sqcup^+)$ and $(\sqcup-)$.

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

$\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH2}$ 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 $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{min}$ checks whether a query F can be minimally entailed from a KB *by means of the following procedure:*

- *the calculus $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$ is applied to $\langle S' \mid U \rangle$;*
- ***if**, for each branch $\mathbf{B}$ built by $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$, either:*
  (i) $\mathbf{B}$ is closed;
  (ii) the tableau built by the calculus $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH2}$ for $\langle S \mid U \mid \mathbf{B}^{\Box^-} \rangle$ is open;
  ***then** the procedure answers* yes*;*

***else** the procedure answers* no.

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

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

Proof. See [20]. □

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

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

Proof. See [20]. □

Chapter 5

Theorem proving for Description Logics with Typicality

5.1 The previous theorem prover: PreDeLo

PreDeLo is a theorem prover for the Preferential Description Logic $\mathcal{ALC}+\mathbf{T}_{min}$ described in section 3.2.

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 called $DL - Lite_{core}$.

The Prolog implementation is inspired by the "lean" methodology, with the idea 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 [13]: in the sequel we will sketch its general approach to prove and highlight where we changed the implementation in order to distribute the calculus.

In [13], the tableaux calculus for checking entailment in the rational extension of $\mathcal{ALC}$ is also introduced.

5.2 The tableaux calculi's implementation

5.2.1 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 would be syntactically represented as:

```
r ex C
```

Since this notation is not natural, potentially error prone and also for the 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 are syntactically represented as:

```
ex r in C
```

5.2.2 Labels

Labels are defined as two elements lists, e.g.:

```
[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 that we can state 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 its name and its age.

The ageing 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).

The theorem prover keeps a list of current labels in memory.
As it should be clear, at the beginning all labels have age 1. Dynamic rules such as $(\Box^-)$, $(\forall^-)$ and $(\exists^+)$ generate new labels using a support predicate called `gen_label/3`.
When invoked, this predicate checks this list of the current labels 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;
- in case all of them were 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.

5.2.3 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 various types of formulas:

- A label–concept formula $x : F$, where x is the label and F the formula, is represented as a 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 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:

- the list S that initially represents the ABox;
- the 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 for applying the unfolding rule ($\sqsubseteq$) in the calculus (see figure 5.1) states that a formula can be unfolded on a label, unless it has already been done for that label.

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

if x occurs in S and $x \notin L$.

Figure 5.1: $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$ rule for ($\sqsubseteq$) and its precondition.

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

```
[..., [Worker inc TaxPayer, [[adam, 1], [john, 1]]], ...]
```

5.2.4 Structure

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

- the module `operators`, which contains the definition of the operators, as previously described in 5.2.1;
- the module `alct1`, which is the main module, implementing the $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$ calculus.

  This module exposes the top–level predicate `prove(S,U,F,Tree)` that is called to start the execution of the proof method;
- the module `alct2`, which implements $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH2}$.
  It contains the definition of a top–level predicate called `prove_phase2/6`;
- the module `helpers`, which contains different support predicates used during proof computation, e.g. the predicate `genLabel/3` described in 5.2.2.

Let us further analyse the top–level predicate `prove(S,U,F,Tree)` which 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 representing the proof tree[1].

5.2.5 Execution

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 further useful information. Once these pieces of information are collected in the internal variables, the "non–top–level" predicate `prove/6` is called.

[1] *PreDeLo* uses this output term to build a graphical representation of the proof tree and visualise it on a graphical user interface.

This predicate has the following structure:

```
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/6` predicate that computes that rule and calls itself recursively.

Let us consider the example of the rule shown in figure 5.2.

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

if $x : C \notin S$ and $x : D \notin S$.

Figure 5.2: $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$ rule for $(\sqcup^{+})$ and its precondition.

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

- First, it checks if the list S contains a formula $x : C \sqcup D$;
- If the answer is *no*, it skips the rule and tries another one;
- If the answer is *yes*, it checks the precondition specified in figure 5.2;
- If the precondition is satisfied, it calls `prove/6` recursively two times, one for each conclusion.

This is translated in the following Prolog code:

Listing 5.1: alct1.pl

```
175 /* or + */
176 prove(S,U,Lt,Labels,ABOX,Tree) :-
177     member([X,C or D],S),
178     \+(member([X,C],S)),
179     \+(member([X,D],S)),!,
180     /* build the lists for the left and right sub-trees */
181     prove([[X,C]|S],U,Lt,Labels,ABOX,Tree1),!,
182     prove([[X,D]|S],U,Lt,Labels,ABOX,Tree2),!,
183     buildTree('ph1','or+',S,U,[Tree1,Tree2],Tree),!.
```

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 5.3 shows an example.

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

with y fresh,
if $\nexists z \prec x$ s.t. $z \equiv_{S,x:\neg\Box\neg C} x$ and $\nexists u$ t.c. $\{u < x, u : C, u : \neg\Box\neg C, S^M_{x\to u}\} \subseteq S$
$\forall v_i$ occurring in S.

Figure 5.3: $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$ rule for $(\Box^-)$ and its precondition.

An additional support predicate, called `proveRecB/10`, was introduced for each dynamic rule to manage the variable number of recursive calls.

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 the 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 a branch in the proof tree cannot be closed (i.e. no other rule in phase one can be applied). In this case, a verification by the second phase of the calculus ($\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH2}$) must be undertaken by such open branch.
The implementation of $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH2}$ is coded in the module `alct2`, which exposes the predicate `prove_phase2/6`.

Chapter 6

The distributed theorem prover: DysToPic

6.1 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 [36].
This work investigates the application of a concurrent/collaborative approach to the pre–existing theorem prover called *PreDeLo* (described in 5.1): the result is a new software solution, called *DysToPic*, presented in the sequel.

6.2 Our solution: the DysToPic system

The main goal of our work is to improve the performances of the previously implemented theorem prover. In the previous software *PreDeLo*, especially when using $\mathcal{ALC} + \mathbf{T}_{min}$, the combination of an extremely high complexity of the calculus [1] and the sequential implementation used to tackle it, can cause delays of several minutes before receiving a result even for small sized query formulas. As we described previously, the calculus $\mathcal{ALC} + \mathbf{T}_{min}$ is not inherently sequential and can be effectively parallelised.
We therefore aim to build a distributed application for theorem proving in $\mathcal{ALC} + \mathbf{T}_{min}$, so that the computational burden can be spread amongst different machines, which can work in parallel.

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

6.3 A two–phase calculus

For the sake of clarity we recall here that, as we described in detail in chapter 4, $\mathcal{TAB}_{min}^{\mathcal{ALC}+\mathbf{T}}$ uses a tableaux calculus to verify whether a query entails from a knowledge base, i.e. $KB \models_{min} F$, which is equivalent to state that F is verified in all the minimal models of the KB.
To do so, the calculus tries to build a model of $KB \cup \neg F$ (a counterexample). If it fails, then the query entails from the KB, otherwise it does not.

The process is designed as a two–phase calculus.
The first phase generates models of $KB \cup \{\neg F\}$. The ones in which F holds cannot be counterexamples, thus they are discarded. A model $\mathcal{M}_i$ of $KB \cup \{\neg F\}$ in which F does not hold, instead, could be a counterexample: it is considered promising and will be further verified by the second phase of the calculus.
The second phase, indeed, uses $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$: in order for a model $\mathcal{M}_i$ of $KB \cup \{\neg F\}$ to be a counterexample of the entailment, it has to be a *minimal* model of the KB. As stated in section 4.2, this means that, for each candidate model $\mathcal{M}_i$, the second phase tries to build a model of the KB, $\mathcal{M}_j^{KB}$, which is preferred to $\mathcal{M}_i$. If every attempt of the second phase fails (i.e. no $\mathcal{M}_j^{KB} \prec \mathcal{M}_i$), then that candidate $\mathcal{M}_i$ is the minimal model (i.e. the counterexample): this means that $KB \models_{min} F$. Conversely, in case even a single $\mathcal{M}_j^{KB}$ is found to be preferred to an $\mathcal{M}_i$, that $\mathcal{M}_i$ cannot be the counterexample. If every $\mathcal{M}_i$ has at least one corresponding $\mathcal{M}_j^{KB}$ so that $\mathcal{M}_j^{KB} \prec \mathcal{M}_i$, then the whole research for a counterexample has failed, which means that $KB \cup \neg F$ has no models, and $KB \models_{min} F$.

