Parallelizing Description Logic Reasoning

Kejia Wu

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

For the Degree of Doctor of Philosophy

Concordia University

Montréal, Québec, Canada

April 2014

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Mr. Kejia Wu**

Entitled: **Parallelizing Description Logic Reasoning**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this university and meets the accepted standards with respect to originality and quality.

Singed by the final examining committee:

_____ Chair
Dr. Amir Aghdam

_____ External Examiner
Dr. Weichang Du

_____ Examiner
Dr. Jamal Bentahar

_____ Examiner
Dr. René Witte

_____ Examiner
Dr. Hovhannes Harutyunyan

_____ Supervisor
Dr. Volker Haarslev

Approved _____
Chair of Department or Graduate Program Director

_____ 20_____ _____
Dr. Christopher Trueman, Interim Dean
Faculty of Engineering and Computer Science

Abstract

Parallelizing Description Logic Reasoning

Kejia Wu, Ph.D.
Concordia University, 2014

Description Logic has become one of the primary *knowledge representation and reasoning* methodologies during the last twenty years. A lot of areas are benefiting from description logic based technologies. Description logic reasoning algorithms and a number of optimization techniques for them play an important role and have been intensively researched.

However, few of them have been systematically investigated in a concurrency context in spite of multi-processor computing facilities growing up. Meanwhile, *semantic web*, an application domain of description logic, is producing vast knowledge data on the Internet, which needs to be dealt with by using scalable solutions. This situation requires description logic reasoners to be endowed with *reasoning scalability*.

This research introduced concurrent computing in two aspects: classification, and tableau-based description logic reasoning.

Classification is a core description logic reasoning service. Over more than two decades many research efforts have been devoted to optimizing classification. Those classification optimization algorithms have shown their pragmatic effectiveness for

sequential processing. However, as concurrent computing becomes widely available, new classification algorithms that are well suited to parallelization need to be developed. This need is further supported by the observation that most available OWL reasoners, which are usually based on tableau reasoning, can only utilize a single processor. Such an inadequacy often leads users working in ontology development to frustration, especially if their ontologies are complex and require long processing times.

Classification service finds out all named concept subsumption relationships entailed in a knowledge base. Each subsumption test enrolls two concepts and is independent of the others. At most n^2 subsumption tests are needed for a knowledge base which contains n concepts. As the first contribution of this research, we developed an algorithm and a corresponding architecture showing that reasoning scalability can be gained by using concurrent computing.

Further, this research investigated how concurrent computing can increase performance of tableau-based description logic reasoning algorithms. Tableau-based description logic reasoning decides a problem by constructing an *AND-OR* tree. Before this research, some research has shown the effectiveness of parallelizing processing *disjunction* branches of a tableau expansion tree. Our research has shown how reasoning scalability can be gained by processing *conjunction* branches of a tableau expansion tree.

In addition, this research developed an algorithm, *merge classification*, that uses

a *divide and conquer* strategy for parallelizing classification. This method applies concurrent computing to the more efficient classification algorithm, *top-search & bottom-search*, which has been adopted as a standard procedure for classification. Reasoning scalability can be observed in a number of real world cases by using this algorithm.

Acknowledgments

Years ago, my mind was full of enthusiasm for knowledge, but I did not know which direction I should go. When this Ph.D. thesis was completed, my mind was still zealous for knowledge, and I got a compass. I will never find this compass on my own.

My supervisor, Dr. Volker Haarslev, gave me the valuable opportunity of finishing an exciting science exploration. He led me into the domain of automated reasoning. With the guidance of Dr. Haarslev, I learned a lot of skills on how to work as a scientist. I feel very grateful to him for his aids.

Some ideas in my Ph.D. research had been enriched by discussions with Ms. Jinan El Hachem and Mr. Ming Zuo. I appreciate working with these nice colleagues.

I thank Dr. Ralf Möller very much for providing me with necessary facilities to conduct my research.

I might never start my academic career if Dr. Desanka Polajnar, Dr. Jernej Polajnar, and Dr. Liang Chen had not insisted on their approving of me. I feel very grateful to these intellectual scientists.

My family members' contributions to my research are immeasurable. Any words can not express my eternal gratitude to them.

Contents

Listings

List of Algorithms

List of Tables

List of Figures

Chapter 1

Introduction

Knowledge Representation and Reasoning (KR) is a key component of *Artificial Intelligence (AI)*, and Description Logic (DL) has been one of the most advanced achievements of KR. Compared with other KR methodologies, DL provides *strong expressivity*, *acceptable computationality*, and *logical completeness*. By adding constrains, DL establishes a reasonable compromise between expressivity and completeness, and so computationality is assured, although DL reasoning is still a hard problem. Some key reasoning tasks of DL languages having stronger expressivity are known to be ExpTime-complete [23, 37]. Before modern DL reasoning, such hard problems were regarded as impractical ones. However, with language formalization and optimization techniques developed in DL, problems that are hard to be solved are rarely encountered in the real world. Hence, DL has been largely used today.

Semantic web is an important application area of DL now.[1] Originally proposed by the inventor of the World Wide Web, Tim Berners-Lee, and now maintained by World Wide Web Consortium (W3C), "The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries." [6] It endeavors to build a structured information web that

[1] We do not differentiate between "*uppercase* semantic web" and "*lowercase* semantic web". The latter term is used for *ad hoc* solutions that have limited application domains [37].

can be automatically processed in a more efficient way than the present web. With semantic web, information is encoded in some uniformed knowledge representations, and so can be deduced by machines. Until now, the semantic web community has proposed a chain of technologies, such as *Resource Description Framework (RDF)*, *Resource Description Framework Schema (RDFS)*, Web Ontology Language (OWL), *SPARQL Protocol and RDF Query Language (SPARQL)*, and so forth. These technologies endow the current information web with formal machine understandable *semantics*, the foundation of automatic deduction. With other available semantic technologies such as DL reasoning, semantic web will make significant changes for the ways people organize and utilize information.

Meanwhile, semantic web is producing a vast number of automatic-deduction-friendly information. Both the scale and complexity of the knowledge bases generated by semantic web are growing rapidly. The increment of the scale and complexity of semantic web products is challenging the reasoning capability of DL, which is a major computing principle of semantic web.

So, DL, the underpinning of semantic web, should advance with matched information processing capability. After many years efforts, DL reasoning techniques have been capable of processing real-world knowledge bases, which are normally represented as ontologies with some formalized languages now. These techniques have been adopted by DL systems to assist to solve practical deductive problems.

In some areas, DL based ontology engineering technology has become a needful approach that leads to novel findings. In bioinformatics, DL systems help researchers make a number of new observations, which can hardly be deduced by other means. For example, some Nortriptyline treatments involving a special homozygous genotype can be automatically discovered via DL reasoning [24]. Knowledge produced by modern science and technology has vastly exceeded the capability of manual efforts. Without an automatic assistant, such as a DL system, a number of interesting conclusions can almost not be drawn from a great multitude of knowledge.

However, there are some complicated ontologies taking well-known DL systems a long time to process, though DL has constrained the computational complexity to an acceptable range with its own formalism and a number of reasoning optimization techniques. Such complicated ontologies are growing rapidly.

With the development of semantic web, more large and more complex knowledge bases may appear in the near future. Now DL needs to prepare corresponding computing kits to welcome semantic web time. Scalable reasoning techniques are solutions to dealing with complicated ontologies that are growing larger and more complex.

This research investigated how concurrent computing can be applied to DL reasoning in order to gain scalable performance.

1.1 Motivation

Due to the semantic web, a multitude of OWL ontologies are emerging. Quite a few ontologies are huge and contain hundreds of thousands of concepts. Although some of these huge ontologies fit into one of OWL's three tractable profiles, such as the well known *Snomed* ontology into the $\mathcal{EL}$ profile, there still exist a variety of other OWL ontologies that make full use of OWL DL and require long processing times, even when highly optimized OWL reasoners are employed. Moreover, although most of the huge ontologies are currently restricted to one of the tractable profiles in order to ensure fast processing, it is foreseeable that some of them will require an expressivity that is outside of the tractable OWL profiles.

At the same time, *concurrent computing* facilities, such as multi-processor/core computers, have become popular, while most well-known DL reasoners can not fully utilize the computing resources. Hence, concurrent computing solutions that can both deal with complex ontologies in a scalable way and make use of computing resources are necessary.

However, it is very recently that researchers have begun to investigate how to adopt concurrent computing as an optimization of DL reasoning [16].

Classification, one of the core DL reasoning tasks, is just opening the envelope of utilizing concurrent computing. Almost all well-known reasoners employ a so-called *top-search & bottom-search* algorithm to classify ontologies [56]. This algorithm makes use of told subsumption relationships to prune a lot of costly subsumption tests. Concepts are incrementally inserted into a subsumption hierarchy at their most specific positions. This method works efficiently in practical reasoning, and a number of variants proposed on the basis of the original version provide optimizations to some extent [12, 29]. However, only in recent years efforts appeared to investigate parallelization of top-search & bottom-search in order to gain more scalable performance [9]. Some researchers also have begun to optimize DL tableau reasoning with concurrent computing [55].

The work presented in this research is targeted to provide better OWL reasoning scalability by making efficient use of modern hardware architectures such as multi-processor/core computers. This becomes more important in the case of ontologies that require long processing times although highly optimized OWL reasoners are already used. We consider our research an important basis for the design of next-generation OWL reasoners that can efficiently work in a parallel/concurrent or distributed context using modern hardware. One of the major obstacles that need to be addressed in the design of corresponding algorithms and architectures is the overhead introduced by concurrent computing and its impact on scalability.

Shared data in concurrent computing generally needs *synchronization* to ensure soundness, and synchronization itself and its maintenance cost always decrease the performance of concurrent computing. Primary DL reasoning algorithms, which are widely used in OWL reasoning, work along with monolithic data structures that are hardly parted. Those algorithms and data have to be conducted as a whole, and thus can be hardly processed in parallel. Consequently, the efficiency of concur-

rent reasoning is weakened unless it uses novel algorithms that can process data as independently as possible.

A *divide and conquer* algorithm split a problem into sub-problems, deals with the sub-problems independently, and then combines intermediate solutions into a final one. During a divide and conquer process, operations on shared data are largely reduced, and no much communication cost is spent on the sub-problem solving. Therefore, divide and conquer algorithms are suited to improving the performance of concurrent reasoning.

We also noted that a number of researchers are investigating *ontology partitioning*, which may degrade the complexity of reasoning about monolithic ontologies. For example, [30] presented a method of *ontology modularization*. A collection of sub-ontologies can be obtained from a complex ontology by the ontology modularization method, as makes it possible to reason over a collection of sub-ontologies in a concurrent way, and then to assemble a solution for the original ontology from the deduction results of sub-ontologies. This process can be conducted by the divide and conquer pattern.

It was necessary to investigate how the *divide and conquer* strategy can improve the performance of DL concurrent reasoning, and related investigations, which focus on DL classification, were conducted in this research.

Besides classification, how concurrent computing can be applied to some aspects of DL tableau reasoning remains to be investigated. Tableau-based deduction procedures are the primary algorithms used by almost all well-known DL reasoners. Very limited research has been conducted to apply concurrent computing to general DL tableau algorithm. How general tableau algorithms can get scalable performance via concurrent computing is also part of this research.

1.2 Problem Statement

A DL system provides a set of reasoning services. These tasks include concept satisfiability test, TBox consistency test, concepts subsumption test, TBox classification, and so forth. The TBox classification service is an important one exposed to users by all DL systems.

The TBox classification computes all subsumption relationships between named concepts entailed in a terminology of a knowledge base. For a complex knowledge base using expressive constructors, a number of relationships among concepts are complicated and can hardly be deduced out. Classification can generate a taxonomy that completely describes such relationships. Interesting conclusions entailed in the knowledge base are consequently discovered.

The classification service establishes binary relations, and so needs to compute n^2 subsumption tests for a n-concept knowledge base. The majority of the classification computation lies in testing the subsumption relationship between each pair of concepts. Subsumption tests generally involve satisfiability tests and hence essentially have an exponential time complexity [23]. So, traditional classification optimizations always try to avoid such a costly computation.

Instead of a brute force method, the *top-search & bottom-search* algorithm is used as the standard classification procedure. This algorithm can practically prune a lot of subsumption tests by making use of told subsumptions.

As aforementioned, researchers are exploring scalable solutions to deal with more and more complex DL knowledge bases, and applying concurrent computing to DL classification is a choice. However, related research has just begun [9].

This research investigated how classification can be performed in a scalable way. Concurrent computing is a very important option for scalability. So, some work of our research can also be regarded as how concurrent computing should be used in DL classification in order to obtain scalable reasoning performance.

Furthermore, concurrent computing has been partly investigated in general DL tableau reasoning. Tableau-based DL algorithms always produce expansion trees during reasoning. Those tableau expansion trees consist of *disjunctive* branches and *conjunctive* branches. Related work mainly focuses on how to adopt concurrent computing to process disjunctive tableau expansion branches. How concurrent computing can improve reasoning performance by parallelizing processing conjunctive tableau branches had not been investigated yet and is part of this research.

1.3 Contributions

This research introduced *concurrent computing* into DL reasoning. Specifically, shared-memory parallelization is used to optimize DL TBox reasoning. The contributions of this research are reflected in three aspects [85, 86, 87].

The classification computation is a key task of DL reasoning. DL TBox classification calculates all concept subsumption relationships entailed in a knowledge base, and such a subsumption test involves two concepts. In a knowledge base which possesses n concepts, at most n^2 subsumption tests are needed to find out all entailed subsumption relationships, and each test is independent of the others in a brute-force method. Therefore, the subsumption tests in a classification computation can be executed in a parallel way. This research designed an algorithm that can process DL TBox classification in parallel. A corresponding architecture had been designed and implemented for the concurrent algorithm. The experiments showed an obvious reasoning performance improvement.

This research also showed that concurrent computing can improve performance of general tableau-based reasoning. Tableau-based reasoning techniques have been widely adopted by most DL reasoners. In order to find a model, or to show that no models exist, a knowledge base is expanded in terms of a set of tableau deduction rules. Such a tableau expansion process can be viewed as a procedure of constructing

a tree consisting of disjunctive branches and conjunction branches.

Before this research, researchers had conducted some exploration of using concurrent computing to process disjunctive branches of tableau expansion trees, and achieved good performance. One contribution of this research is exploring parallelizing conjunction branches. Compared with parallelizing disjunctive branches, which generally needs to get one clash-free branch, parallelizing conjunctive branches normally needs to explore all branches and so is more suited to concurrent computing. This research designed an algorithm that processes conjunctive branches of a tableau expansion tree in a parallel way, and conducted the experiments and analyzed the performance.

The first achievement of this research utilizes concurrent computing to increase the performance of the *brute-force search* based classification algorithm. This research further tackled the more efficient classification algorithm, *top-search & bottom-search*, with parallel optimization.

One of the main obstacles impacting improving performance of concurrent computing is the inefficient management of *shared resources*, among which *operations* and *data* are the most popular shared resources that need to be managed efficiently. This issue is prominent in concurrent reasoning. *Divide and conquer* strategy may efficiently overcome this weakness of concurrent computing.

This research adopted the *divide and conquer* strategy into DL TBox classification. A global knowledge base is divided into subsets that can be classified *independently*, and thus the classification computations can progress in parallel. When the subset knowledge bases are classified, the results can be merged together, and this process can also be done in parallel. A concurrent algorithm reflecting the aforementioned idea had been developed and implemented in this research. The experiments had shown that reasoning performance could be improved by using this algorithm in a number of cases.

1.4 Thesis Overview

This thesis reports the knowledge that we found during exploring the concurrent reasoning in DL. Related contents, such as DL and concurrent computing, is mentioned, too. The rest of this thesis is organized as follows:

- This research is mainly about how concurrent computing can improve the performance of DL reasoning, so the background knowledge on DL, DL reasoning, semantic web, and concurrent computing is covered in Chapter 2.

- A lot of DL reasoning optimization techniques have been researched in the last twenty years, and some researchers have begun applying concurrent computing to DL reasoning recently; without those fundamental achievements and novel investigations, DL reasoning was unrealistic. Related research is mentioned in Chapter 3.

- We investigated how tableau-based DL reasoning can make use of concurrent computing; we present an architecture that is suitable for concurrent DL classification. This work is addressed in Chapter 4.

- Conjunctive branches of tableau expansion tree can be processed in a parallel way, and thus we investigated how the performance of DL reasoning can be improved by parallelizing tableau conjunctive branches. This work is addressed in Chapter 5.

- Efficient *memory maintenance* is a key to effectively improve the performance of DL reasoning by using concurrent computing. We investigated a concurrent classification algorithm that uses *divide and conquer* strategy. This work is addressed in Chapter 6.

- Future work and some conclusive points of this research are summarized in Chapter 7.

Chapter 2

Preliminaries

The fundamentals on DL, DL reasoning, and optimization techniques are briefly described in this chapter. DL formalism is addressed. Typical DL reasoning services are introduced. Basic DL reasoning methods, especially *tableau-based* ones, are explained. Essential optimization techniques without which practical DL reasoning is impossible are reviewed. *Non-determinism* potentially leads to parallel DL reasoning techniques, so it is highlighted as a feature of DL in this chapter.

2.1 Description Logic

Our work is essentially about DL reasoning, so some background knowledge is presented in this section. A more detailed background on DLs, DL reasoning, and semantic web was referred to [14] and [37].

As a conceptualization KR tool, DL is derived from *network-based* formalisms, especially *semantic networks* [14, 41]. Network-based KR methods present cognitive structures as node-link networks that describe objects, object attributes, and relationships between them. Generally, attributes and relationships are inheritable in those networks [17]. A similar idea was simultaneously developed in *frame systems* at the same time, which focused on using *frames* and relationships among them to de-

scribe knowledge (see [59]). *KL-ONE* was a milestone in semantic network research [18]. Based on experience gained from developing KL-ONE, *non-logical* semantic network depiction was elaborated with *symbolic logic*. Later, DL appeared as a protocol that formalizes with syntax and semantics the principal characteristics of network-based KR. We quote the sketched definition of DL by [14]:

> Description Logics is the most recent name for a family of knowledge representation (KR) formalisms that represent the knowledge of an application domain (the "world") by first defining the relevant concepts of the domain (its terminology), and then using these concepts to specify properties of objects and individuals occurring in the domain (the world description).

Following its definition, [14] commented on two features of DL: *(i)* the formal syntax and model theoretic semantics, and *(ii)* the emphasis on reasoning as central function.

DL is used to represent knowledge. *Concepts* and *roles* are elements constructing DL expressions. The former conceptualize knowledge domain instances, and the latter describe binary relations between domain instances. DL axioms are constructed by associating the two essentials via a set of connectives, *concept constructors* and *role constructors*. For example, the syntax for the language $\mathcal{AL}$ is defined as follows [14]:

$C, D \longrightarrow A$	|	(* atomic concept *)
$\top$	|	(* universal concept *)
$\bot$	|	(* bottom concept *)
$\neg A$	|	(* atomic negation *)
$C \sqcap D$	|	(* intersection *)
$\forall R.C$	|	(* value restriction *)
$\exists R.\top$	|	(* limited existential quantification *)

In the productions, A corresponds to a *concept name*, C or D to either a compound concept or a concept name, and R to a role name.

Generally, a DL language's semantics is expressed via Taski-style model-theoretical *interpretations* [79]. Such an interpretation for $\mathcal{ALCN}$, $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, consists of a

non-empty set of individuals ($\Delta^{\mathcal{I}}$) and a function ($\cdot^{\mathcal{I}}$) such that:

$$C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$$
$$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$$
$$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$$
$$\bot^{\mathcal{I}} = \emptyset$$
$$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$$
$$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$$
$$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$$
$$(\forall R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \forall y((x,y) \in R^{\mathcal{I}} \implies y \in C^{\mathcal{I}})\}$$
$$(\exists R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y((x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}})\}$$
$$(\geq nR)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \left|\{y \mid (x,y) \in R^{\mathcal{I}}\}\right| \geq n, n \in \mathbb{Z}_{\geq 0}\}$$
$$(\leq nR)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \left|\{y \mid (x,y) \in R^{\mathcal{I}}\}\right| \leq n, n \in \mathbb{Z}_{\geq 0}\}$$
$$C \sqsubseteq D \text{ iff } C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$$
$$C \equiv D \text{ iff } C \sqsubseteq D \wedge D \sqsubseteq C$$
$$x^{\mathcal{I}} \in \Delta^{\mathcal{I}}$$
$$C(x) \text{ or } x : C \text{ if } x^{\mathcal{I}} \in C^{\mathcal{I}}$$
$$(R(x,y))^{\mathcal{I}}, (xRy)^{\mathcal{I}}, ((x,y) : R)^{\mathcal{I}}, \text{ or } (\langle x,y\rangle : R)^{\mathcal{I}} \text{ if } \langle x^{\mathcal{I}}, y^{\mathcal{I}}\rangle \in R^{\mathcal{I}}$$
$$x \doteq y \text{ if } x^{\mathcal{I}} = y^{\mathcal{I}}$$
$$x \neq\!\!\!\doteq y \text{ if } x^{\mathcal{I}} \neq y^{\mathcal{I}}$$

An interpretation $\mathcal{I}$ satisfies an axiom $C \sqsubseteq D$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. An axiom $C \equiv D$ is considered as an abbreviations for the set of axioms $\{C \sqsubseteq D, D \sqsubseteq C\}$. An assertion $C(x)$ is satisfied by $\mathcal{I}$ if $x^{\mathcal{I}} \in C^{\mathcal{I}}$, (xRy) if $\langle x^{\mathcal{I}}, y^{\mathcal{I}}\rangle \in R^{\mathcal{I}}$, $x \doteq y$ if $x^{\mathcal{I}} = y^{\mathcal{I}}$, and $x \neq\!\!\!\doteq y$ if $x^{\mathcal{I}} \neq y^{\mathcal{I}}$. An overall introduction to DL syntax, semantics, notation, and extensions can be found in the appendix of [14].

A modern DL *knowledge base (KB)* consists of two components: *terminological knowledge*, defining axiom vocabularies, and *assertional knowledge*, specifying an individual vocabulary. The former is a so-called *TBox*, and the latter an *ABox*. The core of a DL KB is composed by describing *concepts* and *roles*. Concepts may be described as *primitive* and *complete* axioms [12]. Primitive axioms indicate only *necessary* membership descriptions, and complete axioms indicate *necessary* and *sufficient* membership descriptions. For instance, "$A \sqsubseteq B$" is a primitive axiom, and "$A \equiv B$" is a complete one. If the DL expression on the left hand side of an axiom is not an atomic concept name, the axiom is called a *general* one. So, a TBox may hold *general axioms* to express common descriptions.

Definition 1 *A DL* ***axiom*** *has the form of $C \sqsubseteq D$ or $C \equiv D$, with semantics as mentioned above.*

Definition 2 *A DL* ***TBox*** *$\mathcal{T}$ is a conjunction of DL axioms.*

Definition 3 *A DL* ***assertion*** *is a description on instance(s) and has the form of $C(x)$, xRy, $x \doteq y$, or $x \neq y$, with semantics as mentioned above, where x and y are instances, and C and R is a concept and a role expression respectively.*

Definition 4 *A DL* ***ABox*** *$\mathcal{A}$ is a conjunction of DL assertions.*

For example, the TBox axiom "$coffee \sqcup tea \sqsubseteq beverage \sqcap \neg \exists having\_content.alcohol$" expresses "coffee and tea are soft drinks". A set of instances, i.e. objects or individuals, sharing the same characteristics are grouped as a concept, and binary relations over objects are specified as roles. Explicit descriptions on some instances can be expressed with DL assertions. For instance, "$hold(PETRA, HAMLET)$" may be in-

terpreted as "Petra holds the book *Hamlet*", following the semantics of:

$$PETRA^{\mathcal{I}} \in \Delta^{\mathcal{I}}$$
$$HAMLET^{\mathcal{I}} \in book^{\mathcal{I}}$$
$$book^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$$
$$hold^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$$

Similarly, the assertion "$cat(GARFIELD)$", combined with the TBox axiom

$$cat \sqsubseteq \neg\exists like.dog$$

expresses "Garfield is a cat, so he does not like any dog".

More expressive DL languages are composed by using a set of role constructors. Common role relations are *role transitivity* (R^+), *role hierarchy* ($\mathcal{H}$), *role inversion* ($\mathcal{I}$), *role composition* ($\mathcal{R}$), and so on, which can be built via a set of DL role constructors with semantics as follows:

$$\begin{aligned}
(R \sqsubseteq S)^{\mathcal{I}} &= R^{\mathcal{I}} \subseteq S^{\mathcal{I}} \\
(\neg R)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \setminus R^{\mathcal{I}} \\
(R \sqcap S)^{\mathcal{I}} &= R^{\mathcal{I}} \cap S^{\mathcal{I}} \\
(R \sqcup S)^{\mathcal{I}} &= R^{\mathcal{I}} \cup S^{\mathcal{I}} \\
(R^-)^{\mathcal{I}} &= \{(x,y) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (y,x) \in R^{\mathcal{I}}\} \\
(R^+)^{\mathcal{I}} &= \bigcup_{i \geq 1} (R^{\mathcal{I}})^i, \textit{ i.e. the transitive closure of } R^{\mathcal{I}} = (R^{\mathcal{I}})^+ \\
(R \circ S)^{\mathcal{I}} &= \{(x,z) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid \exists y((x,y) \in R^{\mathcal{I}} \wedge (y,z) \in S^{\mathcal{I}})\}
\end{aligned}$$

Some DL fragments allow instance names to be part of the concept languages, not only ABox elements. Such an expressivity can be obtained by a *nominal* constructor

($\mathcal{O}$). *One-of* is a typical nominal constructor, and has the following semantics:

$$(\{a_1, \ldots, a_n\})^{\mathcal{I}} = a_1^{\mathcal{I}} \sqcup \cdots \sqcup a_n^{\mathcal{I}}$$

Thus, each DL fragment has a distinct expressivity. Besides $\mathcal{AL}$ mentioned above, some interesting DL fragments include $\mathcal{ALC}$ ($\mathcal{AL}$ with full concept negation and concept disjunction), $\mathcal{SHIQ}$ ($\mathcal{ALC}$ with role transitivity ($\mathcal{ALC}_{R^+}$ or $\mathcal{S}$), role hierarchy ($\mathcal{H}$), role inversion ($\mathcal{I}$), and qualified number restrictions ($\mathcal{Q}$)), $\mathcal{SHOIN}$ ($\mathcal{S}$ with role hierarchy, nominal ($\mathcal{O}$), role inversion, and unqualified number restriction ($\mathcal{N}$)), $\mathcal{SHIF}$ ($\mathcal{S}$ with role hierarchy, role inversion, and unqualified number restrictions of the form $\leq 1R$ (i.e. functional restrictions, $\mathcal{F}$)), $\mathcal{SROIQ}$ ($\mathcal{S}$ with role composition ($\mathcal{R}$), nominal, role inversion, and qualified number restrictions), $\mathcal{EL}$ (allowing concept intersection, and full existential restrictions ($\mathcal{E}$)), and so on. As mentioned above, *reasoning* is one of the two main aspects of DL, and computational complexity in DL reasoning changes with expressivity supported by the DL fragments. At present, [88] is maintaining an online program that provides a DL fragments complexity query service.

Reasoning in DL is to search for harmonious interpretations, as indicated by DL semantics. Similar to other branches of *predicate logic*, symbols in DL KBs are expected to be assigned with reasonable interpretations. Sometimes, DL reasoning endeavors to consistently interpret a KB as a whole despite some specific symbols in it being uninterpreted; sometimes, given a consistent KB, DL reasoning attempts the interpretation of some specific symbol in terms of it. After having elementary language features introduced, the following terminology is cardinal to understand DL reasoning tasks:

Definition 5 *A **model** of an axiom is an interpretation $\mathcal{I}$ iff $\mathcal{I}$ satisfies the axiom: $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, or $C \equiv D$ if $C \sqsubseteq D$ and $D \sqsubseteq C$. A **model** of $\mathcal{T}$, a set of axioms, is an interpretation $\mathcal{I}$ iff $\mathcal{I}$ satisfies every axiom in $\mathcal{T}$.*

Definition 6 *A concept C is **satisfiable** with respect to $\mathcal{T}$, a set of axioms, if there exists a model of $\mathcal{T}$ such that $C^{\mathcal{I}} \neq \emptyset$.*

Definition 7 *A concept C is **subsumed** by a concept D with respect to $\mathcal{T}$, a set of axioms, if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for every model of $\mathcal{T}$.*

Definition 8 *Concepts C and D are **equivalent** with respect to $\mathcal{T}$, a set of axioms, if $C^{\mathcal{I}} = D^{\mathcal{I}}$ for every model of $\mathcal{T}$.*

Definition 9 *Concepts C and D are **disjoint** with respect to $\mathcal{T}$, a set of axioms, if $C^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$ for every model of $\mathcal{T}$.*

Based on these core formalisms, a DL system offers reasoning services over KBs, and Section 2.2 addresses them.