This procedure can be seen as the generation and traversal of a tree.
The first phase generates branches $\mathbf{B}_i$, corresponding to models $\mathcal{M}(\mathbf{B}_i)$, and uses $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ to accept or refuse them. A branch is considered *open* until it is verified and, if it does not represent a counterexample, it is *closed*. The branches which are still *open* after the first phase require further verification by $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$, therefore they will be processed by the second phase.
As we showed in 5.2.5, *PreDeLo* was forced to immediately verify with $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ each branch generated by $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$, sequentially, interrupting de facto $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$.
In this section, instead, we explain that there is no need for the first phase of the calculus to wait for the result of one elaboration of the second phase on an open branch, before generating another candidate branch.
Indeed, in order to prove whether a query entails from a KB, the first phase can be executed on a machine; every time that a branch remains open after the first phase, the execution of the second phase for this branch can be performed in parallel, on a different machine. Meanwhile, the main machine, instead of waiting for the termination of the second phase on that branch (as it happened in the sequential implementation of

PreDeLo), can carry on with the computation of the first phase (potentially generating other branches).
If, after the verification of the second phase, a branch remains open, then the whole entailment cannot be satisfied (we have therefore found a counterexample), so the computation process can be interrupted early.
This way, if the first phase reaches the end of its computation (meaning that it has generated every candidate model of the KB), it doesn't have to wait for the "answers" of each second phase in every case:

- if all the branches generated in the first phase remain open (i.e. all the branches generated in the second phase answer *yes*), then the theorem prover answers *yes* (which means the query entails from the KB);
- if any of them is closed (i.e. any branch generated in the second phase answers *no*), the global answer is *no* (which means the query does not entail from the KB).

6.4 Worker-employer

In order to describe the software architecture that we have developed, we refer to the *worker-employer* metaphor.
The system is composed of:

- a single *employer*, which is in charge of verifying the query and yielding the final result. It also implements the first phase of the calculus and uses $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ to generate the various *parallel branches*: the ones that it cannot close, it passes to a *worker*;
- an unlimited number of *workers*, which use $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ to evaluate the models generated by the *employer*;
- a *repository*, which stores all the answers coming from the *workers*.

Figure 6.1 shows us a high–level representation of the system.

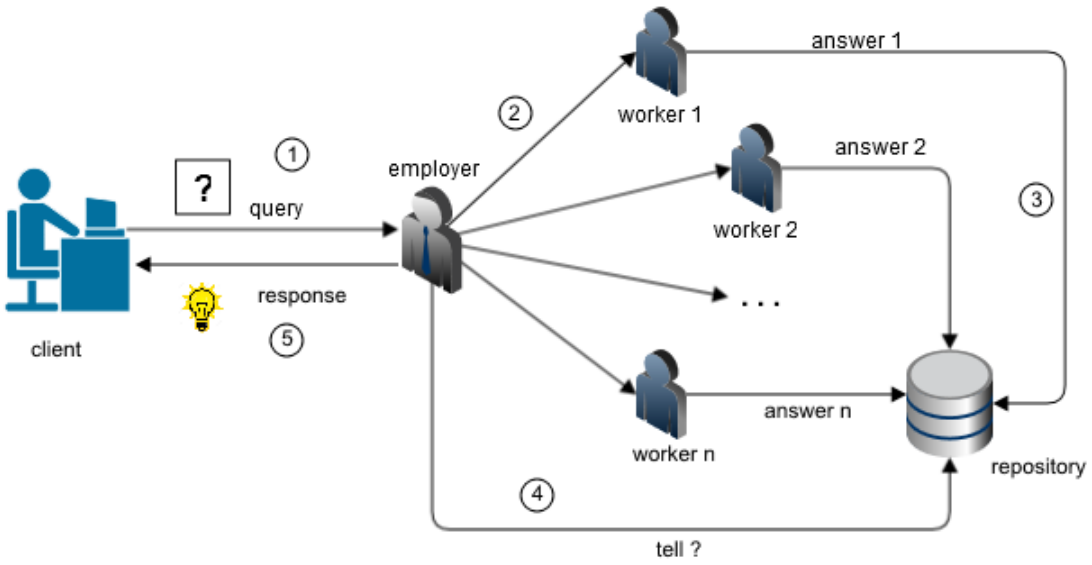

Figure 6.1

When presented with a query (1), the employer begins to analyse it through the first phase of the calculus. If, during the computation of the first phase calculus, the *employer* needs to verify a *parallel branch* via the second phase calculus, it delegates it to one of the registered *workers* (2), and consequently proceeds with its calculations on other branches. When a *worker* terminates its execution, it reports its result to the *repository* (3).
The *employer* has to keep a continuous dialogue with the *repository* (4):

- if every branch has been processed and each worker has answered affirmatively, the employer can conclude that the query does entail from the KB;
- otherwise, since a single negative answer from a *worker* can invalidate the whole entailment, the *employer* can conclude the proof as soon as the first negative answer comes into the *repository*.

The execution ends when the response ("YES" or "NO") is presented to the client (5).

6.5 Technologies used

6.5.1 Calculus implementation - SICStus Prolog

Prolog is a declarative programming language developed at the University of Marseille, as a practical tool for programming in logic. The program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a query over these relations[24].
SICStus is a Prolog system developed at the Swedish Institute of Computer Science. Parts of the system were developed by the project "Industrialization of SICStus Prolog" in collaboration with Ericsson Telecom AB, NobelTech Systems AB, Infologics AB and Televerket[34].

Both the employer and the workers of our system rely on Prolog for their computing cores: respectively `alct1.pl` and `alct2.pl`. The former implements the first phase ($\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$), while the latter implements the second phase of the calculus ($\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH2}$).

6.5.2 Interface between Prolog and Java - Jasper

Jasper is a bi-directional interface between Java and SICStus. The Java-side of the interface consists of a Java package (`se.sics.jasper`) containing classes representing the SICStus runtime system (`SICStus`, `SPTerm`, etc). The Prolog part is designed as a library module (`library(jasper)`).
The library module `library(jasper)` provides functionality for controlling the loading and unloading the JVM (Java Virtual Machine), method call functionality (`jasper_call/4`), and predicates for managing object references.
Jasper can be used in two modes, depending on which system acts as *parent application*.

1. If Java is the parent application, the SICStus runtime kernel will be loaded into the JVM using the `System.loadLibrary()` method (this is done indirectly when instantiating a SICStus object). In this mode, SICStus is loaded as a runtime system.

2. If SICStus is the parent application, Java will be loaded as a foreign resource using the query `use_module(library(jasper))`. The Java engine is initialized using `jasper_initialize/[1,2]`[35].

Our implementation uses both of these modes, with the purpose of decoupling the two phases of the calculus.

The *employer* handles the query in `Phase1Thread.java`, a piece of Java code which presents it to `alct1.pl`, the Prolog core implementing $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ (case 1).

Listing 6.1: Phase1Thread.java

```
27 // Creates a new SICStus object.
28 SICStus sp = new SICStus();
29
30 // Loads the prolog implementation of the tableau for
31 // the first phase of the calculus.
32 sp.load("alct1.pl");
33
34 Map<String, SPTerm> map = new HashMap<String, SPTerm>();
35
36 // Makes the SICStus object open the query.
37 Query query = sp.openPrologQuery(queryString, map);
```

Every time that an open branch is generated, `alct1.pl` calls (case 2) `Phase1RMIStub.java`, another piece of Java code which will send it to the correct *worker*.

Listing 6.2: alct1.pl

```
293 jasper_call(JVM,
294        method('employer/Phase1RMIStub', 'solveViaRMI', [static]),
295        solve_via_rmi(+chars, +chars),
296        solve_via_rmi(NextWorkerName, OpenBranch) ),!.
```

The *workers* will then have to process the open branches with $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$, which is implemented in `alct2.pl`. They will do so in `SolverThread.java` (case 1).

Listing 6.3: SolverThread.java

```
22 Query query = sp.openPrologQuery(remoteQuery, map);
23
24 // Verifies whether the query can be solved.
25 if (query.nextSolution()) {
26        // If so, asks the repository to store a positive answer.
27        repository.store(true);
28 } else {
29        // Otherwise, asks the repository to store a negative answer.
30        repository.store(false);
31 }
32 query.close();
```

6.5.3 Concurrency - Java threads

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel")[1].
The Java platform is designed from the ground up to support concurrent programming, with basic concurrency support in the Java programming language and the Java class libraries. Since version 5.0, the Java platform has also included high-level concurrency APIs[31].
Concurrency is the main goal of our implementation, since we want the execution of the first phase of the calculus to be independent from the second. Java natively supports concurrency via multithreading.

The *employer* uses a separate thread (implemented in `Phase1Thread.java`) to perform the actual invocation of $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$ on the query, while its main thread polls the *repository* waiting for termination[2].