2.2 Reasoning Tasks

A set of reasoning services is available in DL. Typical services include: *(i) deciding concept satisfiability*, *(ii) checking concept subsumption*, *(iii) classifying a terminology*, *(iv) testing an instance's concept-membership*, *(v) testing two instances' relation-membership*, *(vi) computing the most specific concept of an instance*. These are known as standard services in modern DL reasoning systems.

Deciding concept satisfiability is to show whether a concept makes sense. This usually happens when inserting a new concept in a given terminology tree. The validity of the newly created concept is examined according to axioms of the given TBox. In logical methods, the examination is made by constructing an interpretation for the new TBox. If a satisfiable interpretation, i.e. a model, is found, and the new concept is not empty, then it is satisfiable. This logical examination is a process of searching for a model, which witnesses the validity of the new concept. This process can be formalized as *AND-OR* trees [71], a specific formalism of a tableau graph (see Definition 10 in Section 2.3.1).

AND-OR trees are expanded dynamically during the search for a model. AND trees are branched by clauses in concept description; disjunctions lead to OR tree branches. An OR tree directly represents a non-deterministic aspect of DL. A concept is satisfiable if and only if, with respect to the AND-OR tree derived by the TBox, each AND branch does not have a clash, and at least one OR branch has a model. If one needs to show that a concept is unsatisfiable, normally by *refutation* in subsumption testing, one contradiction in any AND branch is sufficient, while contradictions in all OR branches must be found. *AND-OR* branching thus results as a *source of complexity* (see [14, Section 3.1.1]).

Such computational complexity may be reduced by processing *AND-OR* tree branches in parallel. In DL, *AND-OR* trees are normally processed by tableau-based algorithms. OR tree branches generated by tableaux are independent from each other, so parallelizing them is straightforward. Note that an efficiency improvement may also be gained by processing AND branches in parallel in some cases besides parallelizing OR branches, and this idea was presented as an open topic by [55].

Deciding concept satisfiability is identified as an essential service on which other reasoning tasks may depend [14]. A preliminary method of *checking concept subsumptions* is to reduce the problem to *deciding concept satisfiability*. *Merging models*, an optimization technique in classification, needs to cache and to reuse results of *deciding concept satisfiability*. These techniques are going to be addressed in Section 2.3.3.

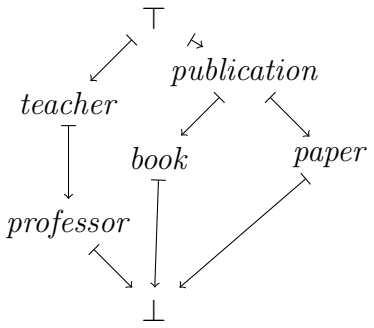
Checking subsumption is executed on pairs of concepts and decides whether a concept (i.e. subsumee or child) is subsumed by another (i.e. subsumer or parent). This decision process may be completed by measuring structural similarity of a pair of concepts or by logical deduction. The latter has a better computability than the former with respect to completeness and soundness. Tableau-based subsumption checking methods are popular logical ones. In tableaux, deduction for subsumption normally uses *refutation*. For example, to show $C \sqsubseteq D$, $C \sqcap \neg D$ should be shown to

be unsatisfiable. Searching for such refutation is normally reduced to *deciding concept satisfiability*. [75] showed the reducibility of *checking subsumption* to *deciding concept satisfiability*. The service of checking subsumption in DL systems is provided to end users as a *query* function, so all subsumptions entailed from a taxonomy can be computed in advance and cached whereby a query operation can be performed efficiently. This cache optimization is usually the result of DL KB *classification*.

Among those standard DL reasoning tasks, TBox *classification* plays an important role. TBox classification generates hierarchical taxonomies. A TBox classification algorithm computes all subsumptions between concept names ($A \sqsubseteq ? B$) that are entailed in a TBox and inserts concepts into a hierarchical structure. A result of classification can be illustrated by a directed graph with $\top$ as the root and $\bot$ as the unique leaf, which represent the most general concept and the most specific concept respectively. Figure 2.1 shows a TBox *classification* example. In the graph, each node subsumes its descendant node(s), and all paths to a node from $\top$ contain its subsumer nodes. Therefore, all information related to concept subsumptions can be extracted from the classified taxonomy.

$$\mathcal{T} : \begin{aligned} \mathit{professor} &\sqsubseteq \mathit{teacher} \\ \mathit{book} &\sqsubseteq \mathit{publication} \\ \mathit{paper} &\sqsubseteq \mathit{publication} \end{aligned}$$

(a) The TBox given.



(b) The classified terminology hierarchy.

Figure 2.1: An example on *classification*.

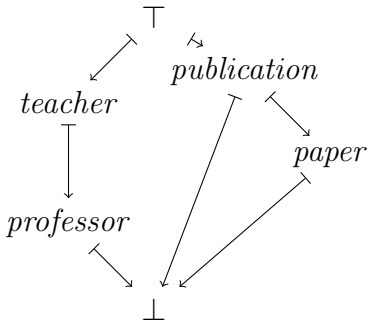
However, it is known that TBox *classification* can be a costly computation. The naive brute-force classification method executes subsumption tests over all elements of $\{\langle A_i, A_j \rangle \mid A_i^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}, A_j^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}, 0 \leq i \leq n, 0 \leq j \leq n\}$. Although the brute-force

method needs only n^2 subsumption tests for a TBox of n concepts, it is generally very expensive due to the costly subsumption testing. A smart option for classification is merely to avoid such expensive computations.

A huge computing expense lies in concept subsumption tests, so the most prominent work on classification optimization focuses on making use of the *reflexive transitive closure* of subsumptions in order to avoid costly subsumption tests—instead of checking subsumption for every pair of concepts in a brute-force way, a large number of subsumption relationships can be figured out by told subsumptions and non-subsumptions directly, and the *top-search & bottom-search* algorithm is the corner stone for such an optimization [56]. The *top-search & bottom-search* algorithm utilizes told subsumption relationships to avoid costly subsumption tests. For example, given a TBox and a partially classified terminology hierarchy shown by Figure 2.2, when searching for the most specific parent concept of *book*, it is unnecessary to test whether $book \sqsubseteq ? professor$ if $book \not\sqsubseteq teacher$ is already known. Our work shows that this technique can be extended to work in parallel.

$\mathcal{T}$: $professor \sqsubseteq teacher$
$book \sqsubseteq publication$
$paper \sqsubseteq publication$

(a) The TBox given.



(b) The partially classified terminology hierarchy.

Figure 2.2: An example on top- and bottom-search based classification.

The reasoning services presented above are key ones in TBox reasoning, which are essential interfaces of a DL system. If a DL system involves an ABox, reasoning over assertional knowledge on individuals must be taken into consideration.

Testing an instance's concept-membership is a basic DL ABox reasoning service.

This test decides whether an instance belongs to some concept. A preliminary technique for this reasoning service is to check whether a consistent ABox is still consistent after adding an assertion of an instance's concept-membership. Other ABox reasoning services may depend on it. For example, *computing the most specific concept of an instance* may test the instance's membership of each concept on one path of a classified TBox. On the other hand, an instance's concept-membership can be retrieved fast from a classified TBox after its *most specific concept* has been computed. Algorithms for *testing an instance's concept-membership* show that ABox reasoning has a close relationship with TBox reasoning.

In principle, techniques employed in reasoning tasks on ABoxes are similar to TBoxes. With tableau-based reasoning algorithms, a TBox concept satisfiability deciding process searches for a model of this concept. During the search, intermediate interpretations are constructed. These interpretations start from describing the most general concept and are expanded in completion graphs. In ABox reasoning the model-searching process constructs interpretations that are consistent with the assertions in the ABox. Therefore, primary TBox reasoning principles are also applicable to ABox reasoning. However, in cases where massive instances are concerned, particular reasoning techniques for ABoxes and TBoxes differ widely, but our research focuses on TBox reasoning.

2.3 Tableau Based Reasoning

2.3.1 Reasoning Methodology

Two types of methodologies have been extensively investigated in DL reasoning so far: *structural* and *logical* ones. Earlier DL reasoners usually adopted structural algorithms in which set-theoretical calculation dominated inference procedures, especially *subsumption-checking*. Structural algorithms perform efficiently over well-

formed concepts sets. However, those algorithms have no guarantee of *completeness* for DL fragments which allow full negation and disjunction [12, 14]. Nevertheless, structural algorithms still work well as optimizations somewhere.

Structural DL reasoning methodology deals with deduction on KBs with well-formed syntactic structures. Sophisticated normalization pre-processes were normally required in structural algorithms [12, 67]. Structural methods (in PTIME) were employed by primitive DL reasoners that did not need full negation and disjunction expressivity. Checking concept subsumptions for such simple DL languages may use structural algorithms directly, where whether a concept is subsumed by another one is inferred by comparing their syntactic definitions. When processing DLs with full negation and disjunction expressivity, the structural methods lose completeness [12, 14]. Nevertheless, as an optimization technique, especially in those DL reasoning methods that utilize *told* information, such as *top-search* and *bottom-search* classification, structural algorithms may be used to generate some *told* information efficiently. Anyway, structural algorithms are not complete in reasoning for more expressive DL fragments. More powerful reasoning capability comes from logical approaches. Among them, *tableau-based* DL reasoning algorithms are shown to be both sound and complete [13].

Most modern DL reasoning systems choose *tableaux* as their primary deduction techniques. The first tableau-based DL algorithm was introduced by [71], who described tableaux as "a set of procedures searching models for knowledge declaration". Although this description characterizes the function of tableau-based methods, or tableaux, thinking of tableaux as dynamically expanded directed graphs reveals their principles more.

This method incrementally deduces entailed information by using a set of completion rules to construct and explore a DL tableau:

Definition 10 *A* ***DL tableau*** $\mathbb{T}$ *is a set of* ***completion graphs****:* $\mathbb{T} = \{\mathbb{A}_0, \ldots, \mathbb{A}_n\}$*, where* $\mathbb{A}_i = \langle V, E\rangle$*:* $V = \{x_0, x_1, \ldots, x_n\}, E = \{R_0, R_1, \ldots, R_n\}, x_i^{\mathcal{I}} \in \Delta^{\mathcal{I}}, R_i^{\mathcal{I}} \subseteq$

$\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.

Tableaux normally start out on recursively unfolding a concept's definition. Then, each completion graph may be expanded deterministically, or new completion graphs may be added into the tableau as disjunctive stages (i.e. branches), in terms of a set of logical rules. Table 2.1 lists the tableau expansion rules of $\mathcal{ALCN}$.[1]

name	**rule**
$\sqcap$**-rule** (conjunction)	If *(i)* $C \sqcap D \in \mathcal{L}(x)$ and *(ii)* $\{C, D\} \nsubseteq \mathcal{L}(x)$, then let $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C, D\}$.
$\sqcup$**-rule**(disjunction)	If *(i)* $C \sqcup D \in \mathcal{L}(x)$ and *(ii)* $\{C, D\} \cap \mathcal{L}(x) = \emptyset$, then let $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C\}$ for one stage and $\mathcal{L}(x) = \mathcal{L}(x) \cup \{D\}$ for another one.
$\exists$**-rule** (role exists restriction)	If *(i)* $\exists R.C \in \mathcal{L}(x)$ and *(ii)* $\forall y(xRy \implies C \notin \mathcal{L}(y))$, then extend $\mathcal{L}(x)$ to new node $\mathcal{L}(y)$ via new created edge labeled R and let $\mathcal{L}(y) = \{C\}$.
$\forall$**-rule** (role value restriction)	If *(i)* $\forall R.C \in \mathcal{L}(x)$ and *(ii)* $\exists y(xRy \wedge C \notin \mathcal{L}(y))$, then let $\mathcal{L}(y) = \mathcal{L}(y) \cup \{C\}$.
$\geq$**-rule** (at-least cardinality restriction)	If *(i)* $\geq nR \in \mathcal{L}(x)$, *(ii)* no instances, z_1, ... , z_n, such that $xRz_i (1 \leq i \leq n)$, and *(iii)* $\{z_i \neq z_j \mid 1 \leq i < j \leq n\} \subseteq \mathcal{A}$, then extend $\mathcal{L}(x)$ to new nodes, $\mathcal{L}(y_1)$, ... , $\mathcal{L}(y_n)$ via new created edges labeled R respectively and let $\mathcal{A} = \mathcal{A} \cup \{y_i \neq y_j \mid 1 \leq i < j \leq n\}$.
$\leq$**-rule** (at-most cardinality restriction)	If *(i)* $\leq nR \in \mathcal{L}(x)$, and *(ii)* there exist instances, y_1, ... , y_{n+1}, such that $xRy_i (1 \leq i \leq n+1)$, then, for each pair $\langle y_i, y_j \rangle (1 \leq i < j \leq n+1)$ such that $\{y_i \neq y_j\} \nsubseteq \mathcal{A}$, generate a new stage $\mathcal{A}_{i,j}$ by substituting y_i by y_j in $\mathcal{A}$.

Table 2.1: Tableau expansion rules for deciding satisfiability of an $\mathcal{ALCN}$ concept.

Therefore, a tableau may be viewed as an *AND-OR tree*. During expansion, the process searches for logical impossibility (i.e. *contradiction*) in the tableau. A completion graph is called *closed* if it holds a contradiction.

Definition 11 *In $\mathcal{ALCN}$, let $\mathcal{A}$ be a completion graph: If*
(i) $\{C \in \mathcal{L}(x), \neg C \in \mathcal{L}(x)\} \subseteq \mathcal{A}$, *or*

[1]The more complex tableau rules for $\mathcal{SHIQ}$ are shown in Appendix A.

(ii) $\{\leq nR \in \mathcal{L}(x), xRy_i, y_i \neq y_j \mid \exists y_1, \ldots, y_i, \ldots, y_j, \ldots, y_{n+1} : 1 \leq i < j \leq n+1\} \subseteq \mathcal{A}$,

then there does not exist a model for $\mathcal{A}$*, which triggers a* ***contradiction*** *(or* ***clash****).*

Every completion graph is expanded by applying one deterministic rule each time until: *(i)* A contradiction is produced on an instance node, or *(ii)* no rule is applicable. If a tableau completion graph is expanded without any contradiction, satisfiability is thus achieved; otherwise, the concept involved is unsatisfiable.

$$\mathcal{L}(x) \quad = \quad \{\exists R.C \ \sqcap \ \forall R.(A \ \sqcup \ B)\}$$

(a) Initialize the root node x with the given axiom $\exists R.C \sqcap \forall R.(A \sqcup B)$.

$$\mathcal{L}(x) \quad = \quad \{\exists R.C, \forall R.(A \ \sqcup \ B), \exists R.C \ \sqcap \ \forall R.(A \ \sqcup \ B)\}$$

(b) Apply $\sqcap$-rule to $\exists R.C \overset{\cdot}{\sqcap} \forall R.(A \sqcup B)$ at the node x.

$$\begin{array}{rcl} \mathcal{L}(x) & = & \{\forall R.(A \ \sqcup \ B), \exists R.C \ \sqcap \ \forall R.(A \ \sqcup \ B), \exists R.C\} \\ & \Big\downarrow \text{R} & \\ \mathcal{L}(y) & = & \{C\} \end{array}$$

(c) Apply $\exists$-rule to $\overset{\cdot}{\exists} R.C$ at the node x.

$$\begin{array}{rcl} \mathcal{L}(x) & = & \{\exists R.C \ \sqcap \ \forall R.(A \ \sqcup \ B), \exists R.C, \forall R.(A \ \sqcup \ B)\} \\ & \Big\downarrow \text{R} & \\ \mathcal{L}(y) & = & \{A \sqcup B, C\} \end{array}$$

(d) Apply $\forall$-rule to $\overset{\cdot}{\forall} R.(A \sqcup B)$ at the node x.

$$\begin{array}{rcl} \mathcal{L}(x) & = & \{\exists R.C \ \sqcap \ \forall R.(A \ \sqcup \ B), \exists R.C, \forall R.(A \ \sqcup \ B)\} \\ & \Big\downarrow \text{R} & \\ \mathcal{L}(y) & = & \{C, A \sqcup B\} \\ & \swarrow \text{stage 0} \quad \text{stage 1} \searrow & \\ \mathcal{L}(y) \leftarrow \mathcal{L}(y) \cup \{A\} & & \mathcal{L}(y) \leftarrow \mathcal{L}(y) \cup \{B\} \end{array}$$

(e) Apply $\sqcup$-rule to $A \overset{\cdot}{\sqcup} B$ at the node y.

Figure 2.3: A DL tableau expansion procedure.

Figure 2.3 demonstrates an example of tableau-based DL reasoning. This example illustrates how the axiom $\exists R.C \sqcap \forall R.(A \sqcup B)$ is expanded with the tableau completion rules mentioned in Table 2.1. *Labeled graph notation* is used to illustrate a DL tableau in this proposal (see [14, Section 9.3.2.1]).

By introducing tableaux in a nutshell, we see that *non-determinism* is a feature of tableau-based DL reasoning. Obviously, disjunctive branches try to interpret all possible cases, which leads to an inherent *non-deterministic* aspect of tableaux [27]. Furthermore, at-most number restrictions introduce *non-determinism* (see $\leq$-rule in Table 2.1). Also, selecting which rule for expansion may lead to *non-determinism* sometimes. These non-deterministic aspects of tableau-based DL reasoning lead to parallelism potential.

2.3.2 Preliminary Techniques of Tableau-based DL Reasoning

Considering that many DL reasoning services are reducible to deciding concept satisfiability, how is concept satisfiability tested with tableau-based DL reasoning algorithms? As for checking subsumption, whether $C \sqsubseteq D$ holds is generally reduced to the problem whether concept $E \equiv C \sqcap \neg D$ is unsatisfiable: $C \sqsubseteq D$ if and only if E is unsatisfiable. Tableaux are good at constructing such a refutation proof. For example, whether $\forall R.A \sqcap \forall R.B$ is subsumed by $\forall R.(A \sqcap B)$ can be answered by testing the satisfiability of concept $C \equiv (\forall R.A \sqcap \forall R.B) \sqcap (\neg \forall R.(A \sqcap B))$. Figure 2.4 shows the skeleton of such a tableau expansion procedure.

In this example, disjunction normal form $\neg A \sqcup \neg B$ introduces non-deterministic branches *stage 0* and *stage 1*, so *unsatisfiability* needs to be deduced by exploring both branches and by encountering a contradiction on each branch. $\{\neg A, A\} \subseteq \mathcal{L}(y)$ on stage 0 and $\{\neg B, B\} \subseteq \mathcal{L}(y)$ on stage 1 syntactically indicate such contradictions. Therefore, concept C is unsatisfiable, which proves that $(\forall R.A \sqcap \forall R.B) \sqsubseteq \forall R.(A \sqcap B)$ holds. Intuitively, stage 0 and stage 1 in this example can be processed in parallel. In recent research on parallel tableaux, [55] showed the performance improvement

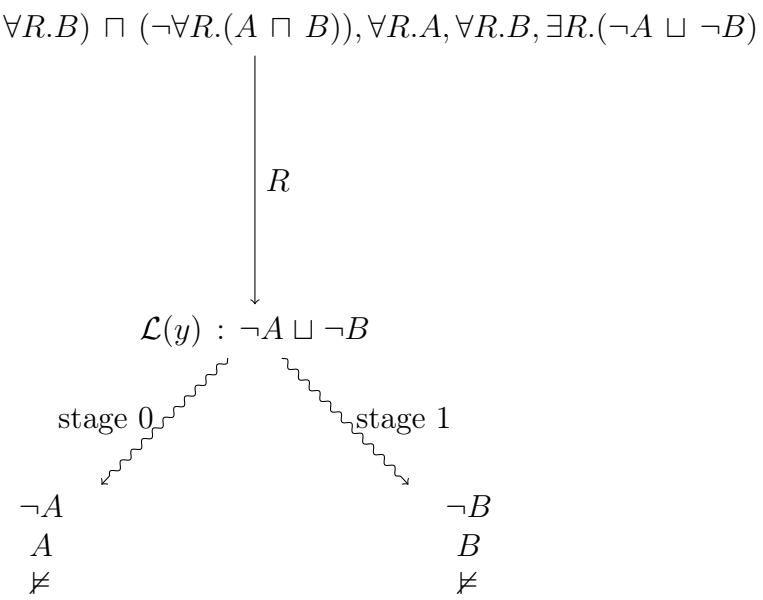


Figure 2.4: The tableau expansion for proving $(\forall R.A \sqcap \forall R.B) \sqsubseteq \forall R.(A \sqcap B)$.

gained from parallelizing disjunction stages. This example demonstrates the fundamental technique to reason DL concepts with tableaux when no TBoxes are involved.

When tableaux reasoning involves a TBox, the expansion rules listed in Table 2.1 must be modified, and U-rule is added (see [40, Section 3.3] and [37]):

U-rule	If $\{T \mid \forall T(T \in \mathcal{T} \wedge T \notin \mathcal{L}(x))\} \neq \emptyset$, then let $\mathcal{L}(x) = \mathcal{L}(x) \cup \mathcal{T}'$, where $\mathcal{T}' = \mathit{internalize}(\mathcal{T})$.

Note that the U-rule should be applied only on fresh instances. The underlying principle for the U-rule is apparent: There must be a model witnessing a consistent knowledge base $\mathcal{KB} = \mathcal{T} \cup \mathcal{A}$, and all instances in the model must agree with the axioms in $\mathcal{T}$. Internalization and other techniques to handle TBox reasoning will be addressed later in this section.

Applying the tableau rules mentioned above now should define a decision procedure, but not always. TBoxes normally contain *general* axioms, i.e. axioms that are generally introduced by *general concept inclusion (GCI)* axioms (e.g. $C \sqsubseteq D$), where

GCIs may lead to *cyclic* TBoxes. Firstly, we introduce the definition on a *acyclic* TBox.

Definition 12 *Say that C_1* ***directly uses*** *C_2 in $\mathcal{T}$ if C_2 appears on the right hand side of an axiom where C_1 is the only one on the left hand side, and* ***uses*** *is the transitive closure of the relation* ***directly uses****. Then $\mathcal{T}$ contains a* ***cycle*** *iff there exists an atomic concept in $\mathcal{T}$ that* ***uses*** *itself. Otherwise, $\mathcal{T}$ is called* ***acyclic*** *[14].*

If *cyclic* axioms are involved, this expansion procedure may actually increase the computational complexity, sometimes even interminably. For example, given $\mathcal{KB} = \mathcal{T} \cup \mathcal{A} = \{C \sqsubseteq \exists R.C\} \cup \{C(x_0)\} = \{\neg C \sqcup \exists R.C, C(x_0)\}$, Figure 2.5 illustrates its partial tableau expansion. The first time applying the $\exists$-rule and the U-rule results in a new instance $\mathcal{L}(x_1)$ and sets $\mathcal{L}(x_1) = \mathcal{L}(x_0)$. Hereby, the expansion pattern will go on endlessly.

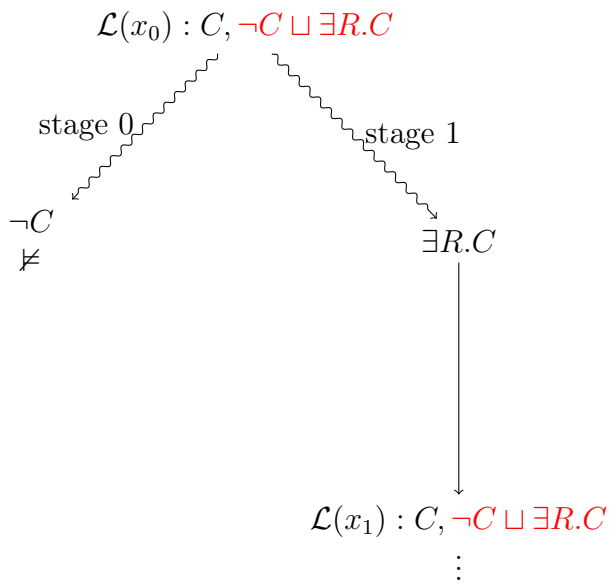


Figure 2.5: Non-termination in applying tableau expansion rules.

Blocking techniques are used to deal with cyclic tableau expansion. Four primary blocking algorithms have been proposed: *(i)* subset blocking, *(ii)* equality blocking,

(iii) pairwise blocking, and *(iv)* dynamic blocking [21]. With blocking techniques, DL tableaux give the guarantee of termination.

How is a DL TBox computed with tableau-based methods? Remember that a DL TBox comprises a set of axioms, which may include GCIs. There is no an explicit tableau rule of dealing with TBoxes allowing GCIs, e.g. $\mathcal{T} = \{A \sqcap B \sqsubseteq C, A \sqsubseteq D\}$. How are TBox axioms processed by tableaux?

A standard procedure to perform deduction over a DL terminology with tableaux starts out on *internalizing* all axioms [11, 70]. *Internalization* is essential processing although it may be avoided in some cases. Theoretically, *internalization* transforms a TBox into a monolithic concept expression and a universal role, the reflexive-transitive closure of the union of all roles, if necessary. For instance, following $\mathcal{O} \models C \equiv D \iff \mathcal{O} \models (C \sqsubseteq D) \sqcap (D \sqsubseteq C)$, the given TBox $\mathcal{T} = \{C_0 \sqsubseteq D_0, \ldots, C_n \sqsubseteq D_n\}$ can be internalized as follows:

$$\begin{aligned}\mathcal{T}' &= \mathit{internalize}(\mathcal{T}) \\ &= \{C, \forall U.C\}, \text{ where } C = (\neg C_0 \sqcup D_0) \sqcap \cdots \sqcap (\neg C_n \sqcup D_n) \text{ and } U = (R_0 \sqcup \cdots \sqcup R_m)^*\end{aligned}$$

An interpretation $\mathcal{I}$ is a model of $\mathcal{T}$, if and only if $\mathcal{I}$ is a model of $\mathcal{T}'$, so tableaux executed on $\mathcal{T}'$ generate models consistent with $\mathcal{T}$. For instance, deciding satisfiability of concept D with respect to $\mathcal{T}$ can be achieved by putting D into $\mathcal{T}'$: $\mathcal{O} \models D \iff \mathcal{T}' \cup \{D\} = \{D, C, \forall U.C\}$ is satisfiable.

Practical DL tableau reasoning on a KB generally requires a pre-processing phase in which a sequence of syntax normalization tasks are executed, such as *negation normal form (NNF)* transformation, absorption, lexical normalization, and so forth. These optimizations can usually improve reasoning efficiency. Table 2.2 lists normalization transformation routines used in $\mathcal{ALCN}$ tableaux reasoning [37]:

from	**to**
$normalize(C \equiv D)$	$normalize(C \sqsubseteq D), normalize(D \sqsubseteq C)$
$normalize(C \sqsubseteq D)$	$normalize(\neg C \sqcup D)$
$normalize(\neg\neg C)$	$normalize(C)$
$normalize(\neg C)$	$\neg normalize(C)$
$normalize(C)$	C, if C is a concept name
$normalize(C \sqcup D)$	$normalize(C) \sqcup normalize(D)$
$normalize(C \sqcap D)$	$normalize(C) \sqcap normalize(D)$
$normalize(\forall R.C)$	$\forall R.normalize(C)$
$normalize(\exists R.C)$	$\exists R.normalize(C)$
$normalize(\neg\forall R.C)$	$\exists R.normalize(\neg C)$
$normalize(\neg\exists R.C)$	$\forall R.normalize(\neg C)$
$normalize(\leq nR.C)$	$\leq nR.normalize(C)$
$normalize(\geq nR.C)$	$\geq nR.normalize(C)$
$normalize(\neg \leq nR.C)$	$\geq (n+1)R.normalize(C)$
$normalize(\neg \geq (n+1)R.C)$	$\leq nR.normalize(C)$
$normalize(\neg \geq 1R.C)$	$\forall R.\neg C$

Table 2.2: Normalization transformation routines for $\mathcal{ALCN}$.

Example: OS World The following example shows a complete *instance checking* reasoning process on a KB, *mobile world*. Firstly, the TBox defines a description of a *mobile software platform*:

$$\mathcal{T} = \{\exists drive.phone \sqcap os \sqsubseteq mobile\_platform, pc \sqcap phone \equiv \bot\}$$

Then, the ABox asserts some facts on an object $WIN7$: $\mathcal{A} = \{os(WIN7), \quad \forall drive.pc(WIN7)\}$. The query to be answered is whether $WIN7$ is a mobile platform: $?mobile\_platform(WIN7)$. The reasoning service decides the query by refutation, so it appends the query's negation to the ABox:

$$\begin{aligned}\mathcal{A} &= \mathcal{A} \cup \{\neg mobile\_platform(WIN7)\} \\ &= \{os(WIN7), \forall drive.pc(WIN7), \neg mobile\_platform(WIN7)\}\end{aligned}$$

Thus,

$$\begin{aligned}\mathcal{KB} &= \mathcal{T} \cup \mathcal{A} \\ &= \{\exists drive.phone \sqcap os \sqsubseteq mobile\_platform, pc \sqcap phone \equiv \bot, \\ &\qquad os(WIN7), \forall drive.pc(WIN7), \neg mobile\_platform(WIN7)\}\end{aligned}$$

The reasoning starts with normalizing KB; without loss of generality, no particular optimizations are employed in the reasoning demonstration, and only essential normalization as introduced in Table 2.2 is performed. Firstly, all *equality* axioms are eliminated, e.g. $pc \sqcap phone \equiv \bot$ broken down into $pc \sqcap phone \sqsubseteq \bot$ and $\bot \sqsubseteq pc \sqcap phone$; then all GCI axioms are resolved:

$$\begin{aligned}\mathcal{T} &= \{\exists drive.phone \sqcap os \sqsubseteq mobile\_platform, pc \sqcap phone \equiv \bot\} \\ &= \{\forall drive.\neg phone \sqcup \neg os \sqcup mobile\_platform, \neg pc \sqcup \neg phone \sqcup \bot, \top \sqcup (pc \sqcap phone)\} \\ &= \{\forall drive.\neg phone \sqcup \neg os \sqcup mobile\_platform, \neg pc \sqcup \neg phone, \top\}\end{aligned}$$

Some trivial application of normalization operations in Table 2.2 are omitted here. Table 2.3 demonstrates the normalized KB:

$\mathcal{KB}$	
$\mathcal{T}$	$\forall drive.\neg phone \sqcup \neg os \sqcup mobile\_platform$
	$\neg pc \sqcup \neg phone$
	$\top$
$\mathcal{A}$	$os(WIN7)$
	$\forall drive.pc(WIN7)$
	$\neg mobile\_platform(WIN7)$

Table 2.3: Normalized $\mathcal{KB}$ of *mobile world*.

Now, the tableau rules presented in Table 2.1 as well as the enhancement rule are applied to search a model for $\mathcal{KB}$. Figure 2.6 shows the expansion in which two closed but contradiction-free stages (indicated by $\models$) are established. Therefore,

WIN7 is shown not to be a *mobile_platform*.

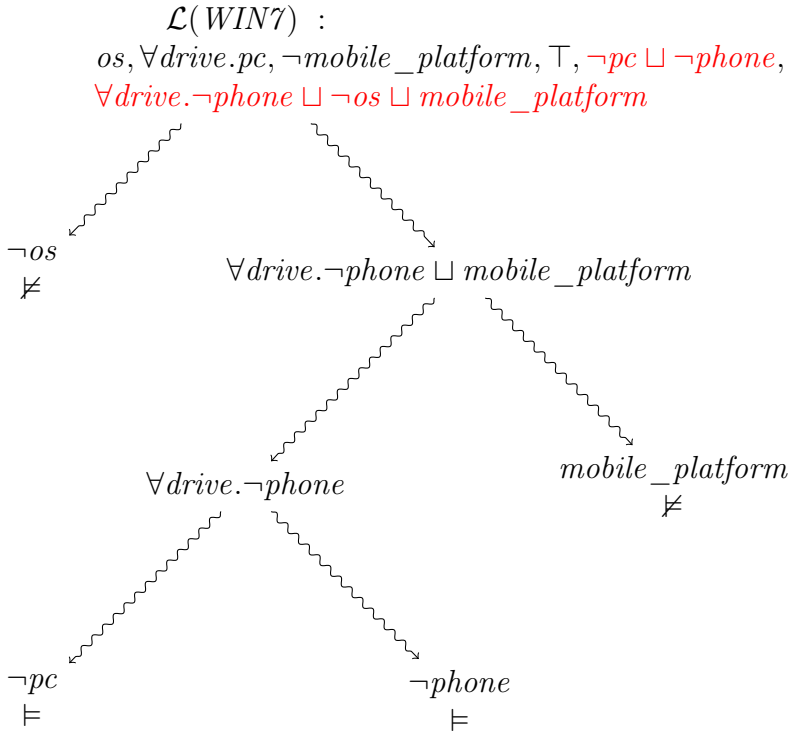


Figure 2.6: Tableau expansion of *mobile world* $\mathcal{KB}$.

This example shows the potential of improving reasoning efficiency by parallelism. Disjunctive stages are independent of each other. Each stage represents a partial model of the complement to the goal query. That is to say these sub-questions of whether a contradiction exists in some stage are independent and so can be answered in parallel.

2.3.3 Optimization

DL reasoning has successfully been applied to solving practical problems, despite the hardness of it. One reason for its successful practice is that the severe worst-case complexity rarely occurs in reality although empirical reasoning performance may vary largely with ontology constructs [14]. In addition to this, *optimization* techniques play a more important role.

Optimization techniques make workable DL reasoning systems possible. Realistic performance of DL systems is *acceptable*, as has been shown by a set of reasoners, FaCT, HermiT, Pellet, RACER, etc. All these DL reasoners not only have implemented basic tableaux but also have incorporated a number of optimization techniques. Without optimization, a tableau-based DL reasoning system is impracticable.

A preliminary optimization technique is for *normalization* (see Section 2.3.2), including NNF and axiom transformation. Lexical normalization optimization makes use of logical duality in order to detect contradictions as fast as possible. A number of normalization optimization schemes have been presented. For example, a popular axiom transformation technique is to normalize every role-existing axiom into role-value one: $\exists R.C \vdash \neg\forall R.\neg C$. The opposite transformation scheme is also acceptable. Table 2.4 shows a possible lexical normalization strategy:

from	**to**
$C \sqcup D$	$\neg(\neg C \sqcap \neg D)$
$\exists R.C$	$\neg\forall R.\neg C$
$\leq nR.C$	$\neg \geq (n+1)R.C$

Table 2.4: A lexical normalization strategy for $\mathcal{ALCN}$.

The lexical normalization optimization should be performed after NNF transformation since some lexical normalization operations do not agree with NNF. Note that axiom transformation does not mean to eliminate the original form. These optimization schemes make contradiction-detection and subsumption-checking more efficient [40].

An interesting observation is that traditional lexical normalization techniques are inclined to reduce *non-deterministic* tableaux expansion, for example, $C \sqcup D$ is transformed to $\neg(\neg C \sqcap \neg D)$ [42]. Thus, on the other side, in order to prompt *parallelism*, a lexical normalization strategy encouraging *non-determinism* should come up with a different normalization as follows:

from	to
$C \sqcap D$	$\neg(\neg C \sqcup \neg D)$
$\exists R.C$	$\neg\forall R.\neg C$
$\geq nR.C$	$\neg \leq (n-1)R.C$

Table 2.5: A lexical normalization strategy encouraging non-determinism for $\mathcal{ALCN}$.

Dramatic reasoning performance increasing comes from the optimization techniques for tableaux. As mentioned in Section 2.3.2, tableau-based DL reasoning normally starts out on *internalization*. However, it is shown that reasoning on such a monolithic internalized concept is inefficient. Disjunction branches contribute much to DL reasoning complexity while *internalizing TBox axioms* naively leads to plenty of disjunction branches:

$$internalize(\mathcal{T}) = \{(\neg C_0 \sqcup D_0) \sqcap (\neg C_1 \sqcup D_1) \cdots \sqcap (\neg C_n \sqcup D_n), \forall (R_0 \sqcup \cdots \sqcup R_m)^*.((\neg C_0 \sqcup D_0) \sqcap (\neg C_1 \sqcup D_1) \cdots \sqcap (\neg C_n \sqcup D_n))\}$$

Note that $\forall (R_0 \sqcup \cdots \sqcup R_m)^*.((\neg C_0 \sqcup D_0) \sqcap (\neg C_1 \sqcup D_1) \cdots \sqcap (\neg C_n \sqcup D_n))$ restricts $(\neg C_0 \sqcup D_0) \sqcap (\neg C_1 \sqcup D_1) \cdots \sqcap (\neg C_n \sqcup D_n)$ to interpret *every* node in tableau expansion trees.

Instead of internalization, *lazy-unfolding* is used to expand a tableau tree [12]. This technique makes it possible that tableaux expansion involves merely concerned concepts in most cases, and reduces the expansion space largely. This strategy is straightforward and sufficient for *acyclic* terminologies where concept names are defined as unique. However, cyclic definitions exist in realistic ontologies. To tackle cyclic terminologies, a TBox is divided into two disjoint parts: *general* and *unfoldable* axioms [40]. The division may be achieved via an incremental procedure, *absorption*. Then the unfoldable set is used by lazy-unfolding while axioms in the general set are internalized. Lazy-unfolding may be implemented as a set of tableaux expansion rules as listed in table 2.6 [14, 40].

A full-fledged tableau expansion procedure can not always be avoided via some

name	rule
U_1-rule	If *(i)* $A \in \mathcal{L}(x)$ and *(ii)* $(A \equiv C) \in \mathcal{T}_u$ and *(iii)* $C \notin \mathcal{L}(x)$, then let $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C\}$.
U_2-rule	If *(i)* $\neg A \in \mathcal{L}(x)$ and *(ii)* $(A \equiv C) \in \mathcal{T}_u$ and *(iii)* $\neg C \notin \mathcal{L}(x)$, then let $\mathcal{L}(x) = \mathcal{L}(x) \cup \{\neg C\}$.
U_3-rule	If *(i)* $A \in \mathcal{L}(x)$ and *(ii)* $(A \sqsubseteq C) \in \mathcal{T}_u$ and *(iii)* $C \notin \mathcal{L}(x)$, then let $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C\}$.

Table 2.6: Tableaux lazy-unfolding rules.

optimizations. Consider the following example:

$$\mathcal{L}(x) = \{C_0 \sqcup D_0, C_1 \sqcup D_1, \ldots, C_n \sqcup D_n, A \sqcap \neg A\}$$

It is possible that applying $\sqcup$-rule continues before finally encountering clash from $A \sqcap \neg A$. Maybe, such unnecessary expansion repeats $2^{(n+1)}$ times. However, these unnecessary spawned branches may be cut by some pruning techniques in many cases.

Dependency directed backtracking is proposed to prune those repeated unnecessary expansion [40]. This optimization requires each concept and role involved in expansion to maintain disjunction branching information, which is used to prune expansion later. It is known that disjunctive branching is a source of DL computational complexity, so *dependency directed backtracking* can largely reduce tableau expansion space and make acceptable DL reasoning possible.

Pseudo model is another important optimization technique. A modern DL reasoning system normally provides functionality to detect inconsistent components of an ontology. Such functionality requires deciding satisfiability of each concept name. Tableau builds a pseudo model in each satisfiability-deciding pass. *Pseudo models* built during deciding satisfiability of concepts can be cached for further DL reasoning services, such as checking concept subsumptions [35, 40]. Cached pseudo models are retrieved and merged at some point, which saves dramatically on tableau expansion

time.

For example, Figure 2.7 shows how pseudo models are used to check $\mathcal{O} \stackrel{?}{\models} A \sqsubseteq B$, given that:

$$A \equiv \exists R.C \sqcap D$$

$$B \equiv \forall S.\neg E$$

The non-subsumption can be deduced if the merged expansion tree is clash-free under certain conditions. This cache optimization technique is sound but incomplete in tests for non-subsumption [40].

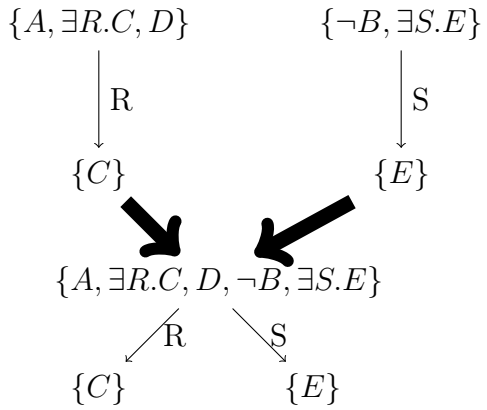


Figure 2.7: Merging models.

According to the report of [44], *semantic branching* can largely impact the tableau-based DL reasoning performance. Because the syntactic expansions of tableaux generate disjunction branches which are not necessarily disjoint, some unsatisfiable concept may be inferred repeatedly in them. For example, if C is an unsatisfiable concept expression, the clash resulted from it when deducing $\{C \sqcup D_1, C \sqcup D_2\}$ may be computed twice. Figure 2.8 (after [40]) shows the situation. *Semantic branching*,

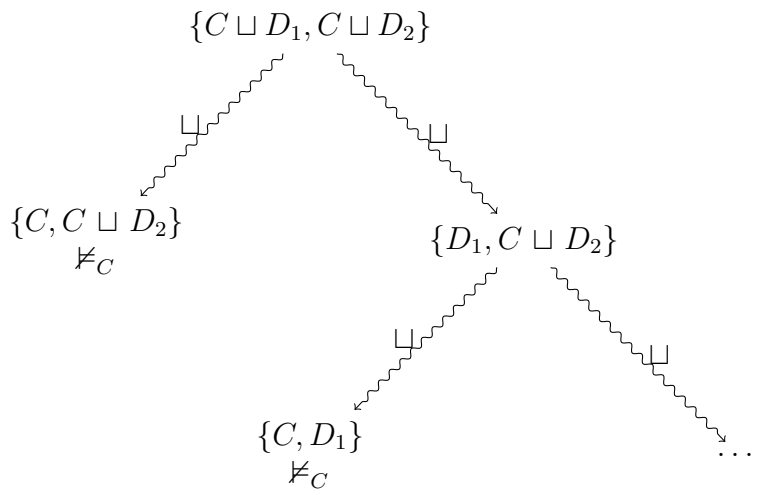


Figure 2.8: The tableau expansion of $\{C \sqcup D_1, C \sqcup D_2\}$.

a DPLL style solution, was presented to deal with this case [40, 44, 82]. This optimization technique makes use of the identification $C \sqcup D \iff C \sqcup (\neg C \sqcap D)$. In brief, when a disjunct of a concept expression disjunction leads to a clash, its complement is placed to another disjunct branch of the disjunction. The complement plays as a trigger to fire a clash early, as it avoids repeated tableau inference.

Other optimization techniques have also intensively been researched, such as *boolean constraint propagation*, and *heuristic guided searching*, and a practical DL reasoning system can not be achieved without them.

However, almost all of these optimization techniques are investigated in *serial* contexts, and large exploration space is left to *concurrency* [14, 40].

2.4 Semantic Web

DL, as a KR formalism that provides practical reasoning power for computing technologies, advances greatly with the evolution of the Internet. The Internet has aggregated an enormous amount of human knowledge, while the utilization of the knowledge is mainly confined to lexical retrieval for the moment. *Semantic web* technology

attempts to make use of contents on the Internet in a more intelligent way.

Semantic web endeavors to logically conceptualize knowledge on the Internet and to automatically deduce the knowledge. How to model and how to reason about knowledge on the Internet are the key issues of semantic web technology. As aforementioned, DL is the underpinning of semantic web technology and provides the theoretical foundation of it. DL reasoning techniques can be used by semantic web directly. The fact is what semantic web technology is using for automated reasoning in real world are those DL reasoning systems that have been developed for long time before the birth of semantic web. Although deduction techniques are part of semantic web, the substantial work of automated reasoning has been extensively researched in DL. Consequently, how to represent knowledge is more important than how to deduce knowledge in semantic web. A collection of semantic web standards have been presented to address the issue of how the knowledge on the Internet can be organized and deduced.

2.4.1 Resource Description Framework

RDF is one of the earlier endeavors as semantic web technology [1]. In order to deduce web information automatically, the first step is to endow web with *semantics*. From the point of view of *linguistic semantics*, the ternary structure of

$$(subject, predicate, object)$$

is the essential formalism of describing *meaning*, and this has also been proven at work in computer science. RDF makes use of this ternary structure, i.e. *RDF triple*, to annotate web with semantics.

An RDF triple consists of three ordered components: subject, predicate (i.e. property), and object. Any web resource which is encoded as a Uniform Resource Identifier (URI) can be described with such RDF triples. Therefore, information

can be organized as RDF graphs: Vertices are web resources, and edges are RDF predicates, which are a special type of web resources.

For example, this thesis may be described in RDF/XML as Listing 2.1:

```
100 <?xml version="1.0"?>
101 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-
        ns#"
102     xmlns:thesis-base="http://example.org/rdf/thesis#"
103     xmlns:contact="http://example.org/rdf/contact#">
104     <rdf:Description rdf:about="http://example.org/rdf/thesis/
            kejia-thesis">
105         <thesis-base:title>Parallelizing Description Logic
                Reasoning</thesis-base:title>
106         <thesis-base:author rdf:resource="http://example.org/rdf/
                person#kejia"/>
107         <thesis-base:school>Concordia University</thesis-
                base:school>
108     </rdf:Description>
109     <rdf:Description rdf:about="http://example.org/rdf/person#
            kejia">
110         <contact:full-name>Kejia Wu</contact:full-name>
111         <rdf:type rdf:resource="http://example.org/rdf/academic#
                Ph.D."/>
112     </rdf:Description>
113 </rdf:RDF>
```

Listing 2.1: RDF/XML example: Kejia's thesis.

This document describes that: the resource `http://example.org/rdf/thesis/kejia-thesis` has the type of `http://example.org/rdf/thesis#thesis`, it has other properties `http://example.org/rdf/thesis#title`, `http://example.org/`

subject	predicate	object
http://example.org/rdf/thesis/kejia-thesis	http://example.org/rdf/thesis#title	"Parallelizing Description Logic Reasoning"
http://example.org/rdf/thesis/kejia-thesis	http://example.org/rdf/thesis#author	http://example.org/rdf/person#kejia
http://example.org/rdf/thesis/kejia-thesis	http://example.org/rdf/thesis#school	"Concordia University"
http://example.org/rdf/person#kejia	http://example.org/rdf/contact#full-name	"Kejia Wu"
http://example.org/rdf/person#kejia	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://example.org/rdf/academic#Ph.D.

Table 2.7: The RDF triples in the thesis example.

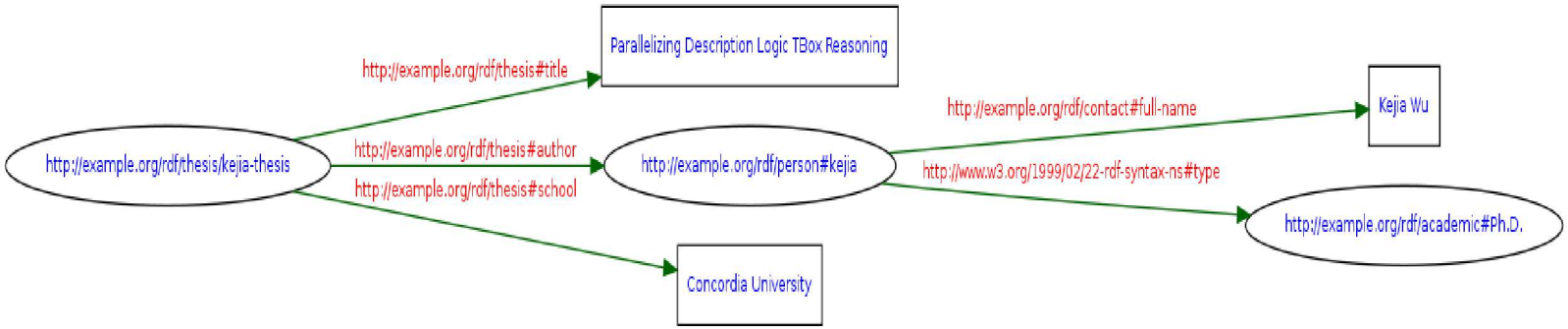


Figure 2.9: The RDF graph of the thesis example.

`rdf/thesis#author`, and `http://example.org/rdf/thesis#school`, and the corresponding resources are linked via the properties. The resource `http://example.org/rdf/person#kejia`, which is reached via the property `http://example.org/rdf/thesis#author`, is described further. This document defines a set of RDF triples, basic structures of RDF, which are listed in Table 2.7. These triples form a graph, and Figure 2.9 demonstrates it.

RDF provides a general facility to encode information, but more specific framework vocabularies are defined via RDFS. Like other XML schema standards, RDFS is used to define meta semantics for RDF vocabularies. The vocabularies used to compose an RDF document categorize description features and can be defined with RDFS. For example, the predicate "title" used in Listing 2.1 can be constrained to a specific subject and a specific object, as is shown by Listing 2.2. An RDFS document is also a validate RDF one, so it defines a set of RDF triples and a graph, too, as are shown by Table 2.8 and Figure 2.10.

```
100 <?xml version="1.0"?>
101 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-
        ns#"
102     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
103   <rdf:Property rdf:about="http://example.org/rdf/thesis#
          title">
104       <rdfs:domain rdf:resource="http://example.org/rdf/
              thesis#thesis"/>
105       <rdfs:range rdf:resource="http://www.w3.org/2001/
              XMLSchema#string"/>
106   </rdf:Property>
107 </rdf:RDF>
```

Listing 2.2: RDFS/XML example: Kejia's thesis.

subject	predicate	object
http://example.org/rdf/thesis#title	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/1999/02/22-rdf-syntax-ns#Property
http://example.org/rdf/thesis#title	http://www.w3.org/2000/01/rdf-schema#domain	http://example.org/rdf/thesis#thesis
http://example.org/rdf/thesis#title	http://www.w3.org/2000/01/rdf-schema#range	http://www.w3.org/2001/XMLSchema#string

Table 2.8: The RDF triples in the thesis schema.

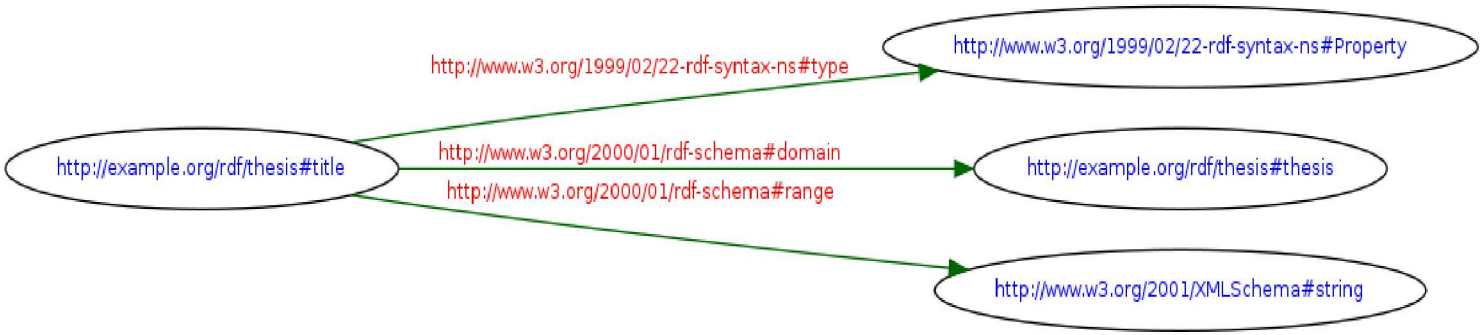


Figure 2.10: The RDF graph of the thesis schema.

RDF inference is made over triples graphs with a set of deduction rules. For example, given the rule $\frac{(s\ p\ o),(p\ \textit{has-domain}\ c)}{(s\ \textit{has-type}\ c)}$ and the RDF graphs shown in Figure 2.9 and 2.10, we can infer `http://example.org/rdf/thesis/kejia-thesis` must have a type of `http://example.org/rdf/thesis#thesis`:

$$@prefix\ \ tb:\ \ < http://example.org/rdf/thesis/ >\ \ .$$

$$\frac{(tb{:}kejia\text{-}thesis\ \ tb{:}title\ \ ''\ldots''), (tb{:}title\ \ has\text{-}domain\ \ tb{:}thesis)}{(tb{:}kejia\text{-}thesis\ \ has\text{-}type\ \ tb{:}thesis)} \tag{2.1}$$

RDF can describe limited simple relationships between web resources. Although RDF can express basic class hierarchies, it can not express complex semantics, such as *negation*. For example, RDF even can not express $\bot \equiv \neg\top$. The core language constructs of RDF are listed in Table 2.9. More powerful expressivity can be obtained with OWL, the syntax and semantics of which are established on the basis of RDF.

rdfs:Class	rdf:Property	rdfs:Resource
rdfs:Literal	rdfs:Datatype	rdf:XMLLiteral
rdfs:range	rdfs:domain	rdf:type
rdfs:subClassOf	rdfs:subPropertyOf	rdf:Statement
rdf:subject	rdf:predicate	rdf:object

Table 2.9: The core language constructs of RDF.

2.4.2 Web Ontology Language

OWL is becoming increasingly important nowadays and has become an important semantic web infrastructure [3]. Compared with RDF, OWL provides more powerful expressivity in order to represent complex knowledge. For example, we know that a Ph.D. thesis is a type of thesis and must be completed by some Ph.D. This piece

of knowledge can hardly be described with RDF, but OWL is good at representing it. Listing 2.3 shows the corresponding OWL/XML snippet. Some interesting OWL language constructs which is absent from RDF are listed in Table 2.10.

```
100 <SubClassOf>
101     <Class IRI="#phd-thesis"/>
102     <ObjectSomeValuesFrom>
103         <ObjectProperty IRI="#has-author"/>
104         <ObjectIntersectionOf>
105             <Class IRI="#author"/>
106             <ObjectSomeValuesFrom>
107                 <ObjectProperty IRI="#has-academic-degree"/>
108                 <Class IRI="#phd"/>
109             </ObjectSomeValuesFrom>
110         </ObjectIntersectionOf>
111     </ObjectSomeValuesFrom>
112 </SubClassOf>
```

Listing 2.3: OWL/XML example: Ph.D. thesis.

disjointWith	equivalentClass	intersectionOf
unionOf	complementOf	disjointUnionOf
AllDisjointClasses	members	allValuesFrom
someValuesFrom	cardinality	minCardinality
maxCardinality	TransitiveProperty	SymmetricProperty
FunctionalProperty	InverseFunctionalProperty	inverseOf
AsymmetricProperty	ReflexiveProperty	IrreflexiveProperty
propertyDisjointWith	AllDisjointProperties	propertyChainAxiom
maxQualifiedCardinality	minQualifiedCardinality	qualifiedCardinality

Table 2.10: Partial core language constructs of OWL.

It is well known that OWL's theoretical base is DL. It is straightforward to establish a mapping between both. For example, OWL axioms $SubClassOf(C\ D)$, $ObjectSomeValuesFrom(R\ C)$, and $ObjectIntersectionOf(ObjectComplementOf(C)\ ObjectAllValuesFrom(R\ D))$ have the same semantics in DL as $C \sqsubseteq D$, $\exists R.C$, and $\neg C \sqcap \forall R.D$, respectively.[2] The OWL/XML snippet shown in Listing 2.3 can be translated into the following DL description:

$$phd\text{-}thesis \sqsubseteq \exists has\text{-}author.(author \sqcap (\exists has\text{-}academic\text{-}degree.phd)) \tag{2.2}$$

Actually, OWL fragments correspond to DL fragments. Table 2.11 shows the mapping [37].[3] DL, especially its fragment $\mathcal{SHIQ}$, lays the foundation of OWL [78].

OWL Profile	**DL Fragment**
OWL Full	not DL
OWL DL	$\mathcal{SHOIN}(D)$
OWL Lite	$\mathcal{SHIF}(D)$
OWL 2 Full	not DL
OWL 2 DL	$\mathcal{SROIQ}(D)$
OWL 2 EL	$\mathcal{EL}^{++}$
OWL 2 QL	a variant of *DL-Lite*$_{\mathcal{R}}$ (see [19, 68])
OWL 2 RL	*Description Logic Programs (DLP)* (see [33])

Table 2.11: OWL maps to DL.

Thus, OWL is an application of DL, while at the same time OWL promotes development of DL.

Modern DL systems now can focus on reasoning over OWL ontologies. In the past, when an ontology was encoded in several languages, DL reasoners had to deal with multiple popular ones. This problem has been solved by introducing OWL since

[2]Here, OWL 2 *functional-style* syntax is used (see [2]).