Listing 6.4: Employer.java

```
116 // Spawns a new thread which launches the first phase of the calculus
117 Thread phase1Thread = new Thread(new Phase1Thread(queryString));
118 phase1Thread.start();
119
120 Repository repository = (Repository) registry.lookup("Repository");
121
122 // The employer now polls the repository until it has at least one answer.
123 while (true) {
124         ...
125         if (repository.tell())
126                 System.out.println("YES, inference done.");
127         else
128                 System.out.println("NO, inference failed.");
129 }
```

[2]Let us recall the reader that a global termination can be reached when the first counterexample is found, even if not all of the branches have been explored.

During $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$, every time that the *employer* wants to ask a *worker* to verify a branch (this happens in `Phase1RMIStub.java`), a new thread is spawned (`PrologWorker.java`).

Listing 6.5: Phase1RMIStub.java

```
68 // Spawns a new thread, asking it to solve the query.
69 Thread t = new Thread(new PrologWorker(worker, query));
70 t.start();
71
72 // Increments the counter of the queries which have been asked.
73 c.increment();
```

The *worker* itself uses threads: its main thread (`Phase2Worker.java`) simply enqueues each request coming from the *employer* (in `Solver.java`) and spawns a new thread, `SolverThread.java`, which performs $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$.

Listing 6.6: Solver.java

```
38 solverThread = new Thread(new SolverThread(this.name, pendingRequests));
39 solverThread.start();
```

6.5.4 System distribution - Java RMI

The Java Remote Method Invocation (Java RMI) is a Java API that performs the object-oriented equivalent of remote procedure calls (RPC), with support for direct transfer of serialized Java classes and distributed garbage collection [29]. It enables to create distributed Java applications, in which the methods of remote objects can be invoked from other Java virtual machines, possibly on different hosts[30].
As stated in 6.5.2, since *DysToPic* relies on a Java interface to Prolog, it was natural to design the distribution feature around Java APIs, too[3].
The RMI architecture makes use of a *Registry* (`java.rmi.registry.Registry`) that provides methods for storing (methods `bind`, `unbind`, `rebind`) and retrieving (methods `list` and `lookup`) remote object references bound with arbitrary string names.

[3]With Java versions before Java 5.0 developers had to compile RMI stubs in a separate compilation step using `rmic`. Version 5.0 of Java and beyond no longer require this step.

The *repository* (which runs on the same machine as the *employer*) will begin its execution by asking the RMI registry to bind itself (`AnswersRepository.java`) as "Repository".

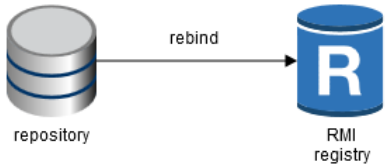

Figure 6.2

Due to a limitation in the RMI protocol, we had to make the repository expose a method called `proxyRebind(String name, Remote stub)`, which binds the Remote object that is supplied to it to the local RMI registry, with the selected `name` [4].
Each *worker* then asks the RMI registry to lookup for the *repository* (steps (1) and (2)) and uses `proxyRebind` to bind itself (steps (3) and (4)) to the RMI registry as `"Worker*"`, where `*` is any string, i.e. both `"Worker1"` and `"WorkerFoo"` are valid names.

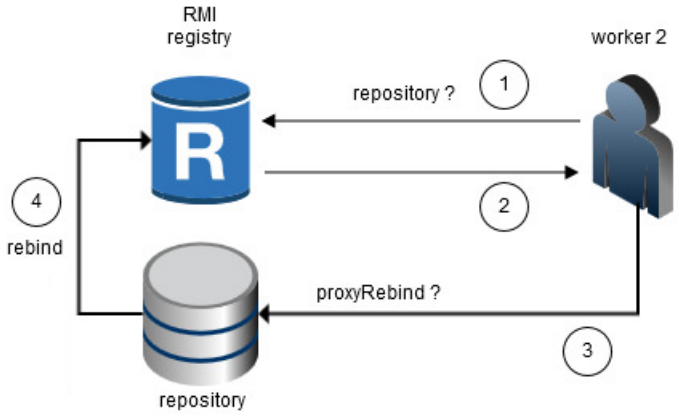

Figure 6.3

When the *employer* starts, it retrieves the list of all the *workers* which are currently bound to the RMI registry and consults it every time it needs a *worker*.

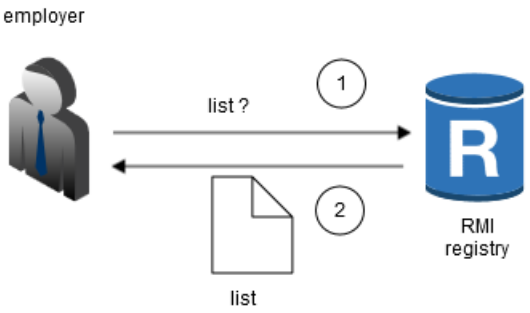

Figure 6.4

[4]This is required since by default one cannot bind a service on a RMI registry which is on a remote machine.

6.6 An example of execution

1. The RMI registry is launched on the machine 'A' on port 'x': every other machine will have to perform their requests to it.

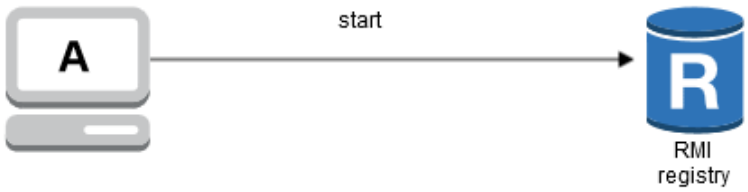

Figure 6.5

2. The repository (`AnswersRepository.java`) is then executed on the same machine. It binds itself on the RMI registry as "Repository".

Figure 6.6

3. Then an arbitrary number of workers (`Phase2Worker.java`) is launched on an arbitrary number of machines (let's say machines 'B' to 'N') - for instance, one worker per CPU[5].

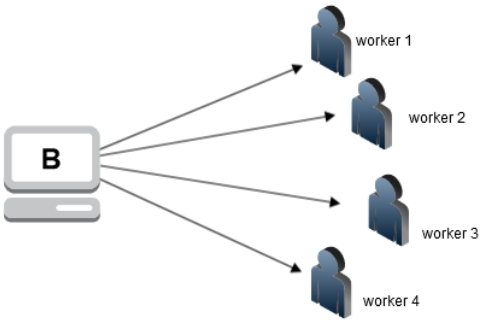

Figure 6.7

Each worker binds itself to the RMI registry (through the `proxyRebind` method of `AnswersRepository.java` - steps (1) to (4) in figure 6.8) and then starts its single instance of `SolverThread` (step (5)).

`SolverThread.java` creates a SICStus object with the code of `alct2.pl` (which corresponds to $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$) and then waits until a request (an object of class `Request.java`) comes from the employer.

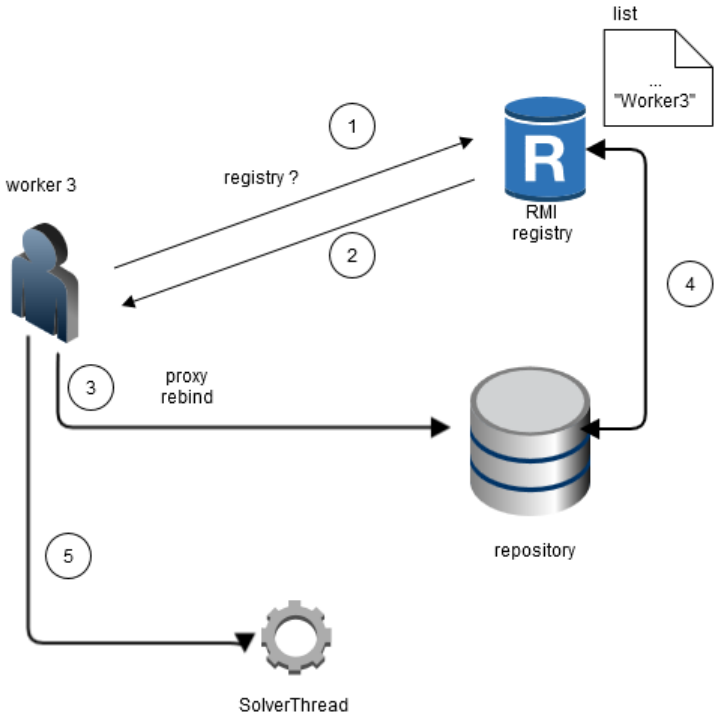

Figure 6.8

[5]We chose to use the convention that each worker has a different name, which has to be a string starting with "Worker" (i.e. "Worker1" "Worker2", and so on).

4. Afterwards, the *employer* (`Emloyer.java`) is started on the machine 'A', and (1) receives from the user a `queryString` (which contains both the KB and the actual query to be verified). It retrieves a list of each worker bound to the RMI registry (steps (2) and (3)) and (step (4)) writes it onto a file (`workersList.txt` - this file is the means for communicating the names to `alct1.pl`).

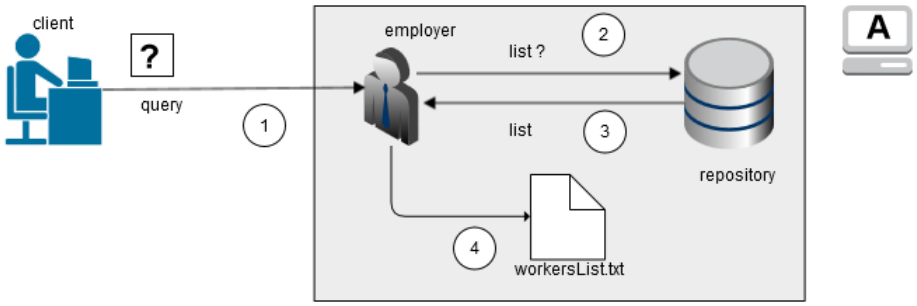

Figure 6.9

[The following description refers to figure 6.10 on page 61.]

5a. Then the *employer* spawns a new thread, `Phase1Thread.java` (1).

This creates a SICStus object and loads it with the code of `alct1.pl`, which contains the rules of $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$. Then the `queryString` is passed (2) to this SICStus object as a parameter (via Jasper). The query is executed and the control passes to the Prolog code (`alct1.pl`), which starts manipulating the `queryString` and generating various branches (3). Some may be closed (a), others may be open (b):

- If every generated branch is found to be closed, the computation can end with an affirmative answer: the query does entail from the KB.
- Otherwise, for each *open* branch found, `alct1.pl` invokes (via Jasper) the method `solveViaRMI(String workerName, String query)` - this happens in `Phase1RMIStub.java` (4). The `query` corresponds to the branch to be verified, while the `workerName` is extracted from the file `workersList.txt`[6].

`Phase1RMIStub.java`'s main thread, then (5) spawns a new thread (`PrologWorker.java`) and (6) increases the counter of the open branches (an object of class `Counter.java`).

An object of class `PrologWorker.java` allows the *employer* to use a *worker*'s method, `solve`[7], which the *employer* can use to solve the query, i.e. to ask the *worker* to use $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$ to verify whether the branch is *open* or *closed*.

[6] This is done FIFO, reading a name from the top of the file, and writing it on the bottom, and allows a simple form of load balancing.

[7] In the distributed computing environment, the server-side object participating in the distributed object communication is called a *skeleton*, while the client-side object is called a *stub*. In this particular case, the *worker* is the server, so the method is specified in the *skeleton* and the employer calls it on its *stub*.

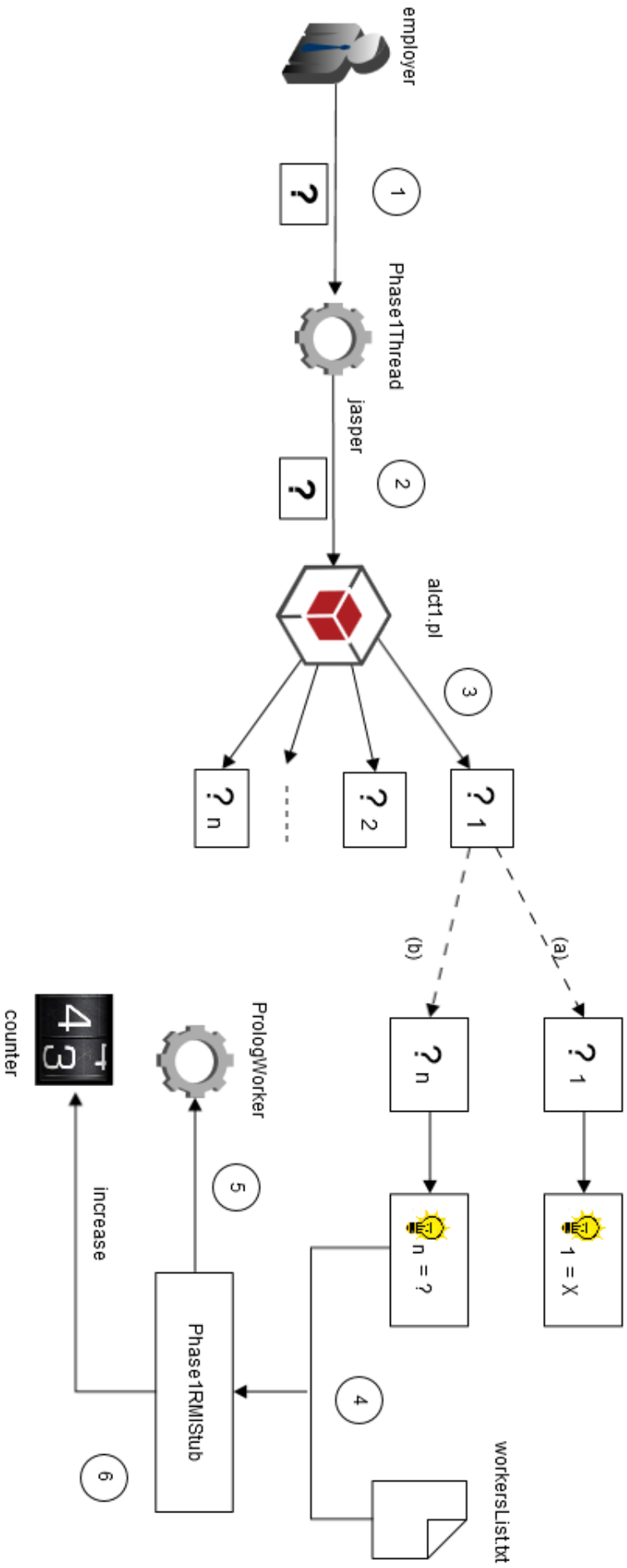
employer
?
1
Phase1Thread
jasper
?
2
alct1.pl
3
? n
? 2
? 1
(b)
(a)
? n
? 1
n = ?
1 = X
PrologWorker
5
counter
4
3
increase
Phase1RMIStub
4
6
workersList.txt

Figure 6.10

5b. Meanwhile, as we can see in figure 6.11, the *employer*'s main thread (just after spawning `Phase1Thread.java`) enters a loop in which it polls the *repository* until a global answer can be found. This happens when:

- one *worker* has answered `false`, invalidating the whole entailment (i.e. the method `tell` returns `false`) - here the *employer* answers 'NO';
- all the open branches (counted by `Counter.java`) have been tested and every *worker* has answered `true` (the method `tell` still returns `false`) - here the *employer* answers 'YES'.

Until then, the *employer* just remains in a busy wait condition.

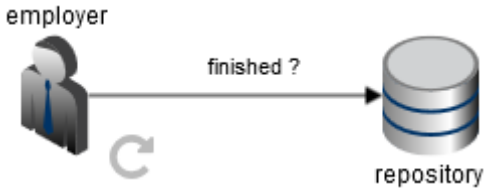

Figure 6.11

6a. When the `PrologWorker` thread of the *employer* calls `solve(String remoteQuery)` on a *worker*, for instance "Worker2" on machine 'B' (steps (1) to (3)), the *worker* simply asks its solver object (class `Solver.java`) to enqueue the request (4).