[3]*OWL 2 Full* exists in semantic web with the most descriptive expressivity, and some reasoning services, such as *deciding TBox consistency*, in *OWL 2 Full* are undecidable (see [60]).

a multitude of modern computational ontologies are now encoded in OWL. So, DL research lines up with the semantic web technology, and the mature DL reasoning techniques can be used for semantic web deductions.

2.4.3 Others

RDF and OWL consist of the core semantics representation facilities of semantic web technology. With RDF and OWL, automatic deduction becomes possible in semantic web technology. There are also other significant semantic web standards, such as SPARQL. SPARQL provides the query functionality mainly for RDF.

SPARQL uses SQL-style syntax, and its typical implementation may utilize a set of computational operations, SPARQL *algebra*, and similar systems are generally used by implementations of SQL [5]. With SPARQL, normal users can retrieve information via a convenient means, instead of constructing complex RDF graphs. For example, the SPARQL query illustrated by Listing 2.4 may return the theses written by a specific author.

```
100 PREFIX  thesis-base: <http://example.org/rdf/thesis/>
101
102 SELECT  ?title
103 WHERE {
104     ?thesis thesis-base:author "Kejia Wu" .
105     ?thesis thesis-base:title ?title
106   }
```

Listing 2.4: SPARQL example: thesis.

Before the current semantic web proposals become available, there have existed a number of knowledge representation and automatic inference methodologies, many of which use *first-order logic rule* as their reasoning technique, such as *datalog*, and how to make use of those existing knowledge modeling technologies is a topic

of semantic web research. The ongoing *Rule Interchange Format (RIF)* standard is an effort in this direction [4]. RIF makes it possible to combine traditional rule-based technologies with available semantic web facilities, and therefore rule-reasoning engines may be used for semantic web deduction.

These standards make up the fundamental semantic web facilities. Meanwhile, semantic web technology produces enormous knowledge bases to which researchers need to seek for *scalable* solutions. *Concurrent computing* may be a candidate solution and is attracting many DL researchers.

2.5 Concurrent Computing

A number of *concurrent computing* methodologies are presented in recent years. *Parallel computing* and *distributed computing* are the well-known folklore results of *concurrent computing*. However, the distinction between them is obscure, as was discussed in [28, Section 1.7]. [52, Section 1.5] differentiated *shared memory systems* from *non-shared memory systems*. In our research, the term *parallelism* mainly denotes *parallel algorithms* designed for *shared-memory parallel systems*, and the term *distributed systems* denotes the ones in which *shared address space* is not necessarily supported by the underlying architecture.[4]

Parallelism as an optimization technique is not an easy solution. Many practical issues on parallelism must be researched. For example, *thread-locking* is the preferred method to monitor a *critical code section* in shared-memory parallel reasoning. However, thread-lock model's intrinsic rejection of shared states diminishes its better application potential in computing. The *serialization* degree of deduction increases while applying thread-locking; adding the expense of controlling threads (either recreating or re-using), inappropriately using thread-lock in complex computing, such as computation of knowledge reasoning systems like DL reasoners, introduces

[4]The differentiation is not identical to that of [52].

synchronization-related performance bottlenecks [77]. So, the overhead issue caused by threading must be taken into account.

Design patterns of parallel algorithms are valuable experience that is used to avoid common difficulty of parallelism and are valuable in concurrent computing [54, 80].

Producer/Consumer In real world, some programs process data which is the computation results of other programs. The programs generating data can be view as *producers*, and the data is fed to *consumers*. A typical scenario is the usage of *pipe* operation on UNIX. One producer may mapped to more than one consumer, and one consumer may get data from more than one producers. When a group of producers feed data to a group of consumers, parallelization may play a role. Generally, in such a case, the data processed by the producers are independent, the data used by consumers are independent among them, and thus the data can be processed in parallel on the two stages. For example, a video encoding program, which reads in data, encode it, and compresses it. It normally processes data chunk by chunk. For each chunk, the program must encode it firstly and then compress it. The encoding program is a producer, and the compress program is a consumer. A video file can be divided into a number of chunks, a set of producers can be parallelized, and so consumers.

Fork/Join This pattern uses *divide and conquer* strategy to solve a type of problems that can be divided into sub-problems, and all results of a sub-problems need to be jointed together for completion of the calculation. The divide operation is generally recursive. The computation on each fork is normally independent to the others, so this is a popular pattern employed in concurrent computing.

MapReduce This pattern is popular on cluster-based computing today, and it can be used in shared memory parallelization. MapReduce uses a manager agent to read

in a problem, normally a key/value pair.[5] The manager agent divides the problem into sub-problems, which are generally key/value pairs, and dispatches them to a set of worker agents. This process is *map*. The manager agent finally needs to collect the outputs of worker agents and to combine them in terms of keys. This process is *reduce*.

Speculation There generally exist more than one strategies to solve a problem in real world. Similarly, in complex computation, several algorithms can be used to decide a problem. It is hard to say which algorithm is best to solve a specific problem. For example, the practical efficiency of sort algorithms depends on specific data. A universal solution to pick up the best one is trying through all candidates. Obviously, applying all strategies in a parallel way to a problem may be more efficient than in a sequential way. This way of picking up strategies is *speculation*.

Replication It is a pretty common case that multiple computation agents manipulate shared resources in concurrent computing. *Locking* is the universal solution to assure *synchronization* in those cases. However, locking may largely degrade the concurrency performance in some cases. *Replica* of shared data may assure both synchronization and performance by making local copies.

Active Object [53]

When there are a lot of operations, it becomes problematic to manage them in an efficient way. *Active object* pattern makes operation request and operation execution be separated in different threads. The components like *proxy* and *scheduler* are used to manage operations.

Thread Pool In concurrent computing, the cost of creating and destroying processes or threads is significant. In real world, the maximum number of computation

[5]Here, an *agent* may be a computation node, a process, or a thread.

resources is constrained. *Thread pool* pattern is for decreasing the cost of creating/destroying threads and for re-using threads in shared-memory concurrent computing.

Double-checked Locking Repeated operations should always be avoided. This issue is solved by *locking* mechanism. However, the cost of acquiring and releasing a lock is generally expensive, so acquiring a lock should be avoided as much as possible, too. The *double-checked locking* satisfies the requirement. This pattern firstly checks whether a shared resource has been initialized, and the resource is consumed if it is available. If the needed resource has not existed, the thread will try to compute it, and the computation is synchronized by a lock. Listing 2.5 illustrates the scenario.

```
100  if (foo = ∅) {
101    lock {
102      if (foo = ∅) {
103        foo ← initialize_foo();
104      }
105    }
106  }
```

Listing 2.5: The *double-checked locking* pattern.

Chapter 3

Related Work

DL is advancing along with extended expressivity and novel reasoning methods. Although *tableau* is a sound and complete DL reasoning solution, a gap exists between theory and practice. A workable DL reasoning system depends on a number of optimization techniques. A naive DL reasoner without any optimizations has no guarantee of a practical termination. Besides extensive research in serial contexts, some parallel DL reasoning techniques seeking for scalable solutions are being presented. In this chapter, the work on DL formalism, on those cardinal techniques for workable DL reasoning, and on concurrent DL reasoning is reviewed.

DL consists of a family of languages and is evolving by extended expressivity, as aforementioned. The overall view on the DL formalism, such as syntax, semantics, notation, conversion, and extension, is illustrated in [14]. [43] witnesses the latest major DL language, $\mathcal{SROIQ}$, as well as the preliminary tableau-based reasoning technique for it. The main extension to achieve $\mathcal{SROIQ}$ is via a set of functions applied to roles, and thus an *RBox* is mentioned in [43]. $\mathcal{SROIQ}$ is known to be ready for an important OWL fragment, OWL DL 2 (see Table 2.11). Actually, $\mathcal{SROIQ}$ is directly extended from $\mathcal{SHOIQ}$, another weighty DL fragment which allows for role *transitivity*, *inversion*, and *subsumption* and was formalized with tableau in [47]. Based on $\mathcal{SHOIQ}$, $\mathcal{SROIQ}$'s expressivity is augmented via a set of

constructors allowing: *(i)* role disjointness, *(ii)* role reflexivity, role irreflexivity, and role anti-symmetry, *(iii)* negated role assertions, role inclusion axioms of the form $R \circ S \sqsubseteq R$ and $S \circ R \sqsubseteq R$, *(iv)* the universal role, and *(v)* concepts of the form $\exists R.Self$ to express *local reflexivity* [43].

Most essential optimization techniques in tableau-based DL reasoning thus far have been summarized in [14, Chapter 9] and quite a few were originally investigated by [40]. Those cardinal optimization techniques are able to improve DL reasoning efficiency dramatically, and so most modern DL reasoning systems adopt them or some of them. *FaCT*, the research result of [40], employs and analyzes most of those optimization techniques, including: *(i)* lexical normalization, *(ii)* semantic branching, *(iii)* simplification, *(iv)* dependency-directed backtracking, *(v)* heuristic guided search, and *(vi)* caching [44]. The well known DL reasoner *FaCT++* uses these optimization techniques as well as additional ones, such as *lazy unfolding*, *absorption*, and *blocking*, to deal with $\mathcal{SHOIQ}$, as is addressed in [82, 83]. Among these optimization methods, a caching technique, *model-merging*, can expedite tableau expansion dramatically and was intensively investigated by [35].

The *absorption* technique can largely reduce disjunctions, a source of DL reasoning complexity, introduced by GCIs, and has been investigated by [40, 41, 47, 48, 81, 84]. Non-determinism introduced by disjunction branches can be totally avoided in the case of processing axioms which can be translated into *Horn clauses*. *Hypertableau* DL reasoning presented by [64] makes use of this observation to decrease disjunction branches [15, 61, 62, 63, 73]. Furthermore, hypertableau of [64] can also deal with the inefficiency of the so called *and-branching* introduced by *value exist rule* and *at-least cardinality restriction*.

A number of classification optimization algorithms have been researched. *Top-search* and its duality *bottom-search* are elementary methods handling classification and were addressed originally by [56] and were extended by [12]. *Reflexive transitive closure* and *told subsumption* can assist in pruning subsumption tests so as to accel-

erate classification, as was discussed even early in [12]. [72] presented a method of exploiting this idea. This algorithm tries to make use of reflexive transitive closure over *told* and *possible* subsumptions so that potential non-subsumptions can be figured out fast. However, [29] pointed out the method of [72] was actually too naive to expedite the calculation much, and completed it with more sophisticated mechanism to utilize reflexive transitive closure. Classification research mentioned so far does not involve concurrency. These optimizations can improve DL reasoning efficiency largely, while few of them have been investigated under a *concurrency* context. However, the canonical top-search algorithm, as well as its duality bottom-search, can be executed in parallel potentially. [8, 9] have worked on parallelizing the dual procedures, top-search and bottom-search, and the experimental results promise the feasibility of parallelized DL reasoning.

[55] completed a parallel $\mathcal{SHN}$ reasoner. This reasoner achieved processing *disjunction* and *at-most cardinality restriction* rules in parallel, as well as primary DL tableaux optimization techniques. The experimental results of the reasoner show noticeable performance improvement in comparison with non-parallel reasoners. However, the involved test cases are either small or restricted. For example, test case 2 employed in [55] constructed a concept which was the union of eight concepts excluding any non-determinism.

[16] proposed two hypotheses on parallelized ontology reasoning: *independent ontology modules* and *a parallel reasoning algorithm*. *Independent ontology modules* strive for structuring ontologies as modules which can be computed in parallel. The idea of partitioning ontologies into modules is supported by [32], [30], and [31]. According to the second hypothesis, extensive research on parallelized logic programming does not contribute much to DL reasoning. Furthermore, some DL fragments, without disjunction and at-most cardinality restriction constructors, do not profit much by parallelizing non-deterministic branches in tableau expansion.

[58] applied a *constraint programming* solver, *Mozart*, to $\mathcal{ALC}$ tableau reasoning

in parallel, and this idea was implemented by [57]. The experimental results show scalability to some extent.

[49, 74] proposed a *consequence-based* DL reasoning method, mainly dealing with Horn ontologies. Based on the consequence-based reasoning and the work of [10], [50, 51] achieved a substantial reasoner that can classify $\mathcal{EL}$ ontologies *concurrently*.

Compared with TBox, tableau-based ABox reasoning research literature is not so much. [34] originally presented a tableau calculus to decide the ABox consistency problem. The work researched ABox reasoning on $\mathcal{ALCNH}_{\mathcal{R}^+}$. Further investigation about ABox reasoning on $\mathcal{SHIQ}$ was conducted by [46]. For the moment, [25] has completed research on nominals and qualified cardinality restrictions of DL, in particular $\mathcal{SHOQ}$.

Recently, some research focuses on how DL reasoning is applied to industrial standards, such as RDF and OWL, and on issues arising in the process. That trend leads to the research upsurge of reasoning about massive web ontologies in a scalable way. MapReduce [65], ontology mapping [38], ontology partitioning [31], rule partitioning [76], *distributed hash table (DHT)* [26], swarm intelligence [20], etc., are presented for that goal. Some of the researches tried to apply concurrent computing to DL reasoning, but the efficiency of these methods is not prominent; few of them were researched under a *shared-memory parallelism* context, and we believe that the *non-shared memory distributed computing* scheme they used is not suitable for DL reasoning, while *shared-memory parallelism* can play a role in improving DL reasoning efficiency. Therefore, our research was conducted under this premise: using shared-memory parallelism.

Chapter 4

Parallelizing Tableau-based Classification

4.1 Introduction

TBox classification is a core inference service of DL reasoners. An intention of using KR technologies like DL is to construct knowledge taxonomies. A DL TBox taxonomy describes whether a concept is subsumed by another one, i.e. a *concept subsumption relationship*. TBox classification computation figures out all subsumption relationships entailed in a knowledge base. Such a test of calculating a subsumption relationship between two concepts generally uses tableau-based algorithms and is expensive. Meanwhile, a concept subsumption relationship test is independent of the others. Therefore, DL TBox classification can be computed in a concurrent way. This research applies *concurrent computing* to tableau-based DL TBox classification. A parallel classification algorithm and corresponding architecture have been developed. The experiments showed that scalable reasoning performance can be gained by the parallel algorithm.

4.2 Architecture

Some distinct features are of concern when developing a parallel reasoner. The design must assure approachable performance improvement from parallelism. Hereby, some aspects of a parallel DL reasoner's architecture are different from sequential ones. The essential trade-offs for devising $\mathcal{D}$eslog are presented in this section.

4.2.1 Framework

The shared-memory parallel reasoner $\mathcal{D}$eslog consists of three layers: *(i) pre-processing* layer, which adapts OWL representation to internal data structures; *(ii) reasoning engine* layer, which performs the standard DL services and is composed by two key components, the *service provider* and the *tableau rule applier*; *(iii) post-processing* layer, which collects, caches, and saves reasoning results; *(iv) infrastructure* layer, which provides core components and utilities, such as structures representing concepts and roles, and the object copy tool. Figure 4.1 shows the overview of the framework.

OWL ontology data is read into the pre-processing layer first. Typical pre-processing operations, such as NNF, axiom re-writing, and axiom absorption, are executed in this layer. The reasoner's run-time options, such as service, thread number, and rules application order, are also set up in this layer. We implement this layer with the OWL API [39]. The pre-processed data is fed to the reasoning engine.

The reasoning engine performs primary inference computation. The first key component of the reasoning engine is the service provider. As with popular DL reasoning systems, $\mathcal{D}$eslog provides standard DL reasoning services, such as testing TBox consistency, concept satisfiability, etc. As we know, these services may depend on each other. In $\mathcal{D}$eslog, the classification service depends on subsumption, and the latter depends on the satisfiability service. The service provider uses a set of tableau-based deduction calculus to complete computing.

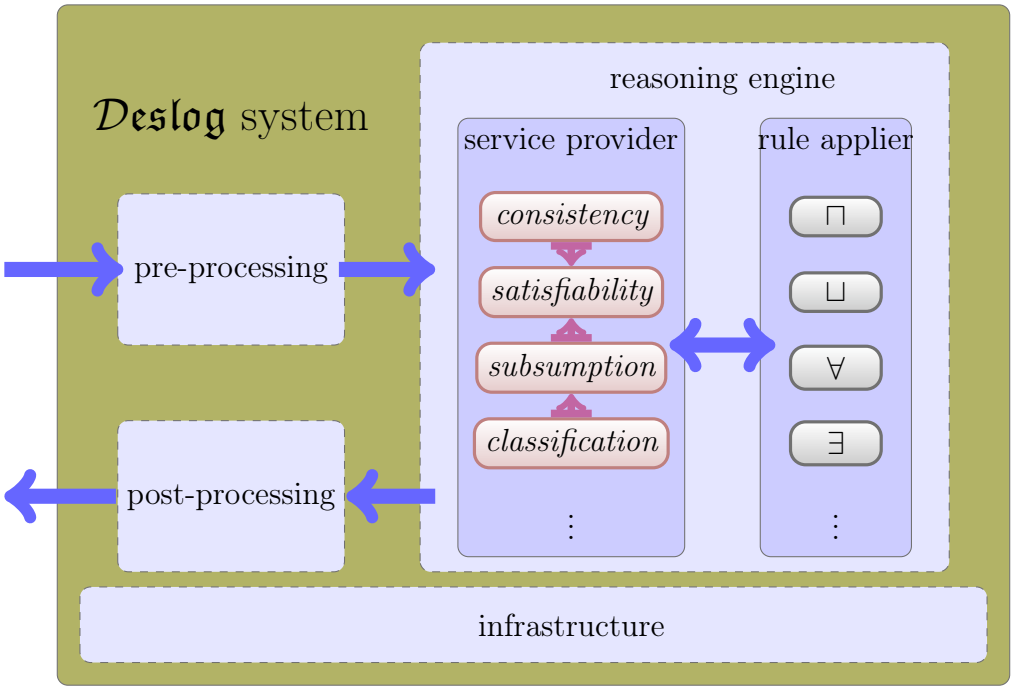


Figure 4.1: The framework of Deslog.

The reasoner adopts tableaux as the primary reasoning method, which is conducted by another key component of the reasoning engine, the tableau expansion rule applier. The main function of this component is to execute tableau rules in some order to build expansion forests. In $\mathcal{D}\mathfrak{eslog}$, tableau expansion rules are designed as configurable plug-ins, so choosing which rules in what application order can be specified flexibly. At present, $\mathcal{D}\mathfrak{eslog}$ has implemented the $\mathcal{ALC}$ tableau expansion rules [14].

All the three layers mentioned above use the facilities provided by the infrastructure layer. All common purpose utilities reside on this layer. For example, the threads manager, the global counters, and the *globally unique identifier (GUID)* generator. In addition, the key data structures representing DL elements and basic operations on them are built in this layer, as is addressed in Section 4.2.2.

4.2.2 The Key Data Structures

Different from popular DL reasoning systems, $\mathcal{D}\mathfrak{eslog}$ aims improving reasoning performance by employing parallel computing, while data structures employed by sequential DL reasoners are not always suitable for parallelism.

The tree structure has been adopted by many tableau-based reasoners. However, a naive tree data structure introduces a data race in a shared-memory parallel environment and can hardly play a role in such a concurrency setting. Therefore, we need to devise more efficient structures in order to reduce shared data as much as possible.

The new concurrent data structures must provide support to DL tableaux. One important function of the trees is to save non-deterministic branches generated during tableau expansion. Non-deterministic branches are mainly produced by the *disjunction rule* and the *at-most number restriction rules*. To separate non-deterministic branches into independent data vessels, which are suited to be processed in parallel, we adopt a list-based structure, *stage*, to maintain single non-deterministic branches,

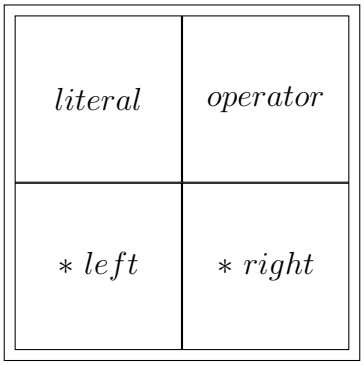


Figure 4.2: $\mathcal{D}\mathfrak{eslog}$ data structure—concept.

and a queue-based structure, *stage pool*, to buffer all branches in a tableau. Every stage is composed by the essential elements of a DL ontology, concepts and roles.

As with any DL reasoners, the representation of concepts and roles are fundamental design considerations. The core data structure of $\mathcal{D}\mathfrak{eslog}$ is a four-slot list representing a *concept*. *Literal* uniquely identifies a distinct concept. *Operator* indicates the dominant DL constructor applied to a concept. Available constructors cover intersection, disjunction, value existing, value restriction, and so on, and this slot is possibly empty. The remaining two slots hold pointers to extend nested concept definitions, namely *left* and *right*. Figure 4.2 illustrates a DL concept encoded with the $\mathcal{D}\mathfrak{eslog}$ protocol.

Roles in $\mathcal{D}\mathfrak{eslog}$ are handled as a special type of concepts and have a similar structure as concepts. For instance, the DL expression $\forall R.(A \sqcap B)$ is encoded as demonstrated by Figure 4.3. Further properties needed for describing a role can be added to the generic structure, e.g. the number restriction quantity. Furthermore, a role data structure is also backed by a list that records instance-pairs. With this design, DL concepts can be lined up seamlessly. Instances (i.e. labels in tableau expansion) are lists holding their typing data, concepts. There are also helper facilities, such as a role pool and an instance pool, which are useful to accelerate indexing objects.

A notable point on encoding is expressing the *complement* of an atomic concept

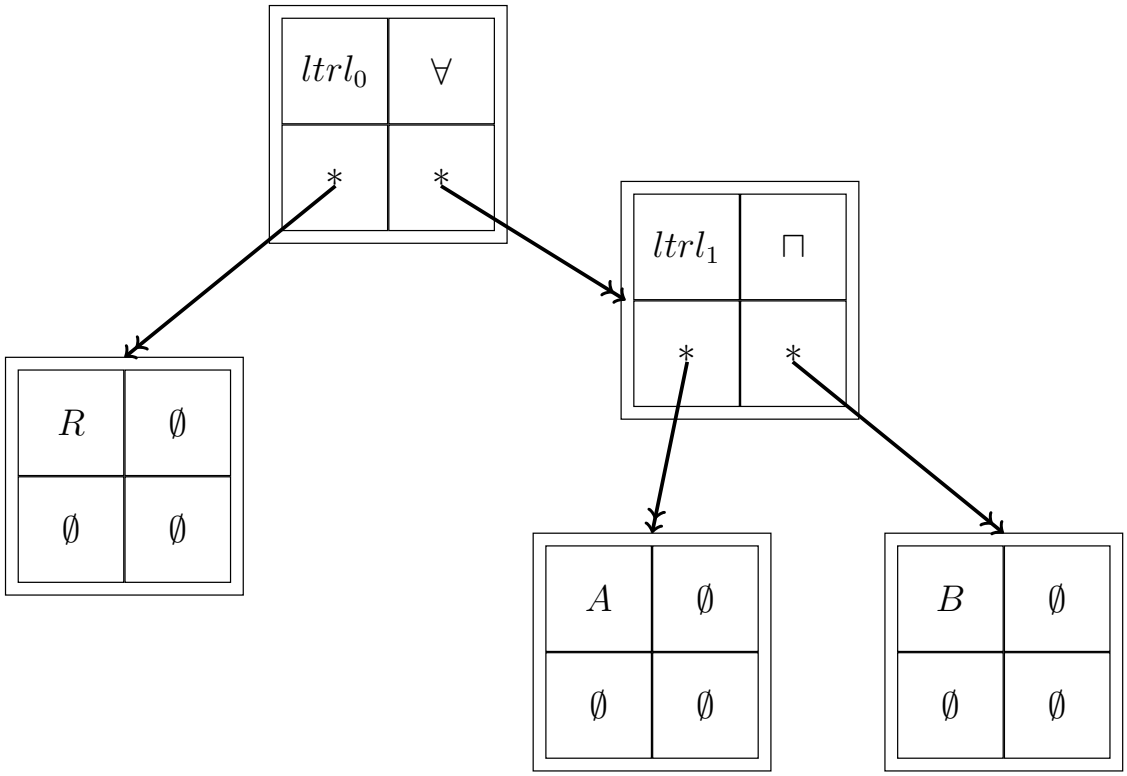


Figure 4.3: $\mathcal{D}\mathfrak{eslog}$ data structure—DL expression $\forall R.(A \sqcap B)$.

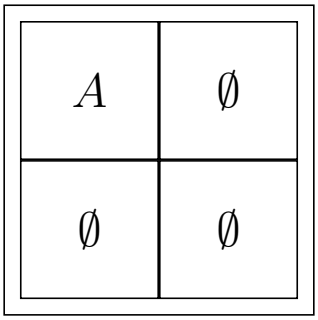


Figure 4.4: Atomic concept A

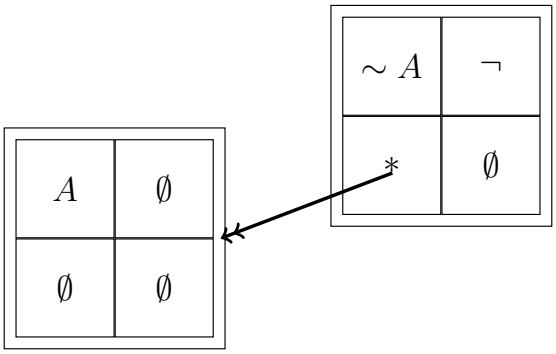


Figure 4.5: Atomic concept $\neg A$

(i.e. concept name). According to $\mathcal{D}\mathfrak{eslog}$ conversion, an atomic concept's encoding is indicated by Figure 4.4 and 4.5.

The principle of $\mathcal{D}\mathfrak{eslog}$'s design is to model objects and behaviors involved in DL reasoning as independent abstractions as much as possible in order for concurrent processing. For instance, branches created during tableau expansion are encapsulated into standalone objects. Thus, a whole tableau expansion forest is designed as a list of branch objects. Tableaux expansion rules and even some key optimization techniques are also designed as independent components. In a concurrent manner, DL reasoning is achieved by invoking corresponding components on the layers.

4.2.3 Implementation

As aforementioned, multi-processor computers are becoming main stream, it is required to maximally utilize all processors of a computer, and shared-memory multi-threading computing is quite suitable for this purpose. From a converse point of view, it is multi-processor computers that actually release the power of the shared-memory multi-threading computing.

One significant aspect of this research is investigating whether some important DL reasoning optimization technique is suited to be implemented in a parallel reasoner and how it should be adapted if plausible. $\mathcal{D}$esl$\mathfrak{og}$ has practised the following optimization techniques:

1. **Lazy-unfolding** This preliminary technique enables a reasoner to unfold a concept only when necessary, so a number of time is saved on [12].

2. **Axioms absorption** The *disjunctive branches* introduced by naively internalizing all subsumptions and general axioms declared in a TBox is the primary source of costly computing. With the axioms absorption technique, a TBox is separated into two sub-boxes, the *general* and the *unfoldable*. Then using internalization to process the general and using lazy-unfolding to process the unfoldable can reduce reasoning time dramatically [40, 84].

3. **Semantic branching** This DPLL style technique can prune some disjunctive branches so that a number of reasoning time is saved on by avoiding computing the same problem repeatedly [40].

Other primary optimization techniques, such as *dependency directed backtracking* [14, Chapter 9][40] and *model merging* [35] are being implemented. It is noticeable that not all significant optimization techniques are suitable for a concurrency environment: Some of them can not avoid depending on complex shared data and so may degrade the performance of a concurrent program a lot. Based on these elemental

techniques, we completed a suite of standard TBox reasoning services.

The current system implements a parallel $\mathcal{ALC}$ TBox classifier. It can concurrently classify an $\mathcal{ALC}$ terminology. The parallelized classification service of $\mathcal{D}\mathfrak{eslog}$ computes subsumptions in a brutal way [12]. It is obvious that the algorithm is sound and complete and has order of n^2 time complexity for calling subsumption tests in a sequential context. In order to figure out a terminology hierarchy, the algorithm calculates the subsumptions of all atomic concepts pairs. A subsumption relationship only depends on the involved concepts pair, and does not have any connections with the computation order. Therefore, the subsumptions can be computed in parallel, and the soundness and completeness are retained in a concurrent context.

A difficult issue in implementing a parallel DL reasoner is managing overhead. This issue is relatively easy for high level parallel reasoning, where multiple threads mainly execute reading operations on some shared data, so we implemented the parallel classification service firstly.