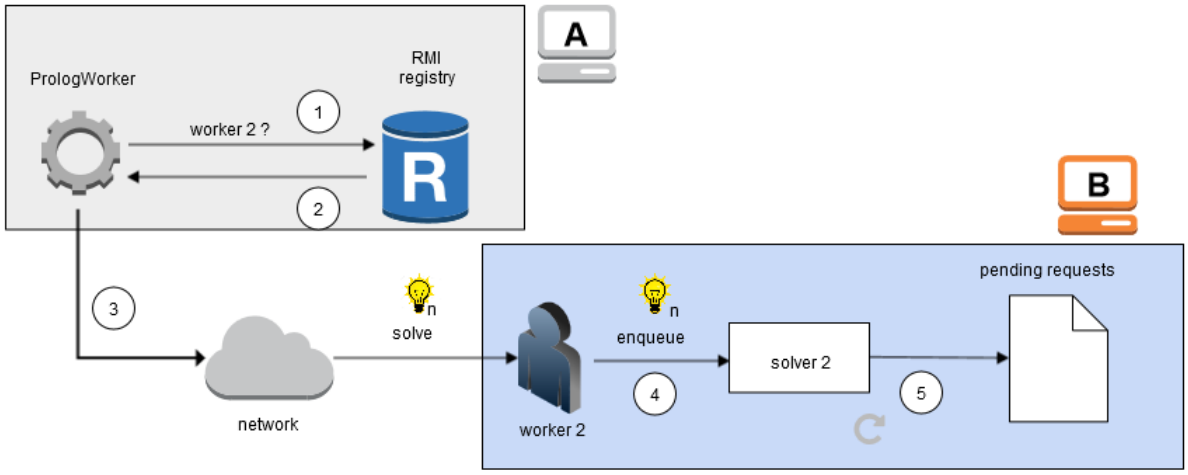

Figure 6.12

The `SolverThread` will then take the request from the queue and elaborate it through `alct2.pl` (steps (5) to (8)). This elaboration yields an answer of the form `true` or `false` (9), which is then sent (through the method `store(Boolean answer)`) to the repository (steps (10) to (12)). The `SolverThread` then restarts its infinite loop and looks for another request that may or may not have come from the *employer* in the meantime.

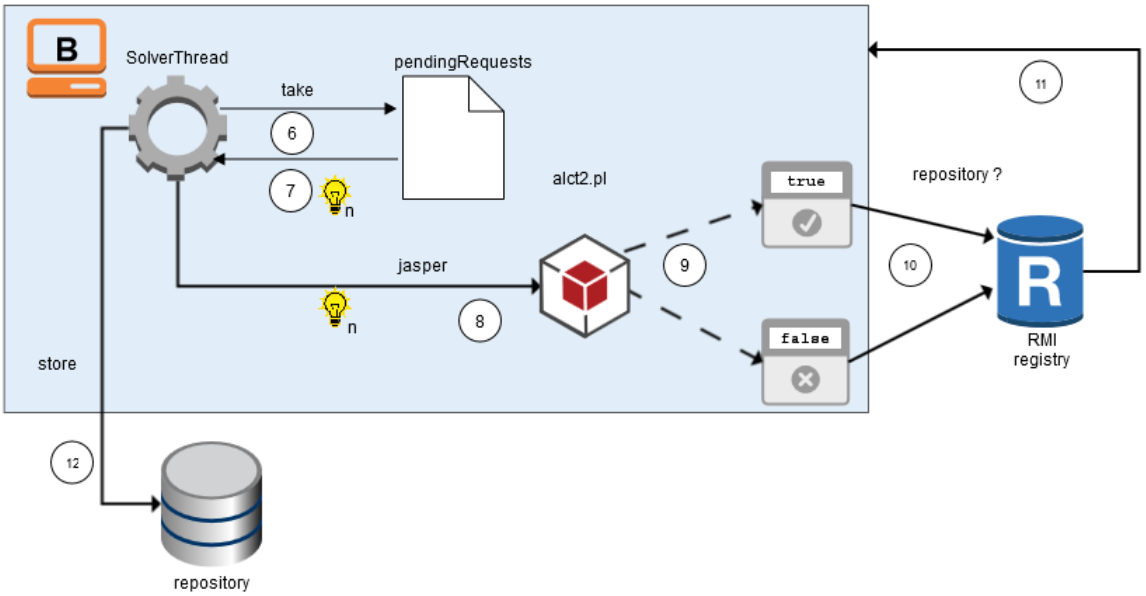

Figure 6.13

Chapter 7

Preliminary performance testing

We have made a preliminary attempt to show how *DysToPic* performs, especially in comparison with *PreDeLo*.

Due to the novelty of the $\mathcal{ALC} + \mathbf{T}_{min}$ language, there are no public KBs available for testing purposes.
A draft of an automated random generator of KBs (and queries) was developed for *PreDeLo*. We can confirm that the data that it produces are still compatible with *DysToPic*, although the generator itself calls for further analysis. This is outside the scope of this thesis.
This chapter is far from being an exhaustive verification; however we can already say that the results for our prover are at the very least comparable with those obtainable by *PreDeLo*. In the sequel we will illustrate both some interesting test cases and the environment in which the tests were performed.

7.1 Hardware and software platforms

7.1.1 Software

Both *PreDeLo*'s and *DysToPic*'s Prolog cores have been compiled with SICStus 4.1.1 (version x86_64-linux-glibc2.7).
Due to compatibility reasons between Java and SICStus (via the Jasper library), the Java applications had to be developed using the Java 6 platform (we used the javac compiler at version 1.6.0_32)[1].

[1]Newer versions of SICStus have been released which are compatible with Java 7, although they require a new license.

7.1.2 Hardware

The machines used for our tests are:

- **ninjavm**: a desktop PC with

  CPU: Intel Core i5-3570K (3.4-3.8GHz, 6MB L3 cache) - 4 cores, 4 threads;
  RAM: 8GB
  running Ubuntu 14.04.1 64 bit OS (kernel 3.13.0-35) in a virtual machine, with 2 available virtual cores and 4GB of available RAM; the host OS is Windows 7.

- **evo**: another desktop PC with

  CPU: Intel Pentium G2030 (3.0GHz, 3MB L3 cache) - 2 cores, 2 threads;
  RAM: 4GB
  running natively Ubuntu 14.04.1 64 bit OS (kernel 3.13.0-29)

- **x220**: a Lenovo X220 laptop with

  CPU: Intel Core i7-2640M (2.8-3.5GHz, 4MB L3 cache) - 2 cores, 4 threads;
  RAM: 8GB
  running natively Ubuntu 14.04.1 64 bit OS (kernel 3.13.0-35)

- **x230vm**: a Lenovo X230 laptop with

  CPU: Intel Core i7-3520M (2.9-3.6GHz, 4MB L3 cache) - 2 cores, 4 threads;
  RAM: 8GB
  running Ubuntu 14.04.1 64 bit OS (kernel 3.13.0-35) in a virtual machine, with 2 available virtual cores and 4GB of available RAM; the host OS is Windows 7.

All are fairly recent machines and can be considered almost equal in terms of calculating capacity. The following results have been obtained with each machine connected to a LAN over a domestic router, via Ethernet links at 100 Mbit/s.

7.2 Expectations

Following what we stated in 6.3, we recall that the possible outcomes of an execution of $\mathcal{TAB}_{min}^{\mathcal{ALC}+\mathbf{T}}$ on a query are:

1. an immediate *YES* at the end of the first phase ($\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$), without need of any verification with the second phase ($\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$).

   This can happen in case the query is trivially true or the KB is inconsistent;

2. a global *NO*, which requires that at least one *worker* had answered `"false"` to an open branch;

3. a global *YES*, which requires that every *worker* had answered `"true"` to every open branch [2] .

Clearly we know that the increased parallelism achieved by *DysToPic* comes at the price of an initial overhead required for the set–up of the various distributed components of its architecture. *DysToPic* uses the *repository* as a form of shared memory for the partial results, which can also be a source of slowdowns. Moreover, it is clear that the responses will take some time to come from the remote *workers* to the local *repository*.
For these reasons, we obviously expect the queries of case 1. (immediate *YES*) to be resolved faster by *PreDeLo*.
In general, we also expect that this overhead can cause *PreDeLo* to have a bit of advantage in case the query requires only a small amount of running time.
The cases 2. and 3. are instead more promising for *DysToPic*: its parallelism can be fully exploited when case various branches require the verification of the second phase.

[2] Our tests seem to suggest that the random generator described at the beginning of chapter 7 cannot generate examples of case 3. Further investigation of this issue is required before drawing definitive conclusions. We reaffirm that this is outside the scope of this work.

7.3 Experimental results

7.3.1 Naming convention

Obviously, since *PreDeLo* is a sequential program, it can be only executed on a single machine; we tested it only on **ninjavm**.

On the other hand, *Dystopic* was tested in various configurations. While both the *employer* and the *repository* were always executed on **ninjavm**, in the sequel we call:

- **2W** a configuration in which 2 *workers* were executed on **ninjavm**;
- **4W** a configuration in which 2 *workers* were executed on **ninjavm** and 2 on **evo**;
- **8W** a configuration in which 2 *workers* were executed on **ninjavm**, 2 on **evo** and 4 on **x220**;
- **10W** a configuration in which 2 *workers* were executed on **ninjavm**, 2 on **evo**, 4 on **x220** and 2 on **x230vm**.

As we showed in section 7.1.2, some of the machines are actually virtualised, although we limited the amount of *workers* per machine, so that each machine has exactly one *worker* per logical thread.

7.3.2 Comparison of running times

Query 1: answer YES (case 1)

This query does not produce any open branch in $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$.

ABox	**TBox**
$Bird(tweety)$	$\mathbf{T}(Bird) \sqsubseteq FlyingAnimal$
$Penguin(tweety)$	$Penguin \sqsubseteq Bird$
	$Penguin \sqsubseteq \neg FlyingAnimal$

Query	**Result**
$\neg FlyingAnimal(tweety)?$	YES

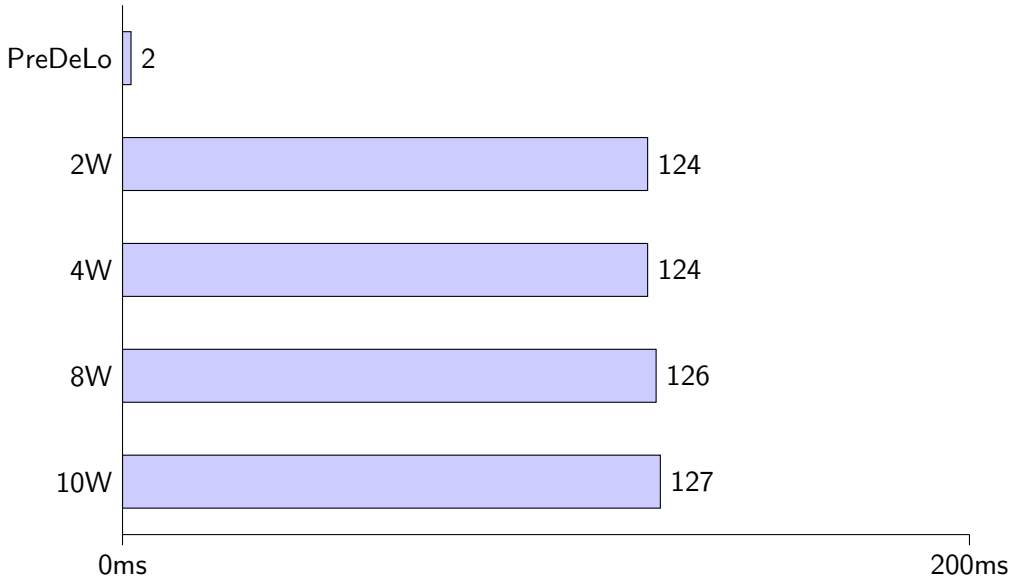

Figure 7.1

This chart shows us that, in this first case, the set–up overhead dominates the running time of *DysToPic*: the query can be easily solved by $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$ and *PreDeLo* does it more efficiently.
On the other hand, from a practical point of view, there is no noticeable difference between the running times of both theorem provers.
We can also notice how a small amount of time is required to start the various *workers*.

Query 2: answer NO (case 2)

This query generates 1 open branch in $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$, which has to be tested by a single *worker* in $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$.

ABox	**TBox**
$Student(mario)$	$\mathbf{T}(Student) \sqsubseteq \neg IncomeTaxPayer$
$WorkingStudent(mario)$	$WorkingStudent \sqsubseteq Student$
$Tall(mario)$	$\mathbf{T}(WorkingStudent) \sqsubseteq IncomeTaxPayer$

Query	**Result**
$\neg IncomeTaxPayer(mario)$?	NO

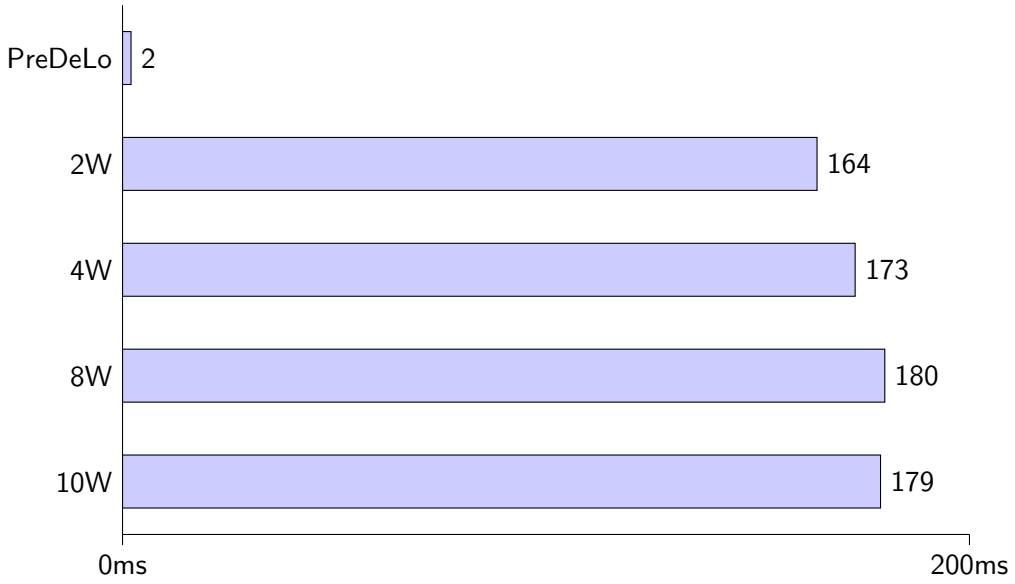

Figure 7.2

The same considerations done for the previous chart still hold in this case.
Even if a branch requires further verification by $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$, this requires very little effort (it takes less than 2 ms if done sequentially) and the running time is still widely dominated by the overhead.
However, *DysToPic* is not noticeably slower than *PreDeLo*.

Query 3: answer YES (case 3)

This query generates 2 open branches in $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$, which have to be tested by the *workers* in $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH2}$.

ABox	**TBox**
$Student(mario)$	$\mathbf{T}(Student) \sqsubseteq \neg IncomeTaxPayer$
$WorkingStudent(mario)$	$WorkingStudent \sqsubseteq Student$
$Tall(mario)$	$\mathbf{T}(WorkingStudent) \sqsubseteq IncomeTaxPayer$

Query	**Result**
$IncomeTaxPayer(mario)$?	YES

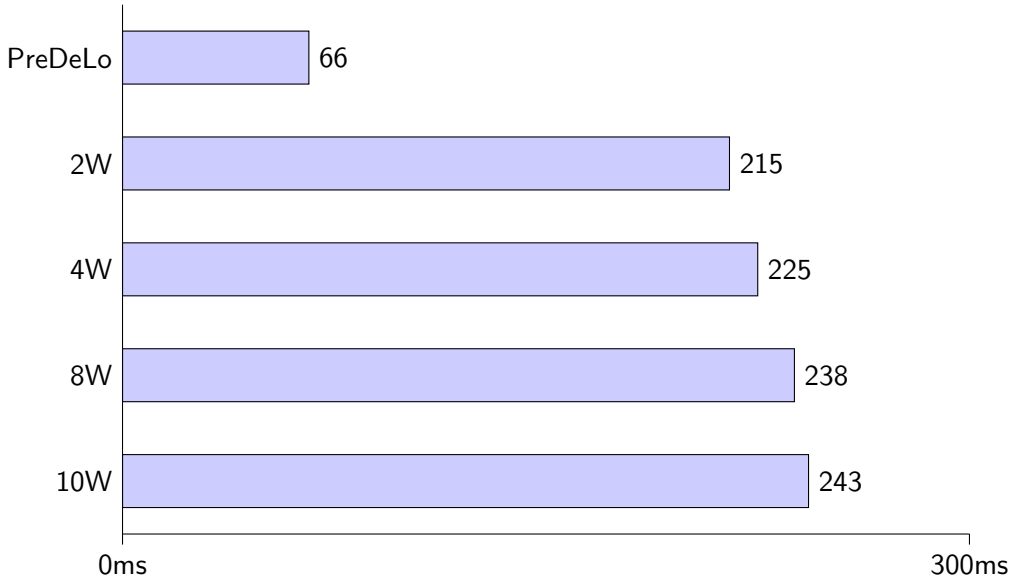

Figure 7.3

This chart confirms that two, "easy", open branches are still not enough for *DysToPic* to take advantage of its parallelism; the running times are a bit longer for *PreDeLo*, but still nothing noticeable happens.

In general, we can make a query more complicate in various ways, for instance by adding elements to both the ABox or the TBox, or increasing the degree of nesting in the inclusions in the TBox.

We decided to test how each theorem prover reacted to a query which is substantially identical to the previous one, but with a larger KB.

In section 3.2 (Definition 8), we showed how adding *irrelevant* assertions does not modify the result of the entailment. Though, as we can see from the results, this comes at the cost of a larger number of open branches generated by $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$.