Besides the high-level parallelized service, classification, the low level parallelized processing is being developed. In the architecture of $\mathcal{D}\mathfrak{eslog}$, the classification service uses subsumption, and subsumption uses satisfiability. The low level parallel reasoning stresses on the parallel satisfiability test. Specifically, low level parallel reasoning focuses on dealing with the non-deterministic branches, which are represented as stages in $\mathcal{D}\mathfrak{eslog}$.

It seems easy to process stages in parallel, but much endeavor is needed to achieve satisfying scalability via such concurrency.[1] The first noticeable fact is that from a root stage every stage may generate new stages. At present, our strategy is using one thread to process one stage. That means the stage buffer, the stage pool, is frequently accessed by multiple threads. That accessing includes both writing and reading shared data much often. So, designing a high-performance stage buffer and

[1] In this research, *scalability* is the ability that the performance of reasoning about the same problem is improved to some extent with the increase of used processors.

efficient accessing schemes is the essential condition for the assurance of scalable performance improvement. Otherwise, instead of performance improvement, parallelism only results in overhead. We had worked on devising efficient low-level parallelized reasoning.

Although there exist robust shared-memory concurrent libraries available, such as C++ *Boost.Thread* library and Java *concurrent* package, according to our experience, using these concurrent data structures immoderately degrades performance much. Therefore, one needs to design sophisticated structures which had better avoid shared data, or which do not access shared data frequently.

4.3 Evaluation

$\mathcal{D}$eslog is being implemented in Java 6 in conformity with the aforementioned design. The *parallelism* of $\mathcal{D}$eslog is based on a *multi-threading model* and aims at exploiting *symmetric multiprocessing (SMP)* coming along with the popularity of *multi-processor computing* facilities. The system is implemented in Java 6 for its relatively mature parallel ecosystem.[2] Specifically, the *java.util.concurrent* package of Java 6 is utilized. In this research, each Java thread is mapped to a native operating system thread.

We have conducted some experiments to show that a shared-memory parallel tableau-based reasoner can gain a scalable performance improvement.

4.3.1 Experiment

The classification service of $\mathcal{D}$eslog can be executed concurrently by multiple threads. We conducted a group of tests, and they show that $\mathcal{D}$eslog has an obvious scalability.

All tests were conducted on a 16-core computer running Solaris OS and Sun Java

[2]All components and sources of the system can be obtained at *http://code.google.com/p/deslog/*.

6. Many of the test cases were chosen from *OWL Reasoner Evaluation Workshop 2012 (ORE 2012)* data sets. We manually degraded some test cases' expressivity to $\mathcal{ALC}$ so that $\mathfrak{Deslog}$ could reason about them. The computer used for the experiment has 16 physical processors, so we conducted the tests that used at most 36 threads to prove the algorithm's scalability. Table 4.1 lists the metrics of the test cases. The results are shown from Figure 4.6 to Figure 4.8.

4.3.2 Discussion

The data collected from testing both trivial and profound ontologies show a scalable performance improvement. The tests on relatively profound ontologies demonstrate better scalability than on trivial ones.

Because some trivial ontologies' single thread configuration computing time T_1 is rather short, normally shorter than ten seconds, the overhead introduced by maintaining multiple threads can limit the scalability. At the peak values of these trivial ontologies' tests, the reasoning times are reduced to several milliseconds, i.e. the whole work load assigned to a single thread is around several milliseconds in these settings. According to our empirical knowledge, such work load is significant enough with respect to the overhead which is produced by manipulating threads as well as accessing to shared data. Consequently, benefits gained from parallelized processing cannot counteract the system overhead, and the reasoning performance begins to declining.

When the algorithm was used to test ontologies of which sizes are generally large enough, the scalability is linear, sometimes super-linear.[3] These bigger ontologies need a longer single thread configuration computing time, T_1. The overhead introduced by maintaining a *tolerable* number of multiple threads is tiny and becomes insignificant. A tolerable number, N_i, should always be smaller than or equal to the

[3]A super-linear speedup is controversial but sometimes observed, and we accept it as the result of the combined effectiveness of hardware, software, and algorithms [7, 22, 36, 66, 69].

ontology	expressivity	concept count	axiom count
bfo	$\mathcal{ALC}$	36	45
pharmacogenomics_complex	$\mathcal{ALC}$	145	259
economy	$\mathcal{ALCH}(\mathcal{D})$	339	563
transportation	$\mathcal{ALCH}(\mathcal{D})$	445	489
mao	$\mathcal{ALE}+$	167	167
yeast_phenotype	$\mathcal{AL}$	281	276
loggerhead_nesting	$\mathcal{ALE}$	311	347
spider_anatomy	$\mathcal{ALE}$	454	607
pathway	$\mathcal{ALE}$	646	767
amphibian_anatomy	$\mathcal{ALE}+$	703	696
flybase_vocab	$\mathcal{ALE}+$	718	726
tick_anatomy	$\mathcal{ALE}+$	631	947
plant_trait	$\mathcal{ALE}$	976	1140
evoc	$\mathcal{AL}$	1001	990
protein	$\mathcal{ALE}+$	1055	1053

Table 4.1: Metrics of the test cases—$\mathcal{D}\mathfrak{eslog}$.

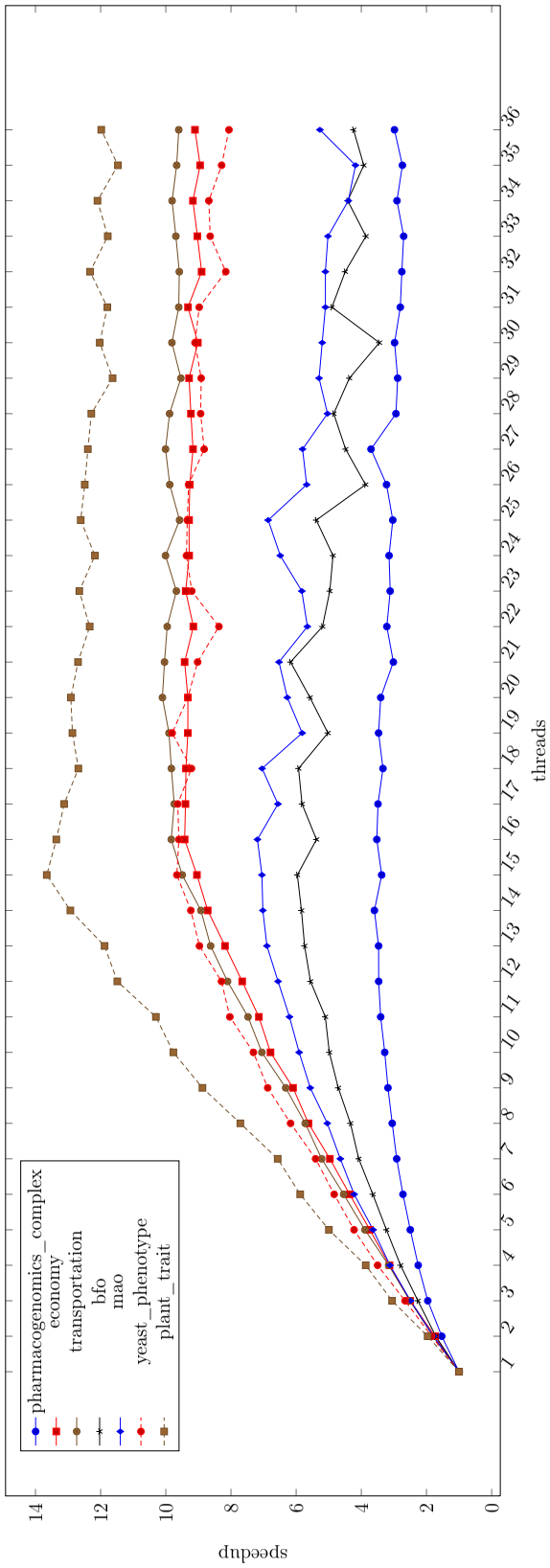


Figure 4.6: The gained scalability—pharmacogenomics_complex, economy, transportation, bfo, mao, yeast_phenotype, and plant_trait.

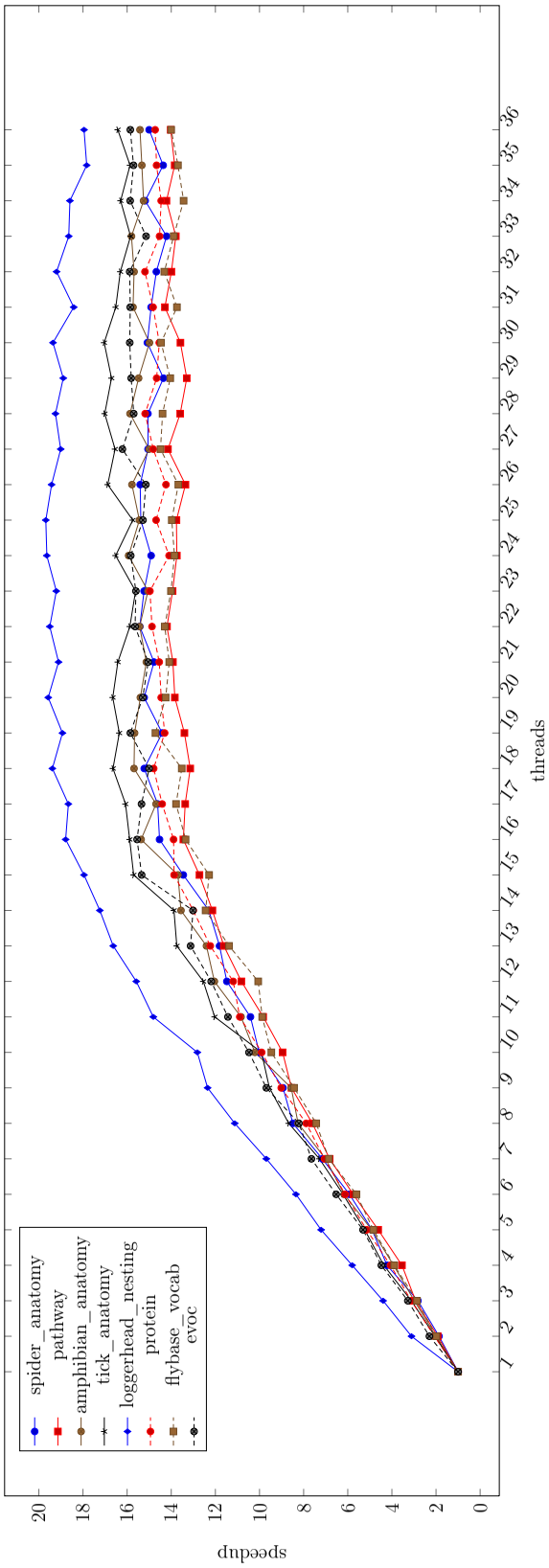


Figure 4.7: The gained scalability—spider_anatomy, pathway, amphibian_anatomy, tick_anatomy, loggerhead_nesting, protein, flybase_vocab, and evoc.

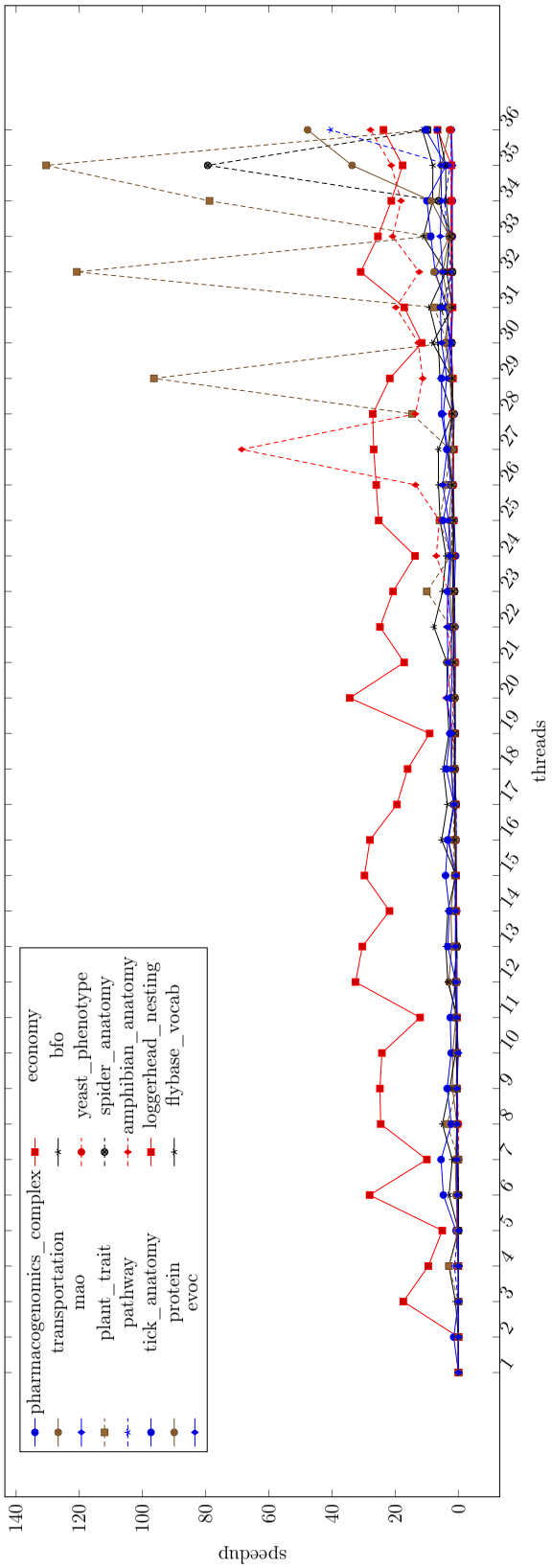


Figure 4.8: The standard deviations of threads' working time in the tests.

total number of the processors of a computer. Here, $N_i \in [1, 16]$.

In some cases, even though the number of threads exceeds 16, the reasoning performance keeps stable in a rather long run. Further scalability improvement may be achieved by adding processors.

Figure 4.8 demonstrates the standard deviation of single thread's working times in the series of test configurations (in the unit of millisecond). Overall, the deviations are limited to an acceptable range, i.e. at most less than 140 milliseconds which is relatively insignificant with respect to system overhead. This implies that work load is well balanced among threads. That is to say, all threads are as much busy as possible. For the most part, when the number of threads is less than the tolerable number, 16, the deviations are normally close to 0. When threads are added after 16, the deviations become greater. This is because some processor executes more than one thread, and hereby the thread context switching produces a lot of overhead. In our original implementation, we had distributed all subsumption candidates into independent lists, every of which mapped a thread, but the deviations were sometimes too large. So, the $\mathcal{D}\mathfrak{eslog}$ classification uses a shared queue to buffer all subsumption candidates, as it assures making all threads busy.

We had conducted similar experiments on a high-performance computing cluster, and the results are not good enough. The speedup gained on the cluster was generally less than three while we assigned at least 16 processors for each test. The most plausible explanation we can give is that the complex hardware and software environment of the cluster degrades the performance of $\mathcal{D}\mathfrak{eslog}$. The cluster consists of three types of computing nodes with respect to the built-in processors: 4-core, 8-core, and 16-core. And the same type of computers may have heterogeneous architectures. A job is scheduled on one or more computers randomly. It is normal to assign a job to more than one computers, and the communication between computers results in a bottleneck. Another possible reason is that the cluster does not assure exclusive usage, which means it is possible that more than one jobs are running on

the same computer at the same time. Therefore, it is necessary to conduct some basic concurrency benchmark tests before testing $\mathcal{D}$eslog.

We also had investigated the feasibility of accommodating some significant tableaux optimization techniques to concurrency settings, but not all of the optimization techniques can be easily adapted to concurrent versions.

4.4 Summary

The objective of this research is to explore how parallelism plays a role in tableau-based DL reasoning. A number of tableau-based DL reasoning optimization techniques have been extensively researched, but most of them are investigated in *sequential* contexts, so adapting these methods to the *parallel* context is an important part of this research.

We have partially shown that shared-memory parallel tableau-based DL reasoning can contribute to scalable solutions. We present our reasoner, $\mathcal{D}$eslog, of which the architecture is devised specially for shared-memory parallel environment. We present an aspect of the reasoner's concurrency performance, and a good scalability is gained for TBox classification. $\mathcal{D}$eslog is a vehicle for related investigations.

The advantage of the concurrent reasoning method proposed in this research is that the reasoning performance can be scaled in a near-linear way since its primary parallelized computing objects, subsumption tests, share few resources. Compared with canonical DL reasoning techniques, this method can fully make use of computing resources, and its performance may surpass canonical methods as long as enough number of processors are available, as is possible in many cases. And this is also the disadvantage of this method: Its performance depends on the computing resources available. Fortunately, computing resources, especially processor resources, are provided in a faster and cheaper pace.

Chapter 5

Parallelizing Tableau Conjunctive Branches

5.1 Introduction

It is well-known that an expansion tree is an *and-or* tree in tableau-based DL reasoning. Disjunctive branches compose the *or* part of a completion tree, conjunctive branches do the *and* part, and generally the two types of branches interlace with one another. Almost all of the few shared-memory parallelized tableau-based DL reasoning investigations focus on exploiting disjunctive branches in expansion trees.

5.2 The Role of Conjunctive Branches

Tableau-based algorithms have been the primary choice of DL reasoning for a long time. The core of a tableau algorithm is a set of rules that are used to construct completion trees. Whether a clash-free completion tree can be built determines the satisfiability of a problem domain. In DL languages with sufficient expressive power, such completion trees are regarded as *and-or* ones [71]. That is to say, both *conjunctive* and *disjunctive* branches exist in the completion trees.

A clash-free completion tree must have at least one disjunctive branch that contains no clashed conjunctive branches. For example, in a skeletal way, a typical tableau algorithm generates the following completion tree at some point when testing the satisfiability of the concept $(\exists r_1.C_1 \sqcap \exists r_2.C_2) \sqcup (C_3 \sqcap \neg C_3)$:

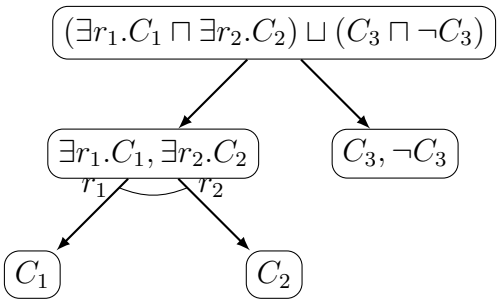


Figure 5.1: The tableau expansion tree of testing the satisfiability of $(\exists r_1.C_1 \sqcap \exists r_2.C_2) \sqcup (C_3 \sqcap \neg C_3)$.

The concept $(\exists r_1.C_1 \sqcap \exists r_2.C_2) \sqcup (C_3 \sqcap \neg C_3)$ must have the disjunctive branches $\exists r_1.C_1 \sqcap \exists r_2.C_2$ or $C_3 \sqcap \neg C_3$ clash-free only if it is satisfiable. In this case, $C_3 \sqcap \neg C_3 \equiv \bot$, so the satisfiability of $(\exists r_1.C_1 \sqcap \exists r_2.C_2) \sqcup (C_3 \sqcap \neg C_3)$ is determined by whether $\exists r_1.C_1 \sqcap \exists r_2.C_2$ is satisfiable. That is to say, if the concept $(\exists r_1.C_1 \sqcap \exists r_2.C_2) \sqcup (C_3 \sqcap \neg C_3)$ is satisfiable, both the conjunctive branches $\exists r_1.C_1$ and $\exists r_2.C_2$ must be clash-free. The algorithm has to explore all conjunctive branches unless an unsatisfiability result is entailed.

Testing satisfiability is an essential function in tableau-based DL reasoning, and its goal is to search for a model by expanding concepts descriptions to completion trees, which consist of disjunctive and conjunctive branches. Testing satisfiability is generally used by other DL reasoning services. As we know an important functionality of modern DL systems is classification, which calculates all subsumption relationships entailed by a terminology:

$$\forall C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}, \forall D^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} : \mathcal{T} \stackrel{?}{\models} C \sqsubseteq D \iff \mathcal{T} \stackrel{?}{\models} \neg C \sqcup D \tag{5.1}$$

With respect to $\mathcal{T}$, $C \sqsubseteq D$ is proven if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ holds for *every* model $\mathcal{I}$ of $\mathcal{T}$. This is calculated by testing the satisfiability of the concept $C \sqcap \neg D$ with respect to

$\mathcal{T}$. The subsumption computation is reducible to testing satisfiability in such a way. It is obvious that $C \not\equiv D$ is the most common case, and thus the majority of such subsumption tests find models. That is to say, in order to gain better performance, a tableau-based DL reasoning algorithm should find models as fast as possible. In a tableau expansion view, such a model is a disjunctive branch with a bundle of conjunctive branches, both clash-free. Considering that such a disjunctive branch exists quite often, faster processing of conjunctive branches in that disjunctive branch should improve reasoning performance. Although research on parallelizing the processing of disjunctive branches is known, parallelizing the processing of conjunctive branches has not been researched so far, but it should play a role in improving the performance of tableau-based DL reasoning.

5.3 Parallelism

As mentioned before, there exist a number of approaches that are being investigated to increase the performance of reasoners, and concurrent computing is an option. In tableau-based DL reasoning, disjunctive and conjunctive branches have always been sequentially processed as of now, although there exists the potential benefit of parallelization.

The search for a model in a disjunctive branch is independent of other disjunctive branches. The satisfiability of a concept is sufficiently supported by any model found among disjunctive branches. With a sequential algorithm, if two disjunctive branches are generated at some point, the second branch is only calculated if the first branch is not clash-free. With a parallel algorithm, multiple disjunctive branches are processed at the same time, and the search terminates when one of them is proven as clash-free. Some research has already been reported on the topic of parallelizing tableau calculation on disjunctive branches [16, 55, 58, 85].

Computation on a conjunctive branch impacts its siblings in a different way

than for disjunctive branches. A model is found by a tableau algorithm if and only if all involved conjunctive branch siblings are clash-free. With a sequential algorithm, all conjunctive branches on a disjunctive branch must be explored so that a clash-free *and*-tree can be built. Parallelizing computation on conjunctive branches in a satisfiable context should theoretically improve performance. Given the fact that most satisfiability tests introduced by classification, a key functionality of a DL reasoning system, return positive results, parallelizing conjunctive branches in tableau-based reasoning should play an important role. However, conjunctive branch parallelization has not been researched as much yet.

Parallelizing the processing of conjunctive branches is necessary to maximally utilize parallel computers. As we discussed in Section 5.2, the majority of computations of tableau-based DL reasoning find clash-free completion trees, each of which can be considered as a disjunctive branch containing a number of conjunctive branches. According to our experience, subsumption tests in classification are often easily satisfiable. Such a satisfiable disjunctive branch is usually the first one being tested. So, a parallel scheme in that case hardly improves reasoning performance. On the other hand, all conjunctive branches of a clash-free disjunctive branch must be explored and determined as clash-free. Therefore, parallelizing the exploration of potentially clash-free conjunctive branches can improve reasoning performance. Research on parallelizing the processing of conjunctive branches in tableau-based DL reasoning may even play a more important role than on disjunctive branches.

5.4 Algorithm Design and Implementation

In this section we present a parallel tableau-based DL algorithm. When we use the word *parallel* in the following, a modern shared-memory multi-thread architecture should always be taken into account.

Concurrent algorithms have much more technical features than sequential ones.

Some solutions require very tricky techniques. For example, in a sequential context, a DL tableau algorithm terminates the search in a disjunctive branch when a clash is found in a conjunctive branch. Such a termination problem needs more complex mechanics to solve in a parallel context. A termination test in a multiple threads context not only needs to check its own state but also the state of its siblings, i.e., it must monitor contradiction detection for all its siblings as the prerequisite for ensuring both *soundness* and *performance*.

The efficient managing of resources is an important trade-off in designing concurrent algorithms, especially in a shared-memory context. A common pitfall in developing shared-memory parallel algorithms consist of taking unlimited threading for granted, as usually happens in recursive algorithms. A compensation for this flaw is the use of shared data to control resources allocated to a parallel program. However, shared data as well as communication between threads always reduces a parallel program's performance.

Our scheme for controlling continuation resources is using a thread pool, which is normally configured with a fixed size. The members of the pool are reusable, which largely reduces system overhead. The most notable shared data consists of an increasing number of sibling conjunctive branches, and we use a concurrent queue to buffer them. Every threaded reasoning continuation picks a conjunctive branch out of the shared queue and processes it. Also, every continuation has to monitor and report its finding, as mentioned before.

The parallelization of processing conjunctive branches is addressed by Algorithms 1, 2, and 3. It consists of two parts: Algorithm 1 as well as Algorithm 2 as the control (master), and Algorithm 3 as the continuation (slave). Algorithm 1 first applies tableau expansion rules that are neither a $\exists$-rule nor a $\forall$-rule. The function *clashed*? returns *true* if *all* disjunctive branches (i.e., *stages* in $\mathcal{D}\mathfrak{eslog}$ [85]) are not clash-free, otherwise it returns *false* and cuts away clashed stages. If all disjunctive branches are not clash-free in this phase, the computation terminates. Otherwise, the model-

search is continued on the generated disjunctive branches, which are provisionally clash-free. That is to say, the generating rule produces conjunctive branches which are kept in a buffer. Then the aforementioned thread pool schedules computation continuations on the conjunctive branch buffer. The computation continuation executed by the pooled thread is addressed by Algorithm 3, which is executed by multiple threads simultaneously.

We implemented Algorithms 1, 2, and 3 with our parallelized tableau-based DL reasoning framework $\mathcal{D}\mathfrak{eslog}$ [85]. The underground parallel mechanics of $\mathcal{D}\mathfrak{eslog}$ is supported by the *java.concurrent* package. All working threads processing continuations are activated and pooled in the bootstrap phase. In each subsumption test run, every thread monitors a *volatile* flag that indicates whether a clash has been detected by its siblings and modifies the flag if it finds a clash (Line 2 and 3, Algorithm 3). If a clash has been detected, all threads and the flag are reset.

A performance bottleneck may result from the low level Java concurrency components. For example, we use a concurrent linked queue to buffer immediate conjunctive branches, and the buffer is accessed by a number of threads concurrently. Also, we use *volatile* flags as shared states with the intention of notifying state modification as fast as possible, and the maintenance of the *volatile* variables may require extra processor resources in a shared-memory parallel computing environment. We can design and construct the high level part of the program, but can hardly control the low level facilities on which the program depends.

5.5 Experiments

Algorithm 1 is expected to improve the performance of tableau-based DL reasoning in such a way that conjunctive branches are processed simultaneously. A higher performance improvement is expected from reasoning about problems where more conjunctive expansion branches are involved. We designed a series of synthesized

Algorithm 1: *parallelize-traces*(*tree*, *rule-queue-without-*$\exists\forall$, *worker-queue*)

input :
tree: a tableau expansion tree.
rule-queue-without-$\exists\forall$: an ordered set containing tableau expansion rules except for $\exists$-rule and $\forall$-rule.
worker-queue: the pool keeping threads.

output:
some-trace-clashed?: true if clash is found, otherwise false.

```
1  begin
2      reload(rule-queue-without-∃∀);
3      rule ← dequeue(rule-queue-without-∃∀);
4      while rule ≠ ∅ ∧ ¬clashed?(tree) do
5          applicable? ← apply(rule, tree);
6          if applicable? then
7              reload(rule-queue-without-∃∀);
8          end if
9          rule ← dequeue(rule-queue-without-∃∀);
10     end while
11     /* some-trace-clashed? is a global variable. */
12     some-trace-clashed? ← clashed?(tree);
13     if ¬some-trace-clashed? then
14         foreach disjunctive-branch ∈ tree do
15             trace-queue ← generate-trace-queue(disjunctive-branch, rule-∃);
16             some-trace-clashed? ← false;
17             if ¬empty?(trace-queue) then
18                 continue? ←true ; /* continue? is a global variable. */
19                 while continue? do
20                     trace ← dequeue(trace-queue);
21                     process-trace(trace, worker-queue);
22                 end while
23                 if ¬some-trace-clashed? then
24                     break;
25                 end if
26             end if
27         end foreach
28     end if
29     return some-trace-clashed?;
30 end
```

Algorithm 2: *process-trace*$(trace, worker\text{-}queue)$

input :
trace: a tableau conjunctive branch.
worker-queue: the pool keeping threads.
output:
some-trace-clashed?: true if clash is found, otherwise false.

```
1  begin
2      if trace ≠ ∅ then
3          worker ← ∅;
4          while worker = ∅ ∧ ¬some-trace-clashed? do
5              worker ← get-idle-worker(worker-queue);
6          end while
7          if some-trace-clashed? then
8              continue? ←false ; /* continue? is a global variable. */
9          else
10             do-job(worker, λ(trace, some-trace-clashed?));
11         end if
12     else
13         if some-trace-clashed? ∨ get-busy-worker(worker-queue) = ∅ then
14             continue? ←false;
15         end if
16     end if
17 end
```

Algorithm 3: $\lambda(trace, *clashed\text{-}flag?*)$

input :
trace: a tableau conjunctive branch.
$*clashed\text{-}flag?*$: a pointer argument indicating whether a clash exists: true if clash is found; otherwise false.

```
1  begin
2      apply-tableau-rules(trace);
3      *clashed-flag?* ← ¬clash-free?(trace);
4  end
```

tests to prove this assumption.