Query 4: answer YES (case 3)

This query generates 37 open branches in $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$, which have to be tested by the *workers* in $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$.

ABox	**TBox**
$Student(mario)$	$\mathbf{T}(Student) \sqsubseteq \neg IncomeTaxPayer$
$WorkingStudent(mario)$	$WorkingStudent \sqsubseteq Student$
$Tall(mario)$	$\mathbf{T}(WorkingStudent) \sqsubseteq IncomeTaxPayer$
$Student(carlo)$	
$WorkingStudent(carlo)$	
$Tall(carlo)$	

Query	**Result**
$IncomeTaxPayer(mario)$?	YES

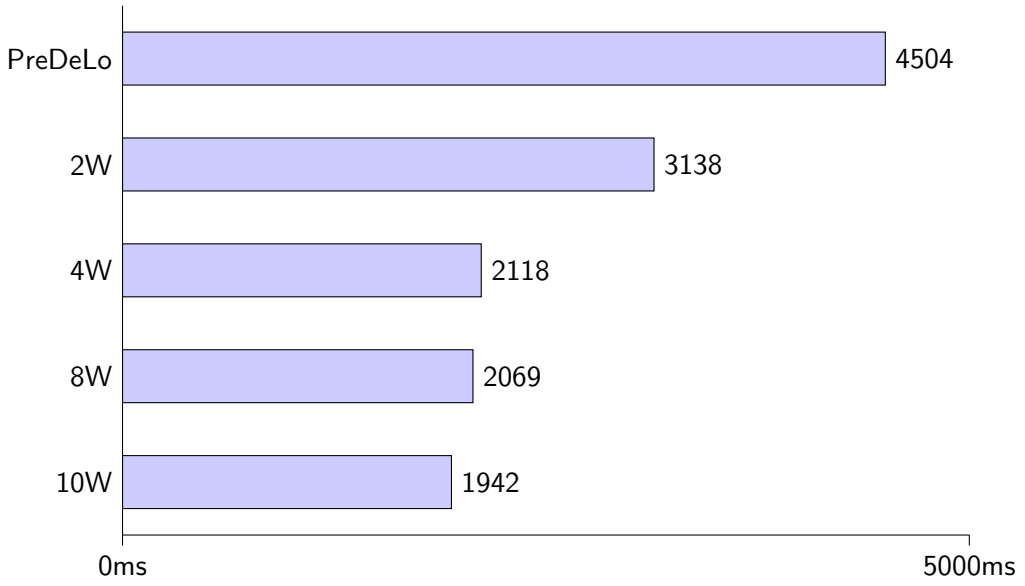

Figure 7.4

This is the first case in which *DysToPic* can prevail; notice that the improvement is also visibly noticeable (in terms of seconds!).

The 37 open branches have to be verified sequentially by *PreDeLo* and this clearly slows it down. The fact that the performances of *DysToPic* do not directly scale at the increase of the number of *workers* may mean that one of these branches is more complex than the others, and its running time dominates the global result.

Query 5: answer YES (case 3)

This query generates 1090 open branches in $\mathcal{TAB}_{PH1}^{\mathcal{ALC}+\mathbf{T}}$, which have to be tested by the *workers* in $\mathcal{TAB}_{PH2}^{\mathcal{ALC}+\mathbf{T}}$.

ABox	**TBox**
$Student(mario)$	$\mathbf{T}(Student) \sqsubseteq \neg IncomeTaxPayer$
$WorkingStudent(mario)$	$WorkingStudent \sqsubseteq Student$
$Tall(mario)$	$\mathbf{T}(WorkingStudent) \sqsubseteq IncomeTaxPayer$
$Student(carlo)$	
$WorkingStudent(carlo)$	
$Tall(carlo)$	
$Student(giuseppe)$	
$WorkingStudent(giuseppe)$	
$Tall(giuseppe)$	

Query	**Result**
$IncomeTaxPayer(mario)$?	YES

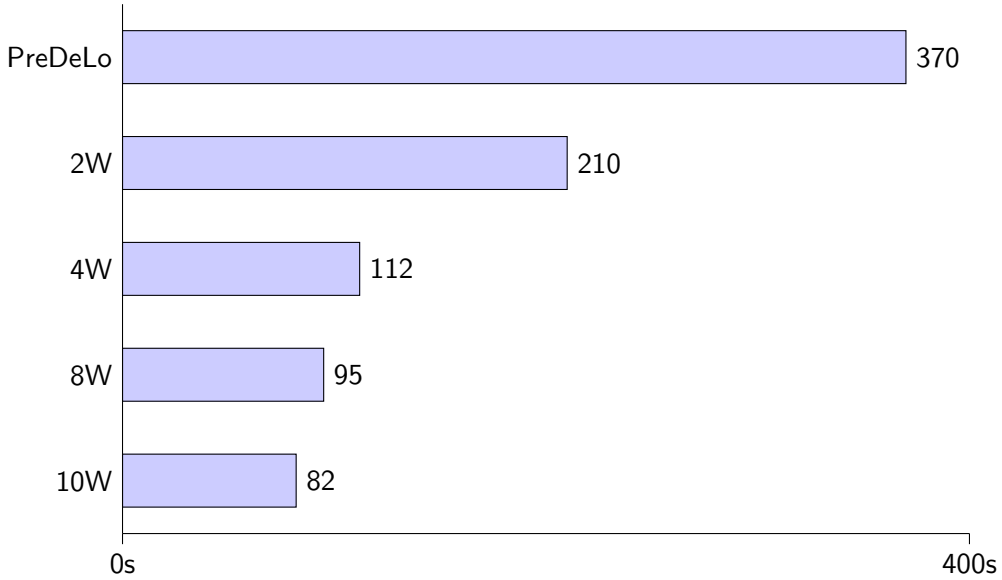

Figure 7.5: Please note that this chart shows data measured in seconds, while the previous charts were measured in milliseconds.

Here, the improvement of *DysToPic* is even more evident. We can see how the performance increase scales well up to 4 *workers*. The improvement with more *workers* is marginal, meaning that probably there are some branches that require prolonged running times.

Chapter 8

Conclusions and future issues

In this thesis we presented *DysToPic*, a distributed theorem prover implementation for $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{min}$.

After describing the Description Logic $\mathcal{ALC}$, we showed how it was non–monotonically extended to obtain an expressive language for knowledge representation, $\mathcal{ALC}+\mathbf{T}_{min}$, for reasoning about prototypical properties and inheritance with exceptions.
Then we introduced the tableau calculus $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{min}$, presented in [20], which has been developed for deciding the satisfiability of $\mathcal{ALC}+\mathbf{T}_{min}$, and we discussed how the degree of expressiveness affects the decision complexity: the problem of deciding if a formula is minimally entailed respect to a KB is in co-NEXP$^{\mathrm{NP}}$. Such a high complexity classifies the problem itself as untreatable, nonetheless logics such as this are adopted in many real-life environments, thanks to various adaptations to the specific application environments, which provide approachable complexities. This shows us how even extremely complex problems can be treated, under specific conditions.

Since the previously developed theorem prover, *PreDeLo*, showed the possibility of a mechanism of automated demonstration of complex formulas for this calculus, we decided to improve such approach.
DysToPic is a Java software, built around two separate Prolog cores which implement the two phases of $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{min}$ ($\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH1}$ and $\mathcal{TAB}^{\mathcal{ALC}+\mathbf{T}}_{PH2}$). It relies on the RMI technology for the distribution of the calculus and it is multi–threaded for its parallelisation. This allows us to run the verifications (of the second phase) of multiple branches in parallel, on multiple machines at the same time, possibly finding faster a counterexample in case of answer "NO".
Even if *DysToPic*'s performances are promising, we intend to study further improvements through the application of refinements. We think that the first step in this direction should be the implementation of a query generator, to obtain a set of significant examples in order to perform further automatised performance tests.

DysToPic's system architecture has been designed to be as general as possible, in order to be open to further applications and improvements.
For instance, the adaptation of this software to the low complexity tableaux calculus $\mathcal{TAB}_{min}^{Lite_c\mathbf{T}}$ (the calculus corresponding to $DL-Lite_c\mathbf{T}_{min}$), would be almost effortless; doing so we would sacrifice some language expressiveness as a tradeoff for the decreased complexity of the decision.
$DL-Lite$ and its related logic $\mathcal{EL}$ are specifically tailored for effective query answering over DL knowledge bases containing large amounts of data (e.g the GALEN medical knowledge base), and are applied in a wide range of fields.

Moreover, our distributed approach is conceptually general, and it could be adapted to various reasoning mechanisms: for instance, those which make use of a semantics with preferred models (e.g. for conditional logics [19], or for circumscription [21], [38], [33]).

Bibliography

[1] Almasi, G. S., and Gottlieb, A. *Highly Parallel Computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[2] Baader, F., and Hollunder, B. Embedding defaults into terminological knowledge representation formalisms. *Journal of Automated Reasoning (JAR) 14*, 1 (1995), 149–180.

[3] Baader, F., and Hollunder, B. Priorities on defaults with prerequisites, and their application in treating specificity in terminological default logic. *Journal of Automated Reasoning (JAR) 15*, 1 (1995), 41–68.

[4] Baader, F., and Nutt, W. The description logic handbook. Cambridge University Press, New York, NY, USA, 2003, ch. Basic Description Logics, pp. 43–95.

[5] Bonatti, P. A., Lutz, C., and Wolter, F. Description Logics with Circumscription. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR 2006)* (Lake District of the United Kingdom, 2006), P. Doherty, J. Mylopoulos, and C. A. Welty, Eds., AAAI Press, pp. 400–410.

[6] Bonatti, P. A., Lutz, C., and Wolter, F. The Complexity of Circumscription in DLs. *Journal of Artificial Intelligence Research (JAIR) 35* (2009), 717–773.

[7] Buchheit, M., Donini, F. M., and Schaerf, A. Decidable reasoning in terminological knowledge representation systems. *Journal of Artificial Intelligence Research (JAIR) 1* (1993), 109–138.

[8] Casini, G., and Straccia, U. Rational Closure for Defeasible Description Logics. In *Proceedings of the 12th European Conference on Logics in Artificial Intelligence (JELIA 2010)* (Helsinki, Finland, September 2010), T. Janhunen and I. Niemelä, Eds., vol. 6341 of *Lecture Notes in Artificial Intelligence (LNAI)*, Springer, pp. 77–90.

[9] Casini, G., and Straccia, U. Defeasible Inheritance-Based Description Logics. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)* (Barcelona, Spain, July 2011), T. Walsh, Ed., Morgan Kaufmann, pp. 813–818.

[10] Donini, F. M., Lenzerini, M., Nardi, D., Nutt, W., and Schaerf, A. An Epistemic Operator for Description Logics. *Artificial Intelligence 100*, 1-2 (1998), 225–274.

[11] Donini, F. M., Nardi, D., and Rosati, R. Description logics of minimal knowledge and negation as failure. *ACM Transactions on Computational Logic (ToCL) 3*, 2 (2002), 177–225.

[12] Eiter, T., Lukasiewicz, T., Schindlauer, R., and Tompits, H. Combining Answer Set Programming with Description Logics for the Semantic Web. In *Principles of Knowledge Representation and Reasoning: Proceedings of the 9th International Conference (KR 2004)* (Whistler, Canada, June 2004), D. Dubois, C. Welty, and M. Williams, Eds., AAAI Press, pp. 141–151.

[13] Giordano, L., Gliozzi, V., Jalal, A., Olivetti, N., and Pozzato, G. L. Predelo 1.0: A theorem prover for preferential description logics. In *AI*IA* (2013), M. Baldoni, C. Baroglio, G. Boella, and R. Micalizio, Eds., vol. 8249 of *Lecture Notes in Computer Science*, Springer, pp. 60–72.

[14] Giordano, L., Gliozzi, V., Olivetti, N., and Pozzato, G. L. $\mathcal{ALC}$+**T**: a preferential extension of Description Logics. *Fundamenta Informaticae 96* (2009), 1–32.

[15] Giordano, L., Gliozzi, V., Olivetti, N., and Pozzato, G. L. On Extending Description Logics for Reasoning About Typicality: a First Step. *TR Dip. di Informatica,* `http://www.di.unito.it/~pozzato/tr09.pdf` (2009).

[16] Giordano, L., Gliozzi, V., Olivetti, N., and Pozzato, G. L. On Extending Description Logics for Reasoning About Typicality: a First Step. In *Technical Report 116/09, Dip. di Informatica, Univ. di Torino* (2009).

[17] Giordano, L., Gliozzi, V., Olivetti, N., and Pozzato, G. L. A tableau calculus for a nonmonotonic extension of $\mathcal{EL}^{\perp}$. In *Proceedings of TABLEAUX 2011 (20th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods)* (Bern, Switzerland, July 2011), K. Brünnler and G. Metcalfe, Eds., vol. 6793 of *LNAI*, Springer-Verlag, pp. 180–195.

[18] Giordano, L., Gliozzi, V., Olivetti, N., and Pozzato, G. L. A tableau calculus for a nonmonotonic extension of the Description Logic $DL-Lite_{core}$. In *AI*IA 2011: Artificial Intelligence Around Man and Beyond - XIIth International Conference of the Italian Association for Artificial Intelligence, Palermo, Italy, September 15-17, 2011. Proceedings* (Palermo, Italy, Sptember 2011), R. Pirrone and F. Sorbello, Eds., vol. 6934 of *LNAI*, Springer-Verlag, pp. 164–176.

[19] Giordano, L., Gliozzi, V., Olivetti, N., and Pozzato, G. L. A minimal model semantics for nonmonotonic reasoning. In *Logics in Artificial Intelligence - 13th European Conference, JELIA 2012* (Toulouse, France, Sptember 2012), J. M. Luis Fariñas del Cerro, Andreas Herzig, Ed., vol. 7519 of *LNAI*, Springer-Verlag, pp. 228–241.

[20] Giordano, L., Gliozzi, V., Olivetti, N., and Pozzato, G. L. A non-monotonic description logic for reasoning about typicality. *Artif. Intell. 195* (Feb. 2013), 165–202.

[21] Grimm, S., and Hitzler, P. A Preferential Tableaux Calculus for Circumscriptive $\mathcal{ALCO}$. In *Proceedings of the 3rd International Conference on Web Reasoning and Rule Systems (RR 2009)* (Chantilly, VA, USA, October 2009), A. Polleres and T. Swift, Eds., vol. 5837 of *Lecture Notes in Computer Science (LNCS)*, Springer, pp. 40–54.

[22] Krisnadhi, A. A., Sengupta, K., and Hitzler, P. Local closed world semantics: Keep it simple, stupid! In *Proceedings of the 24th International Workshop on Description Logics (DL 2011)* (Barcelona, Spain, July 2011), vol. 745 of *CEUR Workshop Proceedings*.

[23] Lehmann, D., and Magidor, M. What does a conditional knowledge base entail? *Artificial Intelligence 55*, 1 (1992), 1–60.

[24] Lloyd, J. W. *Foundations of Logic Programming*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[25] Motik, B., and Rosati, R. Reconciling Description Logics and rules. *Journal of the ACM 57*, 5 (2010).

[26] Nardi, D., and Brachman, R. J. The description logic handbook. Cambridge University Press, New York, NY, USA, 2003, ch. An Introduction to Description Logics, pp. 1–40.

[27] Nute, D. Topics in conditional logic. *Reidel, Dordrecht* (1980).

[28] Obitko, M. Introduction to ontologies and semantic web, 2007.

[29] Oracle. Java se 7 remote method invocation (rmi)-related apis & developer guides, 2014. [Online; accessed September-2014].

[30] Oracle. Java(tm) remote method invocation api (java rmi), 2014. [Online; accessed September-2014].

[31] Oracle. Java(tm) tutorials - lesson: Concurrency, 2014. [Online; accessed September-2014].

[32] Schmidt-Schaubss, M., and Smolka, G. Attributive concept descriptions with complements. *Artif. Intell. 48*, 1 (Feb. 1991), 1–26.

[33] Sengupta, K., Krisnadhi, A. A., and Hitzler, P. Local closed world semantics: Grounded circumscription for owl. In *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I* (Berlin, Heidelberg, 2011), ISWC'11, Springer-Verlag, pp. 617–632.

[34] SICS. Introduction to sicstus, 2014. [Online; accessed September-2014].

[35] SICS. Jasper overview, 2014. [Online; accessed September-2014].

[36] Sripriya, G., Bundy, A., and Smaill, A. Concurrent-distributed programming techniques for sat using dpll-stålmarck. In *HPCS* (2009), W. W. Smari and J. P. McIntire, Eds., IEEE, pp. 168–175.

[37] Straccia, U. Default inheritance reasoning in hybrid kl-one-style logics. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI 1993)* (Chambéry, France, August 1993), R. Bajcsy, Ed., Morgan Kaufmann, pp. 676–681.

[38] Voorbraak, F. A preferential model semantics for default logic. In *Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty* (London, UK, UK, 1991), ECSQAU, Springer-Verlag, pp. 344–351.