The test cases consist of a set of OWL benchmarks developed on the basis of the *tea* ontology.[1] Their size (and complexity) can be scaled by a GCI axiom pattern as follows:

$$\bigsqcup_{i=0}^{\jmath} \forall R_{2i+1}.(C_{2i+1} \sqcap C_{2i+2}) \sqsubseteq_{i=0}^{\jmath} \exists R_{2i}.(C_{2i} \sqcup C_{2i+1}),$$
$$C_i^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}, R_i^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}, i \in \mathbb{N}, \jmath \in \mathbb{N}, i \leq \jmath \quad (5.2)$$

We defined a set of factors to measure the algorithm's performance. Performance improvement is directly reflected by *thread number* and *speedup*. With the same thread number, reasoning performance varies with the number of involved conjunctive branches. So, our program records the number of involved conjunctive branches, μ, in every satisfiability test. N_μ, the total number of the tests in a set of computations, of which every one processes μ conjunctive branches, is calculated after each run. We discovered that the most frequently occurring number of the involved conjunctive branches impacts the final reasoning performance and is the *order* of the conjunctive branches involved in a run. We take the *mean* of the values to calculate this order, which is noted as τ, where at most k conjunctive branches are involved in a satisfiability test:

$$\tau = \frac{\sum_{\mu=0}^{k} N_\mu \times \mu}{\sum_{\mu=0}^{k} N_\mu}, \ k \in \mathbb{N}, \ 0 \leq \tau < +\infty \quad (5.3)$$

For example, we get $\tau = \frac{5\times0+8\times1+13\times2+21\times3}{5+8+13+21} = \frac{97}{47}$, with respect to the sample data shown in Table 5.1.

We conducted several experiments to evaluate Algorithm 1. According to our

[1] http://code.google.com/p/deslog/downloads/detail?name=tea.tar.gz

μ	0	1	2	3
N_μ	5	8	13	21

Table 5.1: A sample data set for Equation 5.3 with $k = 3$.

knowledge, the hardware environment can have quite an impact on the performance of a parallel program in a shared-memory context [55, 85]. In this case, a 16-core computer running Solaris OS and 64-bit Sun Java 6 was employed to test the program. The 16 processors are manufactured on 2 integrated circuits, each having 8 processors. At most 64G physical memory is accessible by the JVM. With various combinations of the number of processing threads and problem sizes, Algorithm 1 demonstrated the capability of being scaled.

The reasoning performance of Algorithm 1 is illustrated by the results shown in Figures 5.2-5.8. When $\tau = 2.09$, i.e., each test processes only $\sim$2 conjunctive branches, parallelizing the processing of conjunctive expansion does not contribute to a performance improvement. It seems that the overhead introduced by threading outclasses the benefits. However, according to our experiments performance improvements can be gained when $\tau \geq 3.09$. Better performance improvements come from greater τ values.

The scalability of parallelizing the processing of conjunctive branches is summarized in Figure 5.9 by illustrating the speedup trend, which is based on the median speedup values from our 9-thread tests (Figure 5.2-5.8), given the observed τ values.

Besides synthesized test cases, we also tested a real-world ontology, *fly_anatomy*. Figure 5.10 shows the result for *fly_anatomy*. The maximum speedup value is around the value of τ, which is in this test case 2.91, and we see that the peak value of the speedup is greater than 2 and that there exists a linear speedup increase before reaching the peak value. This test result shows a stable scalability to some degree.

We amend Equation 5.3 to a more general form as indicated by Equation 5.4, in order to illustrate the program's impacts on real-world ontologies:

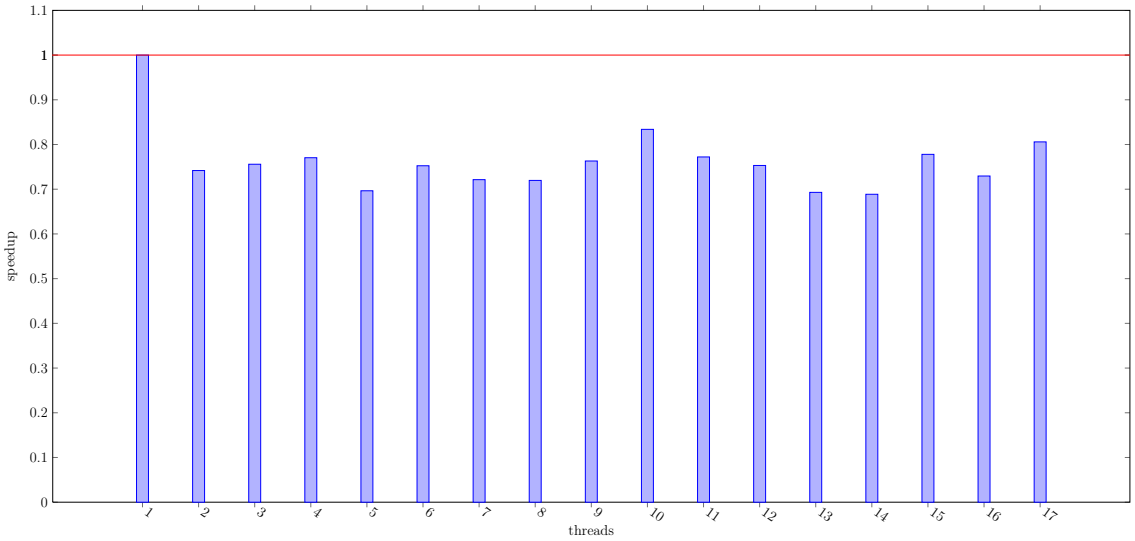


Figure 5.2: The speedup when $\jmath = 2$ and $\tau = 2.09$.

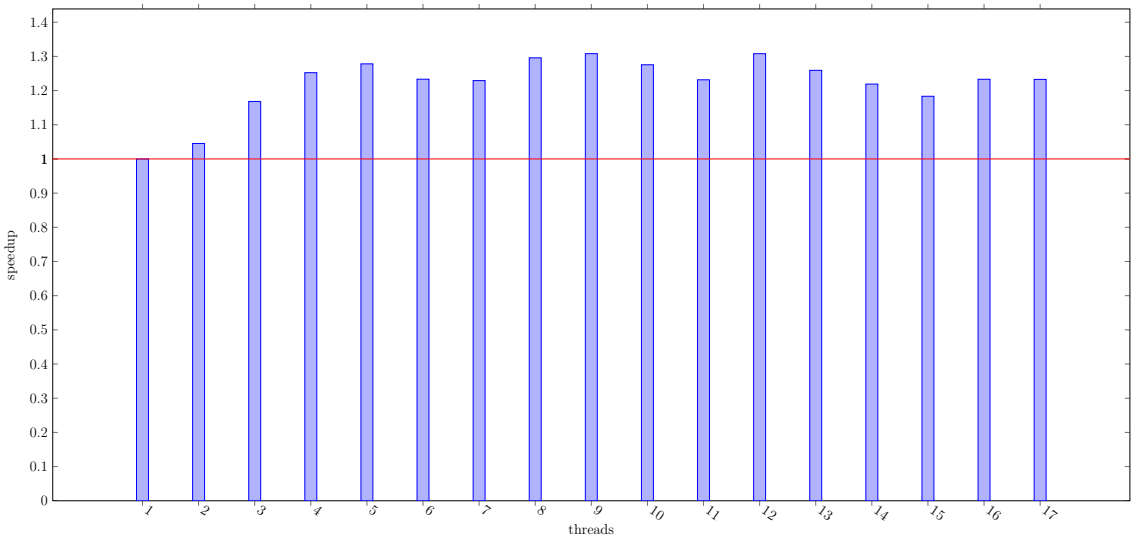


Figure 5.3: The speedup when $\jmath = 3$ and $\tau = 3.09$.

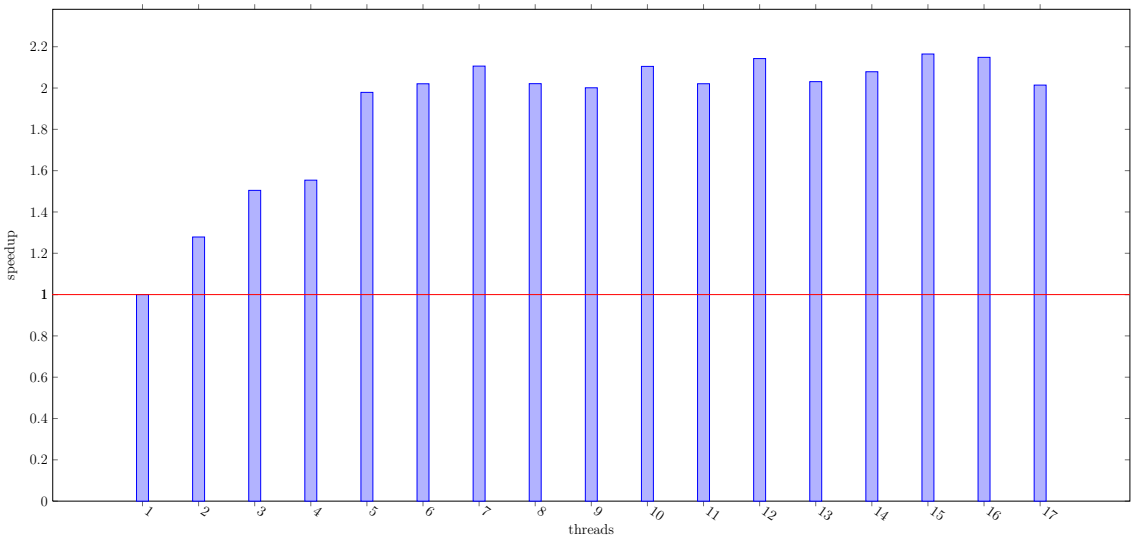


Figure 5.4: The speedup when $\jmath = 4$ and $\tau = 5.08$.

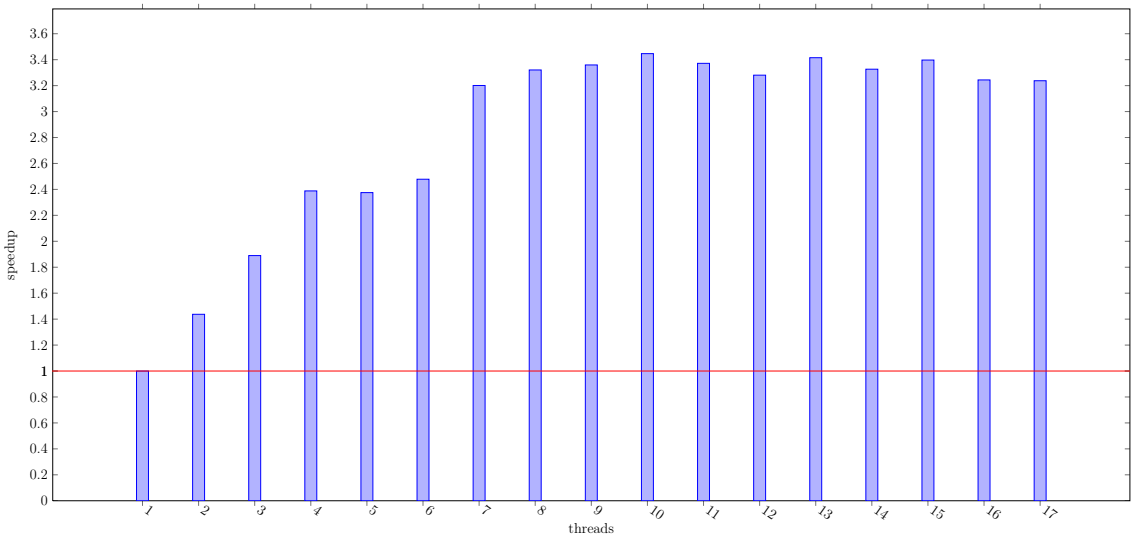


Figure 5.5: The speedup when $\jmath = 7$ and $\tau = 7.13$.

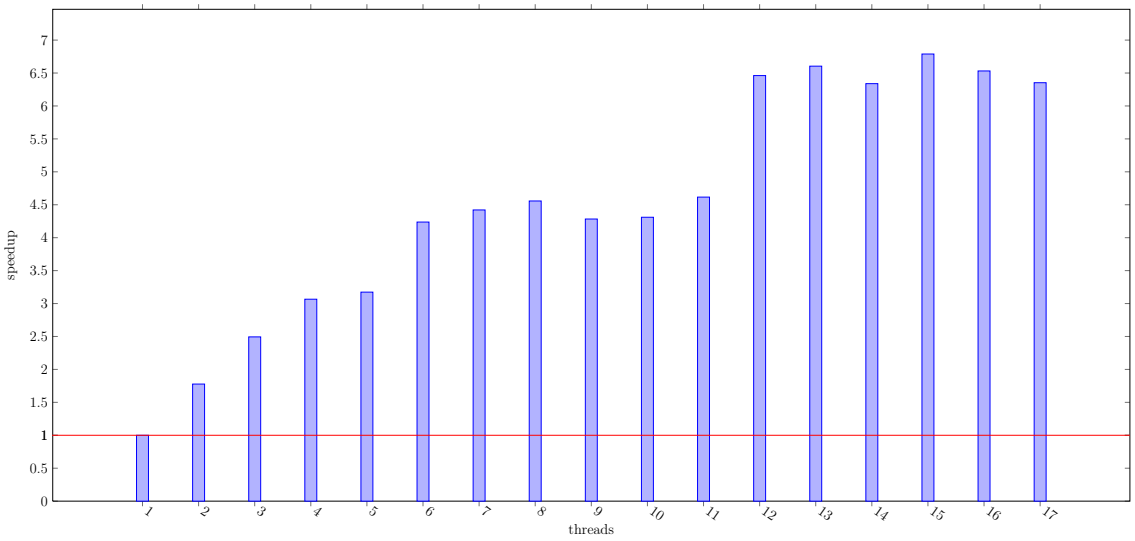


Figure 5.6: The speedup when $\jmath = 11$ and $\tau = 12.11$.

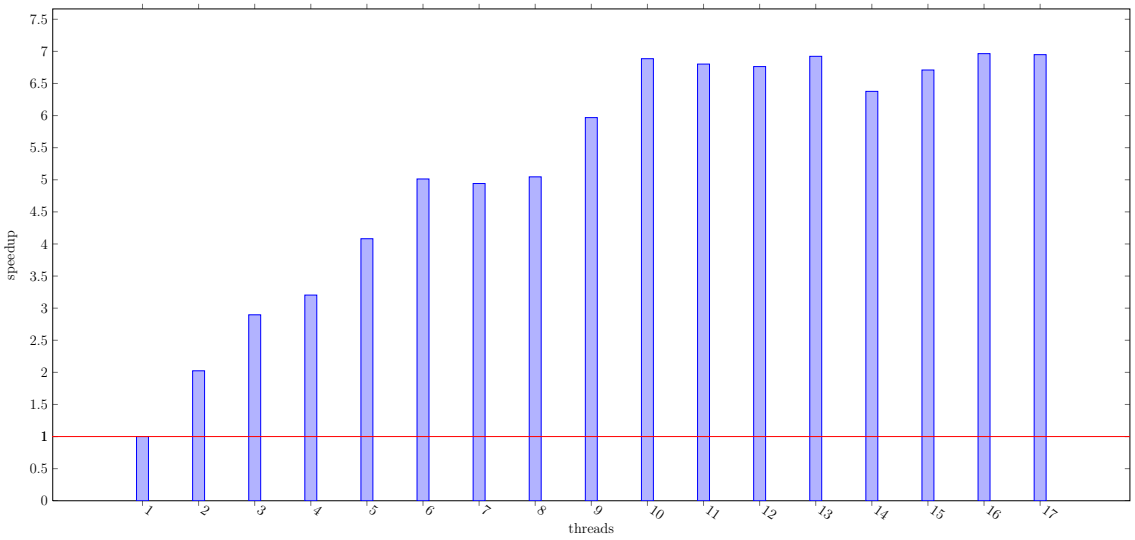


Figure 5.7: The speedup when $\jmath = 17$ and $\tau = 18.08$.

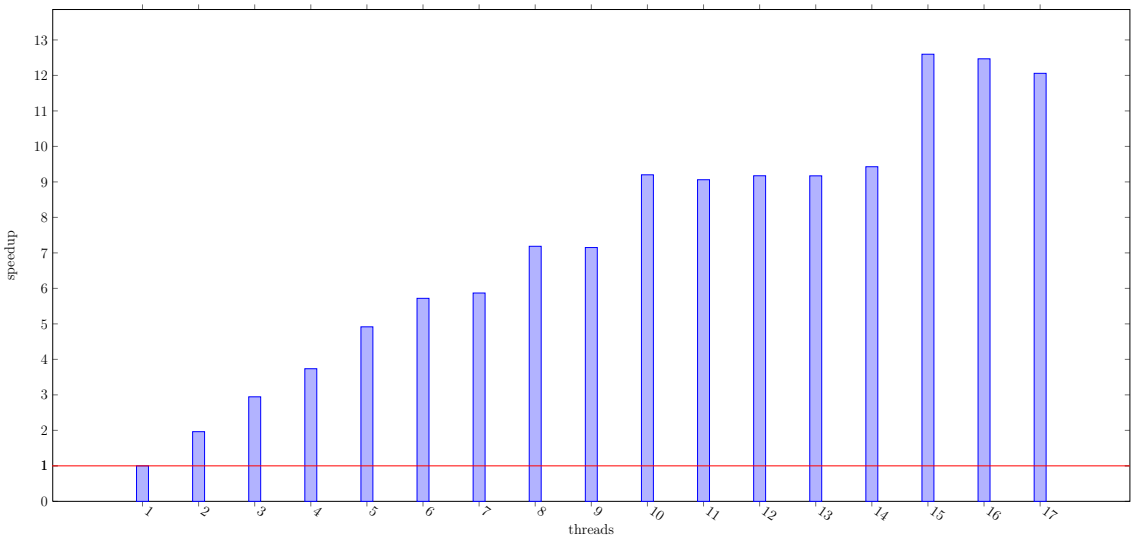


Figure 5.8: The speedup when $\jmath = 28$ and $\tau = 29.06$.

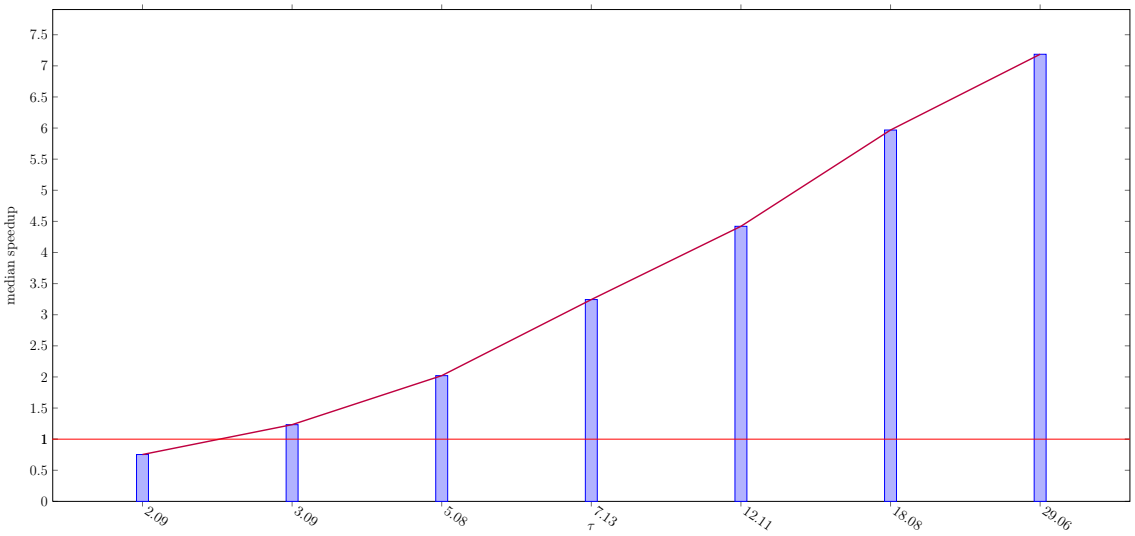


Figure 5.9: The median speedup trend of the variety of τ values.

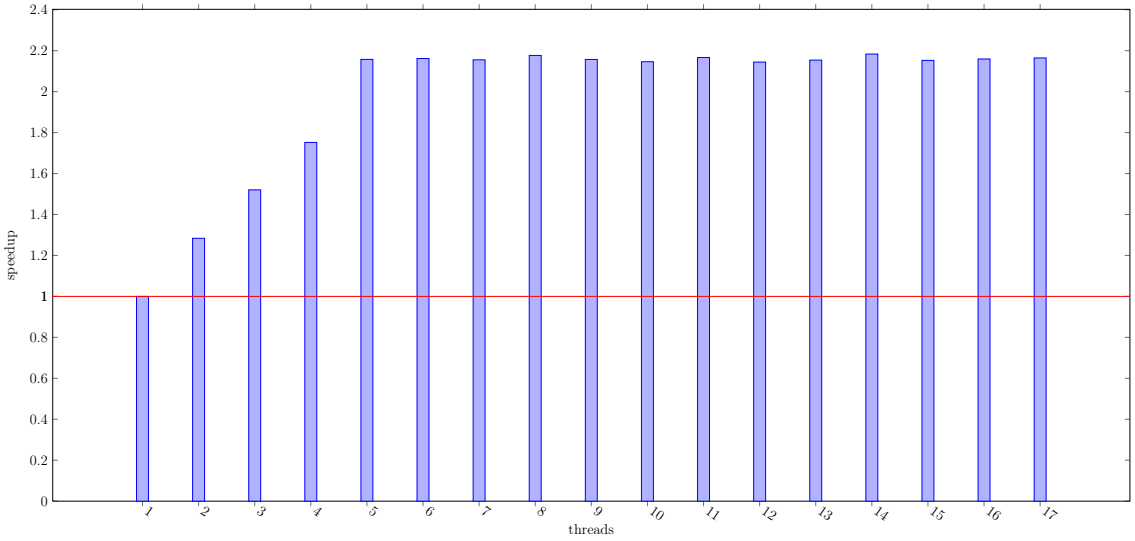


Figure 5.10: Test ontology fly_anatomy, $\tau = 2.91$.

$$\tau_q = \frac{\sum_{\mu=q}^{k} N_\mu \times \mu}{\sum_{\mu=q}^{k} N_\mu},\ 0 \leq q \leq k,\ k \in \mathbb{N},\ 0 \leq \tau < +\infty \tag{5.4}$$

Here, q is a lower bound of a sample space. Namely, tests with conjunctive branches less than q are bypassed. With Equation 5.4, we can focus on the tests with greater q value, e.g. $q = 3$. Figure 5.11 shows the speedups gained by testing some ontologies, with $q = 4$.

Scalability is the most interesting point in this research. Optimistically, we expect to gain linear or even super-linear scalability. In the circumstances of expecting minimal overheads, we anticipate the ratio, between speed-up and the number of thread, $e \geq 1$ in accordance with $e = \frac{s}{n},\ n \leq \tau,\ n \leq p$, where s is speedup, n is thread number, and p is the total number of processors. However, $e \leq 1$ is the most normal case in practice. According to our tests, $e = 0.8398$ is the greatest value, which occurs when $s = 12.5971$, $n = 15$, $\tau = 29.06$, and $p = 16$ (see Figure 5.8). It is obvious that a certain system overhead cannot be avoided and must hinder the program from reaching the normal peak value. Furthermore, most tableau-based satisfiability tests

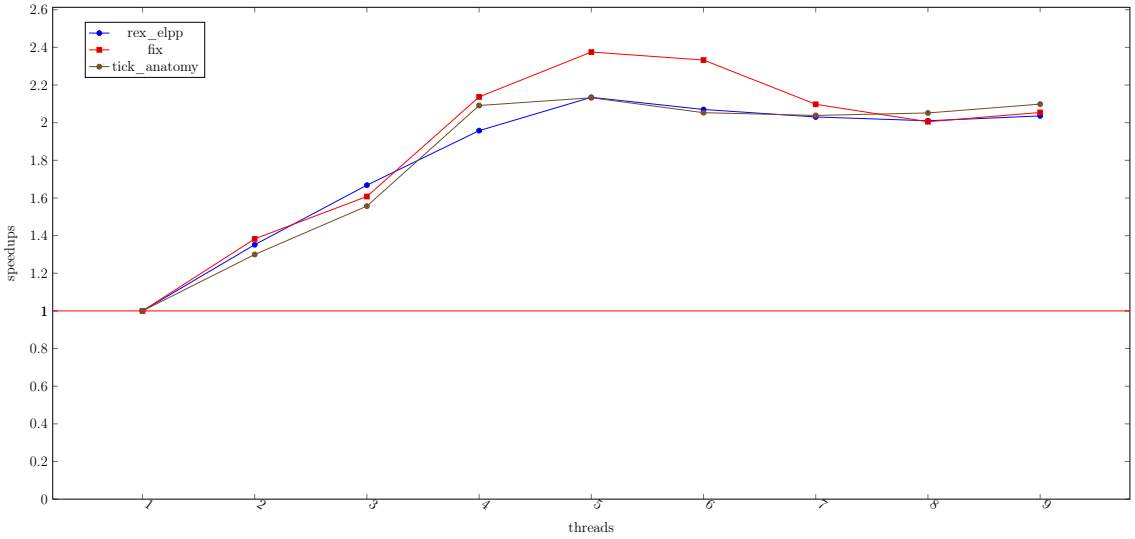


Figure 5.11: Test ontologies rex_elpp, fix, and tick_anatomy with $\tau_4 = 4.73$, 5.00, and 4.22 respectively.

for classification find models, as mentioned above, and parallelizing the processing of conjunctive branches is useful in the case of satisfiable tests. However, negative tests entailing unsatisfiability predominantly exist in classification, too. If a number of clashes can be detected before processing a bundle of conjunctive branches, this parallel algorithm can hardly contribute to a performance gain.

5.6 Summary

The objective of this work is to improve the processing performance of DL tableau algorithms by utilizing cheap multiprocessor computing resources, which are ubiquitous now. A possible solution is the integration of concurrent computing, or more specifically, concurrent reasoning, which should make full use of the availability of multiprocessor computing resources and may improve performance in a scalable way. Our research proved that such scalability is possible.

We have shown that the computing performance of tableau-based DL reasoning can be improved by parallelizing the processing of conjunctive branches of expansion

trees. All of the investigations that explore parallelized tableau-based DL reasoning make an effort to exploit simultaneous processing of disjunctive branches in tableau expansion trees. On the other hand, our research is the first one to seriously investigate the parallel processing of conjunctive branches in tableau expansion trees. We addressed the role of conjunctive branches in tableau expansion trees and noticeable points of parallelizing the processing of conjunctive branches. We presented a parallel algorithm that simultaneously calculates conjunctive branches. We discussed the key characteristics of implementing the algorithm. We evaluated the program, and the essential effectiveness of the algorithm was shown by synthesized tests. We analyzed the scalability of the algorithm based on our proposed τ metric.

The advantage of this work is that scalability of reasoning about some ontologies, which produce a number of tableau conjunctive branches, can be gained by increasing the number of processing resources, and thus computing resources can be fully utilized. The disadvantage is that this method requires processing *shared* data. High-performance concurrent data structures and algorithms are necessary to manage the shared data, in order for achieving a better reasoning performance. This results in a rather complex implementation.

Chapter 6

Merge Classification

6.1 Introduction

One of the major obstacles that needs to be addressed in the design of corresponding algorithms and architectures is the overhead introduced by concurrent computing and its impact on scalability.

Heavily shared data as well as related communication cost always indicates an inefficient performance in parallel environments. Canonical DL reasoning algorithms, which form the basis of OWL reasoning, deal with a problem domain as a whole, which generally produces monolithic data and makes it hard to parallelize employed algorithms. In order to achieve effective parallelized DL reasoning novel methods need to be developed that process data as independently as possible.

Traditional *divide and conquer* algorithms split problems into independent sub-problems before solving them under the premise that not much communication among the divisions is needed when independently solving the sub-problems, so shared data is excluded to a great extent. Therefore, divide and conquer algorithms are in principle suitable for concurrent computing, including shared-memory parallelization and non-shared-memory distributed systems.

Furthermore, recently research on *ontology partitioning* has been proposed and

investigated for dealing with monolithic ontologies. Some research results, e.g. ontology modularization [30], can be used for decreasing the scale of an ontology-reasoning problem. Then, reasoning over a set of sub-ontologies can be executed in parallel. However, there is still a solution needed to reassemble sub-ontologies together. The algorithms presented in this research can also serve as a solution for this problem.

6.2 A Parallelized Merge Classification Algorithm

In this section, we present an algorithm for classifying DL ontologies. Part of the algorithm is based on standard top- and bottom-search techniques to incrementally construct the classification hierarchy (e.g., see [12]). Due to the symmetry between *top-down* ($\top\_search$) and *bottom-up* ($\bot\_search$) search, we only present the first one. In the pseudocode, we use the following notational conventions: Δ_i, Δ_α, and Δ_β designate sub-domains that are divided from Δ; we consider a subsumption hierarchy as a partial order over Δ, denoted as $\leq$, a subsumption relationship where C is subsumed by D ($C \sqsubseteq D$) is expressed by $C \leq D$ or by $\langle C, D\rangle \in \leq$, and $\leq_i$, $\leq_\alpha$, and $\leq_\beta$ are subsumption hierarchies over Δ_i, Δ_α, and Δ_β, respectively; in a subsumption hierarchy over Δ, $C \prec D$ designates $C \sqsubseteq D$ and there does not exist a named concept E such that $C \leq E$ and $E \leq D$; $\prec_i$, $\prec_\alpha$ and $\prec_\beta$ are similar notations defined over Δ_i, Δ_α, and Δ_β, respectively.

Our merge-classification algorithm classifies a taxonomy by calculating its divided sub-domains and then by merging the classified sub-taxonomies together. The algorithm makes use of two facts: (i) If it holds that $B \leq A$, then the subsumption relationships between B's descendants and A's ancestors are determined; (ii) if it is known that $B \not\leq A$, the subsumption relationships between B's descendants and A's ancestors are undetermined. The canonical DL classification algorithm, top-search & bottom-search, is modified and integrated into the merge-classification. The algorithm consists of two stages: divide and conquering, and combining. Algorithm

Algorithm 4: $\kappa(\Delta_i)$

input : The sub-domain Δ_i
output: The subsumption hierarchy classified over Δ_i

```
1  begin
2      if divided_enough?(Δ_i) then
3          return classify(Δ_i);
4      else
5          ⟨Δ_α, Δ_β⟩ ← divide(Δ_i);
6          ≤_α ← spawn κ(Δ_α);
7          ≤_β ← κ(Δ_β);
8          sync;
9          return μ(≤_α, ≤_β);
10     end if
11 end
```

4 shows the main part of our parallelized DL classification procedure. The keyword *spawn* indicates that its following calculation must be executed in parallel, either creating a new thread in a shared-memory context or generating a new process or session in a non-shared-memory context. The keyword *sync* always follows *spawn* and suspends the current calculation procedure until all calculations invoked by *spawn* have returned.

The domain Δ is divided into smaller partitions in the first stage. Then, classification computations are executed over each sub-domain Δ_i. A classified sub-terminology $\leq_i$ is inferred over Δ_i. The procedure *classify* is used by Algorithm 4 and is a general reasoning function that calls Algorithm 5. It is not shown in this research. This divide and conquering operations can progress in parallel.

Classified sub-terminologies are to be merged in the combining stage. Told subsumption relationships are utilized in the merging process. Algorithm 5 outlines the master procedure, and the slave procedure is addressed by Algorithms 6, 7, 8, and 9.

Algorithm 5: $\mu(\leq_\alpha, \leq_\beta)$

input : The master subsumption hierarchy $\leq_\alpha$
The subsumption hierarchy $\leq_\beta$ to be merged into $\leq_\alpha$
output: The subsumption hierarchy resulting from merging $\leq_\alpha$ over $\leq_\beta$

```
1 begin
2 |  ⊤α ← select-top(≤α);
3 |  ⊤β ← select-top(≤β);
4 |  ⊥α ← select-bottom(≤α);
5 |  ⊥β ← select-bottom(≤β);
6 |  ≤α← ⊤_merge(⊤α, ⊤β, ≤α, ≤β);
7 |  ≤i← ⊥_merge(⊥α, ⊥β, ≤α, ≤β);
8 |  return ≤i;
9 end
```

6.2.1 Divide and Conquer Phase

The first task is to divide the universe, Δ, into sub-domains. Without loss of generality, Δ only focuses on *significant* concepts, i.e., concept names or atomic concepts, that are normally declared explicitly in some ontology $\mathcal{O}$, and *intermediate* concepts, i.e., non-significant ones, only play a role in subsumption tests. Each sub-domain is classified independently. The *divide* operation can be naively implemented as an even partitioning over Δ, or by more sophisticated clustering techniques such as *heuristic partitioning* that may result in a better performance, as presented in Section 6.4. The conquering operation can be any standard DL classification method. We first present the most popular classification methods, top-search (Algorithm 6) (its duality, bottom-search, is omitted here).

The DL classification procedure determines the most specific super- and the most general sub-concepts of each significant concept in Δ. The classified concept hierarchy is a partial order, $\leq$, over Δ. $\top\_search$ recursively calculates a concept's *intermediate* predecessors, i.e., intermediate immediate ancestors, as a relation $\prec_i$ over $\leq_i$.

Algorithm 6: $\top\_search(C, D, \leq_i)$

```
input  : C: the new concept to be classified
         D: the current concept with ⟨D, ⊤⟩ ∈ ≤_i
         ≤_i: the subsumption hierarchy
output: The set of parents of C: {p | ⟨C, p⟩ ∈ ≤_i}.
1  begin
2      mark_visited(D);
3      green ← ∅;
4      forall the d ∈ {d | ⟨d, D⟩ ∈≺_i} do /* collect all children of D that subsume
       C */
5          if ≤?(C, d) then
6              green ← green ∪ {d};
7          end if
8      end forall
9      box ← ∅;
10     if green = ∅ then
11         box ← {D};
12     else
13         forall the g ∈ green do
14             if ¬marked_visited?(g) then
15                 box ← box ∪ ⊤_search(C, g, ≤_i) ; /* recursively test whether C
                   is subsumed by the descendants of g */
16             end if
17         end forall
18     end if
19     return box; /* return the parents of C */
20 end
```

6.2.2 Combining Phase

The independently classified sub-terminologies must be merged together in the combining phase. The original top-search (Algorithm 6) (and bottom-search) have been modified to merge two sub-terminologies $\leq_\alpha$ and $\leq_\beta$. The basic idea is to iterate over Δ_β and to use top-search (and bottom-search) to insert each element of Δ_β into $\leq_\alpha$, as shown in Algorithm 7.

However, this method does not make use of so-called told subsumption (and non-subsumption) information contained in the merged sub-terminology $\leq_\beta$. For

Algorithm 7: $\top\_merge^{-}(A, B, \leq_\alpha, \leq_\beta)$

input : A: the current concept of the master subsumption hierarchy, i.e. $\langle A, \top\rangle \in \leq_\alpha$
B: the new concept from the merged subsumption hierarchy, i.e. $\langle B, \top\rangle \in \leq_\beta$
$\leq_\alpha$: the master subsumption hierarchy
$\leq_\beta$: the subsumption hierarchy to be merged into $\leq_\alpha$
output: The merged subsumption hierarchy $\leq_\alpha$ over $\leq_\beta$.

```
1  begin
2  |  parents ← ⊤_search(B, A, ≤_α);
3  |  forall the a ∈ parents do
4  |  |  ≤_α ← ≤_α ∪ ⟨B, a⟩; /* insert B into ≤_α */
5  |  |  forall the b ∈ {b | ⟨b, B⟩ ∈ ≺_β} do /* insert children of B (in ≤_β) below
   |  |  parents of B (in ≤_α) */
6  |  |  |  ≤_α ← ⊤_merge^-(a, b, ≤_α, ≤_β);
7  |  |  end forall
8  |  end forall
9  |  return ≤_α;
10 end
```

example, it is unnecessary to test $\leq?(B_2, A_1)$ with sophisticated reasoning algorithms when we know $B_2 \leq B_1$ and $B_1 \leq A_1$, given that A_1 occurs in Δ_α and B_1, B_2 occur in Δ_β.

Therefore, we designed a novel algorithm in order to utilize the properties addressed by Propositions 1 to 8. The calculation starts with top-merge (Algorithm 8), which uses a modified top-search algorithm (Algorithm 9). This pair of procedures finds the most specific subsumers in the master sub-terminology $\leq_\alpha$ for every concept from the sub-terminology $\leq_\beta$ that is being merged into $\leq_\alpha$.

Proposition 1 *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $\langle B, A\rangle \in \prec_i$ is found in top-search, $\langle A, \top\rangle \in \leq_\alpha$ and $\langle B, \top\rangle \in \leq_\beta$, then for $\forall b_j \in \{b \mid \langle b, B\rangle \in \leq_\beta\}$ and $\forall a_k \in \{a \mid \langle A, a\rangle \in \leq_\alpha\} \cup \{A\}$ it follows that $b_j \leq a_k$.*

Figure 6.1 shows the case, where $\{a_1, \ldots, a_m\}$ is the set of parents of A and $\{b_1, \ldots, b_n\}$ the set of children of B. It is easy to see that $b_j \leq a_k$ due to the

Algorithm 8: $\top\_merge(A, B, \leq_\alpha, \leq_\beta)$

input : A: the current concept of the master subsumption hierarchy, i.e. $\langle A, \top\rangle \in \leq_\alpha$
B: the new concept of the merged subsumption hierarchy, i.e. $\langle B, \top\rangle \in \leq_\beta$
$\leq_\alpha$: the master subsumption hierarchy
$\leq_\beta$: the subsumption hierarchy to be merged into $\leq_\alpha$
output: the merged subsumption hierarchy $\leq_\alpha$ over $\leq_\beta$

```
1  begin
2      parents ← ⊤_search*(B, A, ≤_β, ≤_α);
3      forall the a ∈ parents do
4          ≤_α ← ≤_α ∪ ⟨B, a⟩;
5          forall the b ∈ {b | ⟨b, B⟩ ∈ ≺_β} do
6              ≤_α ← ⊤_merge(a, b, ≤_α, ≤_β);
7          end forall
8      end forall
9      return ≤_α;
10 end
```

transitivity of the subsumption relationship. From our premise we know that $b_j \leq B$, $B \leq A$ and $A \leq a_k$, therefore it holds that $b_j \leq a_k$ for all j, k. ■

Proposition 2 *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $\langle B, A\rangle \in \prec_i$ is found in top-search, $\langle A, \top\rangle \in \leq_\alpha$ and $\langle B, \top\rangle \in \leq_\beta$, then for $\forall b_j \in \{b \mid \langle b, B\rangle \in \prec_\beta \wedge b \neq B\}$ and $\forall a_k \in \{a \mid \langle a, A\rangle \in \prec_\alpha \wedge a_j \neq A\}$ it is still necessary to calculate whether $b_j \leq a_k$.*

Figure 6.2 shows the case, where $\{a_1, \ldots, a_m\} = \{a \mid \langle a, A\rangle \in \prec_\alpha \wedge a \neq A\}$ and $\{b_1, \ldots, b_n\} = \{b \mid \langle b, B\rangle \in \prec_\beta \wedge b \neq B\}$. We know that $B^\mathcal{I} \subseteq A^\mathcal{I}$ or $B^\mathcal{I} \cap (\neg A)^\mathcal{I} = \emptyset$ and $b_j^\mathcal{I} \subseteq B^\mathcal{I}$ leads to $b_j^\mathcal{I} \cap (\neg A)^\mathcal{I} = \emptyset$ but since $(\neg a_k)^\mathcal{I} \supseteq (\neg A)^\mathcal{I}$ it is unknown for all j, k whether $b_j^\mathcal{I} \cap (\neg a_k)^\mathcal{I}$ is always empty or always not empty. ■

Proposition 3 *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $B \not\leq A$ is found in top-search, $\langle A, \top\rangle \in \leq_\alpha$ and $\langle B, \top\rangle \in \leq_\beta$, then for $\forall b_j \in \{b \mid \langle b, B\rangle \in \leq_\beta \wedge b \neq B\}$ and $\forall a_k \in \{a \mid \langle a, A\rangle \in \leq_\alpha\} \cup \{A\}$ it is necessary to calculate whether $b_j \leq a_k$.*

Algorithm 9: $\top\_search^*(C, D, \leq_\beta, \leq_\alpha)$

input : C: the new concept to be inserted into $\leq_\alpha$, and $\langle C, \top\rangle \in \leq_\beta$
D: the current concept, and $\langle D, \top\rangle \in \leq_\alpha$
$\leq_\beta$: the subsumption hierarchy to be merged into $\leq_\alpha$
$\leq_\alpha$: the master subsumption hierarchy
output: The set of parents of C: $\{p \mid \langle C, p\rangle \in \leq_\alpha\}$

```
1  begin
2      mark_visited(D);
3      green ← ∅; /* subsumers of C that are from ≤_α */
4      red ← ∅; /* non-subsumers of C that are children of D */
5      forall the d ∈ {d | ⟨d, D⟩ ∈≺_α ∧ ⟨d, ⊤⟩ ∉ ≤_β} do
6          if ≤?(C, d) then
7              green ← green ∪ {d};
8          else
9              red ← red ∪ {d};
10         end if
11     end forall
12     box ← ∅;
13     if green = ∅ then
14         if ≤?(C, D) then
15             box ← {D};
16         else
17             red ← {D};
18         end if
19     else
20         forall the g ∈ green do
21             if ¬marked_visited?(g) then
22                 box ← box ∪ ⊤_search*(C, g, ≤_β, ≤_α);
23             end if
24         end forall
25     end if
26     forall the r ∈ red do
27         forall the c ∈ {c | ⟨c, C⟩ ∈≺_i} do
28             ≤_α← ⊤_merge(r, c, ≤_α, ≤_β);
29         end forall
30     end forall
31     return box;
32 end
```

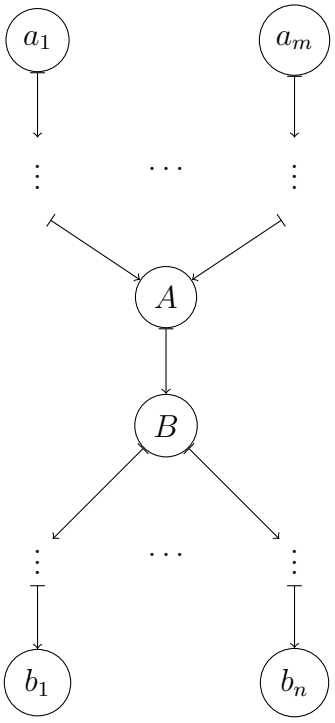


Figure 6.1: $\langle B, A\rangle \in \prec_i \implies b_j \sqsubseteq a_k$.

Figure 6.3 shows the case, where $\{b_1, \ldots, b_n\} = \{b \mid \langle b, B\rangle \in \leq_\beta \wedge b \neq B\}$ and $\{a_1, \ldots, a_m\} = \{a \mid \langle a, A\rangle \in \leq_\alpha\}$. We know that $B^\mathcal{I} \not\subseteq A^\mathcal{I}$ or $B^\mathcal{I} \cap (\neg A)^\mathcal{I} \neq \emptyset$, $b_j^\mathcal{I} \cap (\neg B)^\mathcal{I} = \emptyset$, $A^\mathcal{I} \cap (\neg a_k)^\mathcal{I} = \emptyset$. Although $B^\mathcal{I} \cap (\neg A)^\mathcal{I} \neq \emptyset$ it is unknown whether $b_j^\mathcal{I} \cap (\neg a_k)^\mathcal{I}$ is empty or not because $b_j^\mathcal{I} \subseteq B^\mathcal{I}$ and $(\neg A)^\mathcal{I} \supseteq (\neg a_k)^\mathcal{I}$ and thus neither $b_j \sqsubseteq a_k$ nor $b_j \not\sqsubseteq a_k$ is enforced for all j, k. ■

Proposition 4 *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $B \not\leq A$ is found in top-search, $\langle A, \top\rangle \in \leq_\alpha$ and $\langle B, \top\rangle \in \leq_\beta$, then for $\forall b_j \in \{b \mid \langle B, b\rangle \in \leq_\beta\} \cup \{B\}$ and $\forall a_k \in \{a \mid \langle a, A\rangle \in \leq_\alpha\} \cup \{A\}$ it follows that $b_j \not\leq a_k$.*

Figure 6.4 illustrates the case, where $\{a_1, \ldots, a_m\} = \{a \mid \langle a, A\rangle \in \leq_\alpha\}$ and $\{b_1, \ldots, b_n\} = \{b \mid \langle B, b\rangle \in \leq_\beta\}$. We prove the contrapositive: $b_j \leq a_k \implies B \leq A$. This follows due to the transitivity of the subsumption relationship. From the premise we know that $B \leq b_j$, $b_j \leq a_k$, $a_k \leq A$; thus we have $B \leq A$. ■

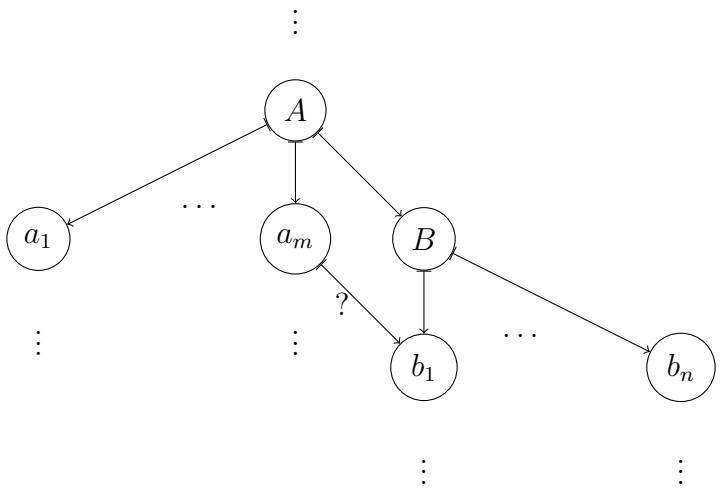


Figure 6.2: $\langle B, A\rangle \in \prec_i$: $b_j \sqsubseteq ?\, a_k$.

Similarly, we present the following propositions for bottom-search. Due to the symmetry between top-search & bottom-search the proofs for Propositions 5 to 8 are very similar to the proofs of Propositions 1 to 4 and are omitted.

Proposition 5 *When merging sub-terminology* $\leq_\beta$ *into* $\leq_\alpha$*, if* $\langle A, B\rangle \in \prec_i$ *is found in bottom-search,* $\langle \bot, A\rangle \in \leq_\alpha$ *and* $\langle \bot, B\rangle \in \leq_\beta$*, then for* $\forall b_j \in \{b \mid \langle B, b\rangle \in \leq_\beta\}$ *and* $\forall a_k \in \{a \mid \langle a, A\rangle \in \leq_\alpha\} \cup \{A\}$ *it follows that* $a_k \leq b_j$*.*

Proposition 6 *When merging sub-terminology* $\leq_\beta$ *into* $\leq_\alpha$*, if* $\langle A, B\rangle \in \prec_i$ *is found in bottom-search,* $\langle \bot, A\rangle \in \leq_\alpha$ *and* $\langle \bot, B\rangle \in \leq_\beta$*, then for* $\forall b_j \in \{b \mid \langle B, b\rangle \in \prec_\beta \wedge b \neq B\}$ *and* $\forall a_k \in \{a \mid \langle A, a\rangle \in \prec_\alpha \wedge a \neq A\}$ *it is necessary to calculate whether* $a_k \leq b_j$*.*

Proposition 7 *When merging sub-terminology* $\leq_\beta$ *into* $\leq_\alpha$*, if* $A \not\leq B$ *is found in bottom-search,* $\langle \bot, A\rangle \in \leq_\alpha$ *and* $\langle \bot, B\rangle \in \leq_\beta$*, then for* $\forall b_j \in \{b \mid \langle B, b\rangle \in \leq_\beta \wedge b \neq B\}$ *and* $\forall a_k \in \{a \mid \langle A, a\rangle \in \leq_\alpha\} \cup \{A\}$ *it is necessary to calculate whether* $a_k \leq b_j$*.*

Proposition 8 *When merging sub-terminology* $\leq_\beta$ *into* $\leq_\alpha$*, if* $A \not\leq B$ *is found in top-search,* $\langle \bot, A\rangle \in \leq_\alpha$ *and* $\langle \bot, B\rangle \in \leq_\beta$*, then for* $\forall b_j \in \{b \mid \langle b, B\rangle \in \leq_\beta\} \cup \{B\}$ *and* $\forall a_k \in \{a \mid \langle A, a\rangle \in \leq_\alpha\} \cup \{A\}$ *it follows that* $a_k \not\leq b_j$*.*

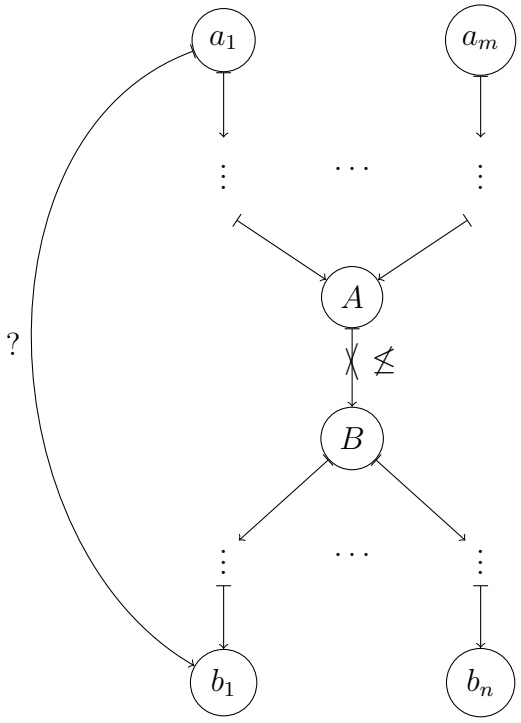


Figure 6.3: $B \not\leq A : b_i \sqsubseteq ? \, a_j$.

When merging a concept B, $\langle B, \top \rangle \in \leq_\beta$, the top-merge algorithm first finds for B the most specific position in the master sub-terminology $\leq_\alpha$ by means of *top-down* search. When such a most specific super-concept is found, this concept and all its super-concepts are naturally super-concepts of every sub-concept of B in the sub-terminology $\leq_\beta$, as is stated by Proposition 1. However, this newly found predecessor of B may not be necessarily a predecessor of some descendant of B in $\leq_\beta$. Therefore, the algorithm continues to find the most specific positions for all sub-concepts of B in $\leq_\beta$ according to Proposition 2. Algorithm 8 addresses this procedure.

Non-subsumption information can be told in the top-merge phase. Top-down search employed by top-merge must do subsumption tests somehow. In a canonical top-search procedure, as indicated by Algorithm 6, the branch search is stopped at this point. However, the conclusion that a merged concept B, $\langle B, \top \rangle \in \leq_\beta$, is not subsumed by a concept A, $\langle A, \top \rangle \in \leq_\alpha$, does not rule out the possibility of

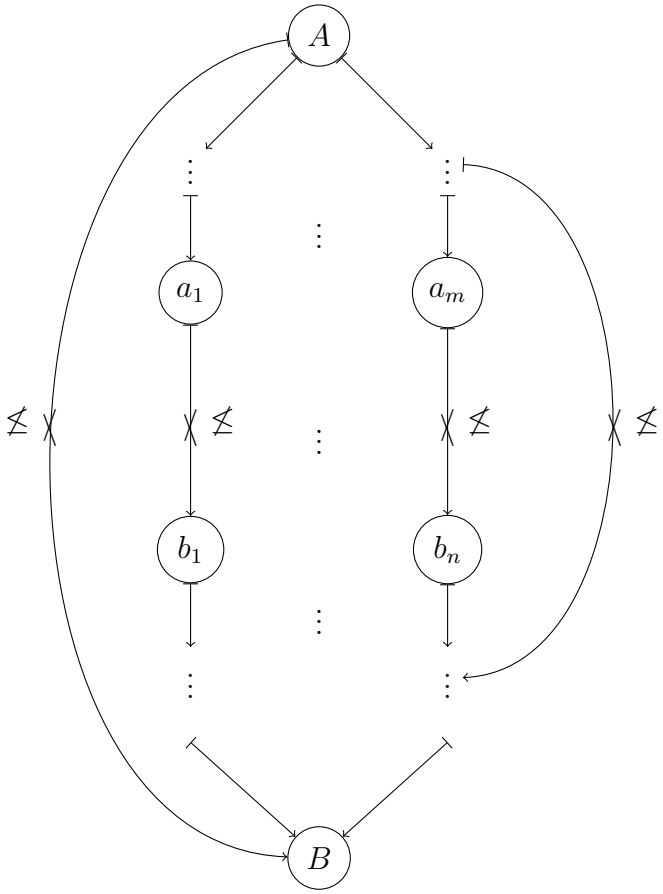


Figure 6.4: $B \not\preceq A \implies b_i \not\preceq a_j$.

$b_j \leq A$ with $b_j \in \{b \mid \langle b, B\rangle \in \prec_\beta\}$, which is not required in traditional top-search and may be abound in the top-merge procedure, and therefore must be followed by determining whether $b_j \leq A$. Otherwise, the algorithm is incomplete. Proposition 3 presents this observation. For this reason, the original top-search algorithm must be adapted to the new situation. Algorithm 9 is the updated version of the top-search procedure.

Algorithm 9 not only maintains told subsumption information by the set *green*, but also propagates told non-subsumption information by the set *red* for further

inference.[1] As addressed by Proposition 3, when the position of a merged concept is determined, the subsumption relationships between its successors and the *red* set are calculated. Furthermore, the subsumption relationship for the concept C and D in Algorithm 9 must be explicitly calculated even when the set *green* is empty. In the original top-search procedure (Algorithm 6), $C \prec_i D$ is implicitly derived if the set *green* is empty, which does not hold in the modified top-search procedure (Algorithm 9) since it does not always start from $\top$ anymore when searching for the most specific position of a concept.

The pseudocode of Listing B.1 and B.2 in Appendix B illustrates a more concise description for Algorithm 8 and 9.

6.2.3 Example

We use an example TBox to illustrate the algorithm further. Given an ontology with a TBox defined by Figure 6.5(a), which only contains simple concept subsumption axioms, Figure 6.5(b) shows the subsumption hierarchy.

Suppose that the ontology is clustered into two groups in the divide phase: $\Delta_\alpha = \{A_2, A_3, A_5, A_7\}$ and $\Delta_\beta = \{A_1, A_4, A_6, A_8\}$. They can be classified independently, and the corresponding subsumption hierarchies are shown in Figure 6.6.

In the merge phase, the concepts from $\leq_\beta$ are merged into $\leq_\alpha$. For example, Figure 6.7 shows a possible computation path where $A_4 \leq A_5$ is being determined.[2] If we assume a subsumption relationship between two concepts is proven when the parent is added to the set *box* (see Line 15, Algorithm 9), Figure 6.8 shows the subsumption hierarchy after $A_4 \leq A_5$ has been determined.

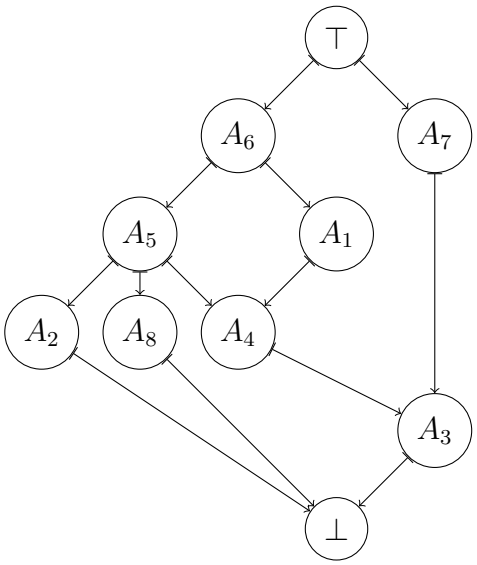
[1] Our implementation of Algorithm 9 treats subsumptions cycles as *synonyms*. For example, if $rat \sqsubseteq mouse$ and $mouse \sqsubseteq rat$, the two concepts are collapsed into one, *rat/mouse*. For sake of conciseness we do not show these details in Algorithm 9.

[2] This process does not show a full calling order of computing $A_4 \leq A_5$ for sake of brevity. For instance, $\top\_merge(A_7, A_6, \leq_\alpha, \leq_\beta)$ is not shown.

$\mathcal{T}$:

$A_6 \sqsubseteq \top$ $A_4 \sqsubseteq A_1$
$A_7 \sqsubseteq \top$ $A_3 \sqsubseteq A_7$
$A_5 \sqsubseteq A_6$ $A_3 \sqsubseteq A_4$
$A_1 \sqsubseteq A_6$ $\bot \sqsubseteq A_2$
$A_2 \sqsubseteq A_5$ $\bot \sqsubseteq A_8$
$A_8 \sqsubseteq A_5$ $\bot \sqsubseteq A_3$
$A_4 \sqsubseteq A_5$

(a) The TBox given.



(b) The classified terminology hierarchy.

Figure 6.5: An example ontology.

6.3 Termination, Soundness, and Completeness

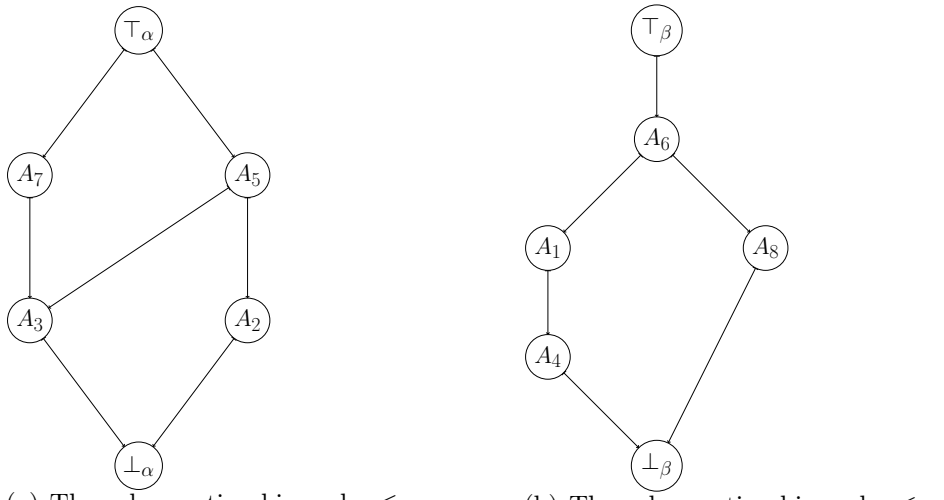
Lemma 1 *The top-merge algorithm, Algorithm 8, always terminates.*

During the process of merging two classified terminologies by using $\top\_merge$ from $\top_\alpha$ and $\top_\beta$, either $\top\_merge$ or $\top\_search^*$ is applied to the successors of one of the concerned concepts.

First of all, there can not exist a subsumption cycle between a concerned concept and its successors, because the involved concepts are collapsed and treated as synonyms once such a cycle is detected. Therefore, without an infinite execution on testing a subsumption cycle between a concerned concept and its successors, a limited number of successors are explored, the search continues until $\bot$ is taken into account, and then the algorithm terminates. Consequently, Algorithm $\top\_merge$ always terminates. ■

Similarly, we can establish the following claims:

Lemma 2 *The bottom-merge algorithm always terminates.*



(a) The subsumption hierarchy $\leq_\alpha$. (b) The subsumption hierarchy $\leq_\beta$.

Figure 6.6: The subsumption hierarchy over divisions.

Theorem 1 *Algorithm 4 always terminates.*

With Lemma 1 and 2, it is easy to prove Theorem 1.

Definition 13 *Let* $S_1 = (x_0, x_1, \ldots, x_m)$ *and* $S_2 = (y_0, y_1, \ldots, y_n)$ *be two paths, and the concatenation of* $S_1 \bullet S_2 = (x_0, x_1, \ldots, x_m, y_0, y_1, \ldots, y_n)$. *For the empty path* λ *and a path* S, *it holds that* $S \bullet \lambda = S$, *and* $\lambda \bullet S = S$.

Definition 14 *In a classified terminology* $\leq$, *a concept* C*'s upper inheritance* $U(C)$ *is a path as follows:*

$$U(C) = \begin{cases} \lambda & C \doteq \top, \\ U(D) \bullet (D) & C \prec D, D \not\doteq \top \end{cases} \tag{6.1}$$

It is obvious that the following proposition hold:

Proposition 9 *For any concept* C *in a classified terminology, there must exist at least one upper inheritance* $U(C)$.

Similarly, we get the following symmetric claims:

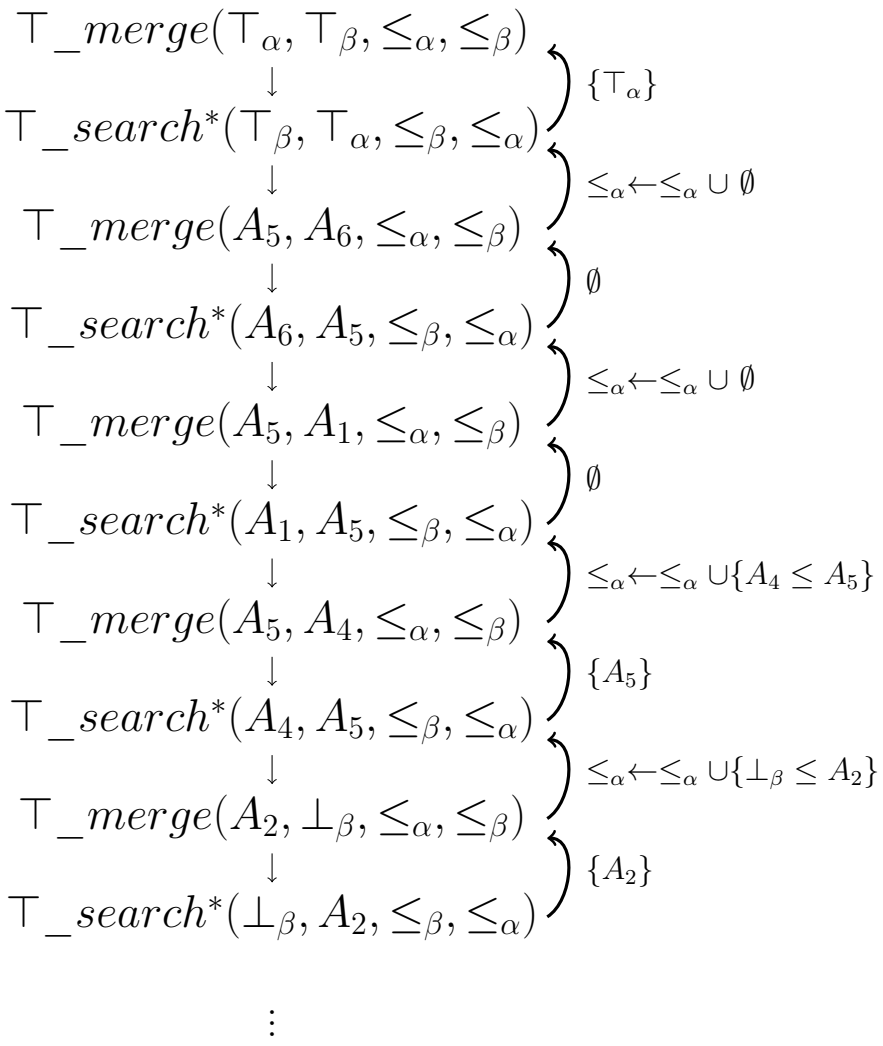


Figure 6.7: The computation path of determining $A_4 \leq_i A_5$.

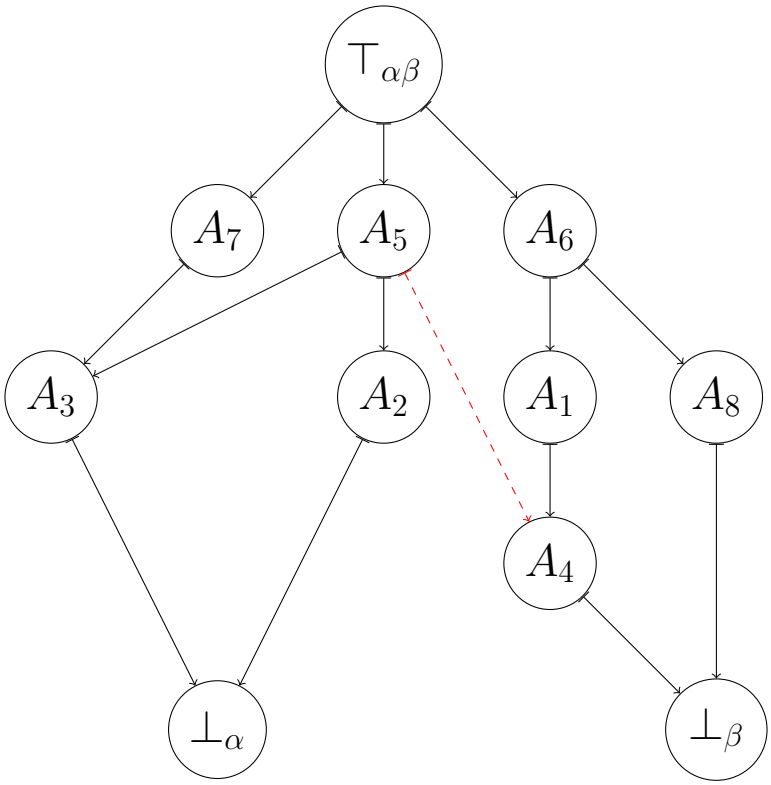


Figure 6.8: The subsumption hierarchy after $A_4 \leq A_5$ has been determined.

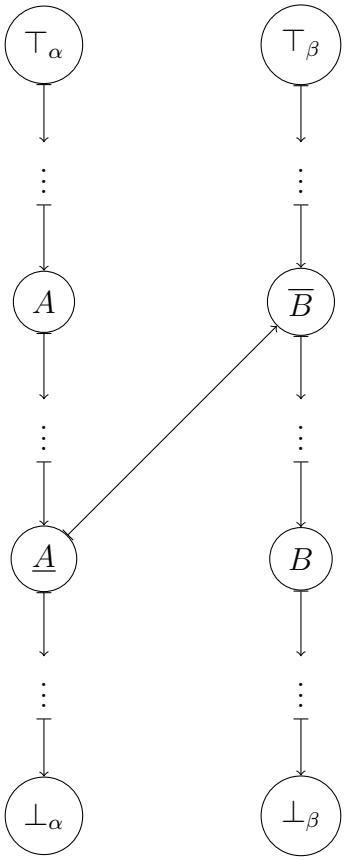


Figure 6.9: $B \leq A$.

Definition 15 *In a classified terminology $\leq$, a concept C's lower inheritance $L(C)$ is a path as follows:*

$$L(C) = \begin{cases} \lambda & C \doteq \bot, \\ (D) \bullet L(D) & D \prec C, D \not\doteq \bot \end{cases} \tag{6.2}$$

Proposition 10 *For any concept C in a classified terminology, there must exist at least one lower inheritance $L(C)$.*

Proposition 11 *The subsumption checking procedure $\leq?$ is correct, i.e., it holds that $\mathcal{O} \models C \sqsubseteq D \Leftrightarrow \leq?(C, D) \rightarrow true$.*

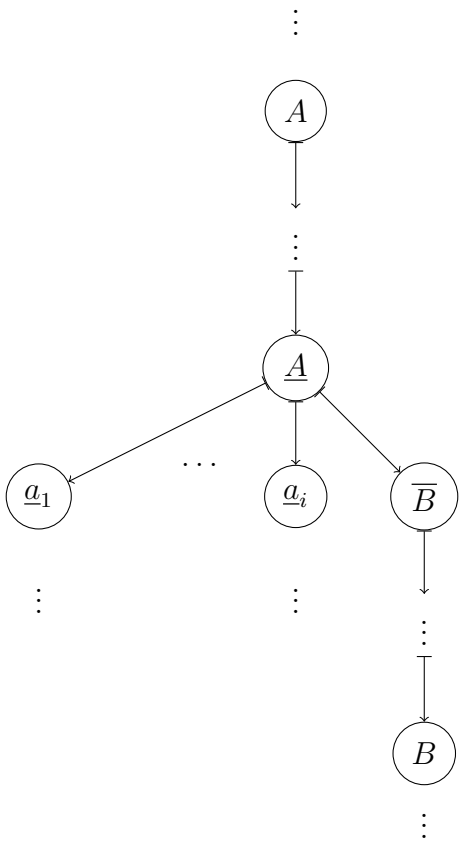


Figure 6.10: $B \leq A$ is derived.

Lemma 3 (Soundness of top_merge) *When merging* $\leq_\beta$ *into* $\leq_\alpha$*, for* $\forall A : \langle A, \top_\alpha \rangle \in \leq_\alpha$ *and* $\forall B : \langle B, \top_\beta \rangle \in \leq_\beta$*, if the* $\top$*_merge algorithm starting from* $\top_\alpha$ *and* $\top_\beta$ *infers that* $B \leq A$*, then* $\mathcal{O} \models B \sqsubseteq A$*.*

This proof is based on Proposition 11. We prove this lemma by contradiction. Let us assume that Algorithm 9 derives $B \leq A$ but $\mathcal{O} \models B \not\sqsubseteq A$.

When the $\top$*_merge* algorithm derives $B \leq A$, there must exist $L(A)$ and $U(B)$ such that, $\exists \underline{A} \in (A) \bullet L(A)$ and $\exists \overline{B} \in U(B) \bullet (B)$, and, as claimed in Propositions 9 and 10, the algorithm determines $\overline{B} \prec \underline{A}$. This means that $\overline{B} \leq \underline{A}$ must be the result of calling $\leq?(\overline{B}, \underline{A})$ at line 14 of Algorithm 9. This situation is shown as Figure 6.9.

In the process of determining $\overline{B} \prec \underline{A}$ all children $\underline{A}_i$ of $\underline{A}$ are tested whether they subsume $\overline{B}$ and the calls of $\leq?(\overline{B}, \underline{A}_i)$ must always have returned *false*, as shown in line 6 of Algorithm 9 and in Figure 6.10. Therefore, $\overline{B} \prec \underline{A}$ is derived.

We already know $\leq?(B, \overline{B}) \rightarrow$ *true* and $\leq?(\underline{A}, A) \rightarrow$ *true*, $\leq?(B, A) \rightarrow$ *true*. So, due to the correctness of $\leq?$ and the transitivity of the subsumption relationship it holds that $\mathcal{O} \models B \sqsubseteq A$, which contradicts our assumption. ■

Similarly, the following corresponding claim can be established.

Lemma 4 (Soundness of bottom_merge) *When merging* $\leq_\beta$ *into* $\leq_\alpha$*, for* $\forall A : \langle \bot_\alpha, A \rangle \in \leq_\alpha$ *and* $\forall B : \langle \bot_\beta, B \rangle \in \leq_\beta$*, if the* $\bot$*_merge algorithm starting from* $\bot_\alpha$ *and* $\bot_\beta$ *infers that* $A \leq B$*, then* $\mathcal{O} \models A \sqsubseteq B$*.*

Following Lemma 3 and 4, as well as Theorem 1, the *soundness* of the merge algorithm is established.

Theorem 2 (Soundness of merge algorithm) *For a merged terminology* $\leq$ *it holds, if* $\langle C, D \rangle \in \leq$*, then* $\mathcal{O} \models C \sqsubseteq D$*.*

Lemma 5 (Completeness of top_merge) *If* $\mathcal{O} \models B \sqsubseteq A$*, then for* $\forall A \subseteq \Delta_\alpha$ *and* $\forall B \subseteq \Delta_\beta$*, the top-merge algorithm infers* $B \leq A$*, when it merges* $\leq_\beta$ *into* $\leq_\alpha$ *starting from* $\top_\alpha$ *and* $\top_\beta$*.*

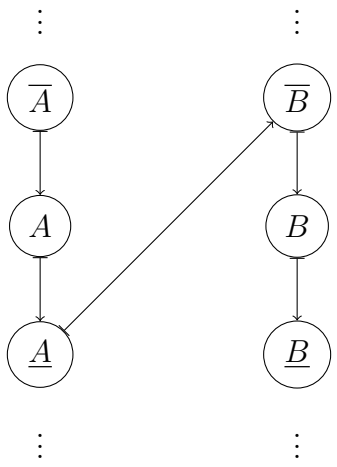


Figure 6.11: $\mathcal{O} \models B \sqsubseteq A$.

This proof is based on Proposition 11.

Let $P(A)$ be the set of all paths from $\top$ to $\bot$ that contain A, i.e. $\forall U(A), L(A) : \{U(A) \bullet L(A)\} \subseteq P(A)$. $P(A) \neq \emptyset$ by Propositions 9 and 10. Similarly, $P(B) \neq \emptyset$ is the set of all paths from $\top$ to $\bot$ that contain B. Because $\mathcal{O} \models B \sqsubseteq A$, $P(A) \cap P(B) \neq \emptyset$, i.e. $\exists U(A), L(A), U(B), L(B) : U(A) \bullet L(A) = U(B) \bullet L(B)$. Lemma 5 can be proved by structural induction: If $\mathcal{O} \models B \sqsubseteq A$, then $B \leq A$ can be derived by searching on $U(A) \bullet L(A) = U(B) \bullet L(B)$ with Algorithm 9. The proof for the base cases are trivial, so we just give the induction part.

Let $A \prec \overline{A}$, $\underline{A} \prec A$, $B \prec \overline{B}$, $\underline{B} \prec B$, and $\overline{B} \prec \underline{A}$. That is to say, $\overline{A} \in U(A)$, $\underline{A} \in L(A)$, $\overline{B} \in U(B)$, and $\underline{B} \in L(B)$, as is shown by Figure 6.11.

Since $\mathcal{O} \models \overline{B} \sqsubseteq \overline{A}$, we have $\leq?(\overline{B}, \overline{A}) \rightarrow \mathit{true}$: Algorithm 9 puts $\overline{A}$ into *green* at line 7. And then $\top\_\mathit{search}^*$ is applied to $\overline{B}$ and every element of *green*, including $\overline{A}$, as is shown by line 21 of Algorithm 9. $\top\_\mathit{search}^*(\overline{B}, \overline{A}, \leq_\beta, \leq_\alpha)$ tests the subsumption relationships between $\overline{B}$ and every child of $\overline{A}$, including A, at line 6. This process recursively continues to test $\overline{B}$ and $\underline{A}$. At this point, all children of $\underline{A}$ do not subsume $\overline{B}$ and thus are put into *red*, so $\mathit{green} = \emptyset$ and $\mathit{box} \leftarrow \{\underline{A}\}$, as is shown by line 22 of Algorithm 9 and Figure 6.12.

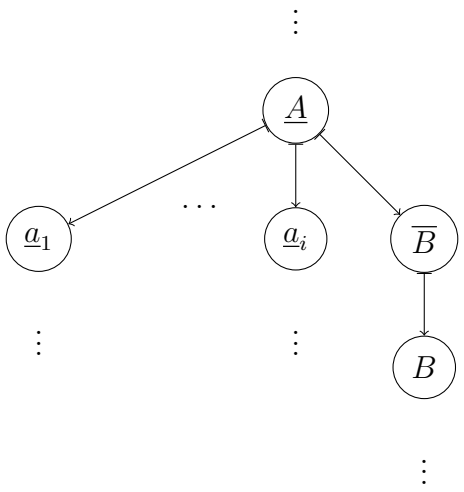


Figure 6.12: $\overline{B} \prec \underline{A}$ is derived.

Now, Algorithm 9 derives $\leq?(\overline{B}, \underline{A}) \rightarrow \mathit{true}$, $\leq?(\underline{A}, A) \rightarrow \mathit{true}$, and $\leq?(B, \overline{B}) \rightarrow \mathit{true}$, it will be determined $\leq?(B, A) \rightarrow \mathit{true}$. ■

Correspondingly, the completeness of the bottom-merge algorithm is established by Lemma 6.

Lemma 6 (Completeness of bottom_merge) *If $\mathcal{O} \models A \sqsubseteq B$, then for $\forall A \subseteq \Delta_\alpha$ and $\forall B \subseteq \Delta_\beta$, the bottom-merge algorithm infers $A \leq B$, when it merges $\leq_\beta$ into $\leq_\alpha$ starting from $\bot_\alpha$ and $\bot_\beta$.*

From Lemma 5 and 6, we can conclude that the merge algorithm is complete.

Theorem 3 (Completeness of merge algorithm) *If $\mathcal{O} \models C \sqsubseteq D$, the merge algorithm will infer that $C \leq D$.*

6.4 Partitioning

Partitioning is an important part of this algorithm. It is the main task in the dividing phase. In contrast to simple problem domains such as sorting integers, where the merge phase of a standard merge-sort does not require another sorting, DL ontologies

Algorithm 10: *cluster*(G)

```
   input  : G: the told subsumption graph
   output: R: the concept names partitions
 1 begin
 2 |  R ← ∅;
 3 |  visited ← ∅;
 4 |  N ← get_top_children(⊤, G);
 5 |  foreach n ∈ N do
 6 |  |  P ← {n};
 7 |  |  visited ← visited ∪ {n};
 8 |  |  R ← R ∪ {build_partition(n, visited, G, P)};
 9 |  end foreach
10 |  return R;
11 end
```

might entail numerous subsumption relationships among concepts. Building a terminology with respect to the entailed subsumption hierarchy is the primary function of DL classification. We therefore assumed that some heuristic partitioning schemes that make use of known subsumption relationships may improve reasoning efficiency by requiring a smaller number of subsumption tests, and this assumption has been proved by our experiments, which are described in Section 6.5.

So far, we have presented an ontology partitioning algorithm by using only told subsumption relationships that are directly derived from concept definitions and axiom declarations. Any concept that has at least one told super- and one sub-concept, can be used to construct a told subsumption hierarchy. Although such a hierarchy is usually incomplete and many entailed subsumptions are missing, it contains already known subsumptions indicating the closeness between concepts w.r.t. subsumption. Such a raw subsumption hierarchy can be represented as a directed graph with only one root, the $\top$ concept. A heuristic partitioning method can be defined by traversing the graph in a breadth-first way, starting from $\top$, and collecting traversed concepts into partitions. Algorithm 10 and 11 address this procedure.

Algorithm 11: *build_partition*$(n, visited, G, P)$

```
input  : n: an concept name
         visited: a list recording visited concept names
         G: the told subsumption graph
         P: a concept names partition
output: R: a concept names partition
1  begin
2      R ← ∅;
3      N ← get_children(n, visited, G, P);
4      foreach n' ∈ N do
5          if n' ∉ visited then
6              P ← P ∪ {n'};
7              visited ← visited ∪ {n'};
8              build_partition(n', visited, G, P);
9          end if
10     end foreach
11     R ← P;
12     return R;
13 end
```

6.5 Evaluation

Our experimental results clearly show the potential of merge-classification. We could achieve speedups up to a factor of 4 by using a maximum of 8 parallel workers, depending on the particular benchmark ontology. This speedup is in the range of what we expected and comparable to other reported approaches, e.g., the experiments reported for the ELK reasoner [50, 51] also show speedups of up to a factor of 4 when using 8 workers, although a specialized polynomial procedure is used for $\mathcal{EL}+$ reasoning that seems to be more amenable to concurrent processing than standard tableau methods.

We have designed and implemented a concurrent version of the algorithm so far. Our program is implemented on the basis of the well-known reasoner JFact, which is open-source and implemented in Java.[3][4] We modified JFact such that we

[3] http://github.com/kejia/mc
[4] http://jfact.sourceforge.net

Algorithm 12: *schedule_merging*(q)

```
   input  : q: the job queue
   output: r: the updated job queue
 1 begin
 2 |   got ← false;
 3 |   while ¬got ∧ size(q) > 0 do
 4 |   |   bolt ← dequeue(q);
 5 |   |   nut ← dequeue(q);
 6 |   |   if ¬null?(bolt) ∧ ¬null?(nut) then
 7 |   |   |   got ← true;
 8 |   |   |   enqueue(q, merge(bolt, nut));
 9 |   |   else if ¬null?(bolt) then
10 |   |   |   enqueue(q, bolt);
11 |   |   |   bolt ← null;
12 |   |   else if ¬null?(nut) then
13 |   |   |   enqueue(q, nut);
14 |   |   |   nut ← null;
15 |   |   end if
16 |   end while
17 |   r ← q;
18 |   return r;
19 end
```

can execute a set of JFact reasoning kernels in parallel in order to perform the merge-classification computation. We try to examine the effectiveness of the merge-classification algorithm by adapting such a mature DL reasoner.

6.5.1 Experiment

A multi-processor computer, which has 4 octa-core processors and 128G memory installed, was employed to test the program. The Linux OS and 64-bit OpenJDK 6 were employed in the tests. The JVM was allocated at least 16G memory initially, given that at most 64G physical memory was accessible. Most of the test cases were chosen from ORE 2012 data sets. Table 6.1 shows the test cases' metrics.

Each test case ontology was classified with the same setting except for an increased number of workers. Each worker is mapped to an OS thread, as indicated by the Java specification. Figures 6.13 and 6.14 show the test results.

In our initial implementation, we used an *even-partitioning* scheme. That is to say concept names are randomly assigned to a set of partitions. For the majority of the above-mentioned test cases we observed a small performance improvement below a speedup factor of 1.4, for a few an improvement of up to 4, and for others only a decrease in performance. Much overhead was shown in these test cases.

As mentioned in Section 6.4, we assumed that a heuristic partitioning might promote a better reasoning performance, e.g., a partitioning scheme considering subsumption axioms. This idea is addressed by Algorithm 10 and 11.

Another issue that happens when partitions are merged in a shared-memory parallel environment is *racing*. In the merge-classification case, each worker puts the classified partition to a shared queue, and then picks two out of it to merge them. Workers race with each other to get merging pairs. That is to say which and how many partitions some worker gets is indeterminate. This may become the source of deadlocks or other concurrency issues. We designed a schedule algorithm to constrain the race from such concurrency issues. Algorithm 12 ensures that each worker starts

ontology	expressivi	concept count	axiom count
adult_mouse_anatomy	$\mathcal{ALE}+$	2753	9372
amphibian_gross_anatomy	$\mathcal{ALE}+$	701	2626
c_elegans_phenotype	$\mathcal{ALEH}+$	1935	6170
cereal_plant_trait	$\mathcal{ALEH}$	1051	3349
emap	$\mathcal{ALE}$	13731	27462
environmental_entity_logical_definitions	$\mathcal{SH}$	1779	5803
envo	$\mathcal{ALEH}+$	1231	2660
fly_anatomy	$\mathcal{ALEI}+$	6222	33162
human_developmental_anatomy	$\mathcal{ALEH}$	8341	33345
medaka_anatomy_development	$\mathcal{ALE}$	4361	9081
mpath	$\mathcal{ALEH}+$	718	4315
nif-cell	$\mathcal{S}$	376	3492
sequence_types_and_features	$\mathcal{SH}$	1952	6620
teleost_anatomy	$\mathcal{ALER}+$	3036	11827
zfa	$\mathcal{ALEH}+$	2755	33024

Table 6.1: Metrics of the test cases—merge-classification.

merging if and only if the worker has obtained two partitions.

We implemented Algorithms 10, 11, and 12, and tested our program. Our assumption has been proved by the test: Heuristic partitioning may improve reasoning performance where blind partitioning can not.

6.5.2 Discussion

Our experiment shows that with a heuristic divide scheme the merge-classification algorithm can increase reasoning performance. However, such performance promotion is not always tangible. In a few cases, the parallelized merge-classification merely degrades reasoning performance. The actual divide phase of our algorithm can influence the performance by creating better or worse partitions.

A heuristic divide scheme may result in a better performance than a blind one. According to our experience, when the division of the concepts from the domain is basically random, sometimes divisions contribute to promoting reasoning performance, while sometimes they do not. A promising heuristic divide scheme seems to be in grouping a family of concepts, which have potential subsumption relationships, into the same partition. Evidently, due to the presence of non-obvious subsumptions, it is hard to guess how to achieve such a good partitioning. We tried to make use of obvious subsumptions in axioms to partition closely related concepts into the same group. The tests demonstrate a clear performance improvement in a number of cases.

While in many cases merge-classification can improve reasoning performance, for some test cases its practical effectiveness is not yet convincing. We have investigated the factors that influence the reasoning performance for these cases, but giving a clear answer in such a complex context as concurrent reasoning is very difficult. The cause may be the large number of GCI axioms found in some ontologies. Even with a more refined divide scheme, those GCI axioms can cause inter-dependencies between partitions, and may cause in the merge phase an increased number of subsumption tests.

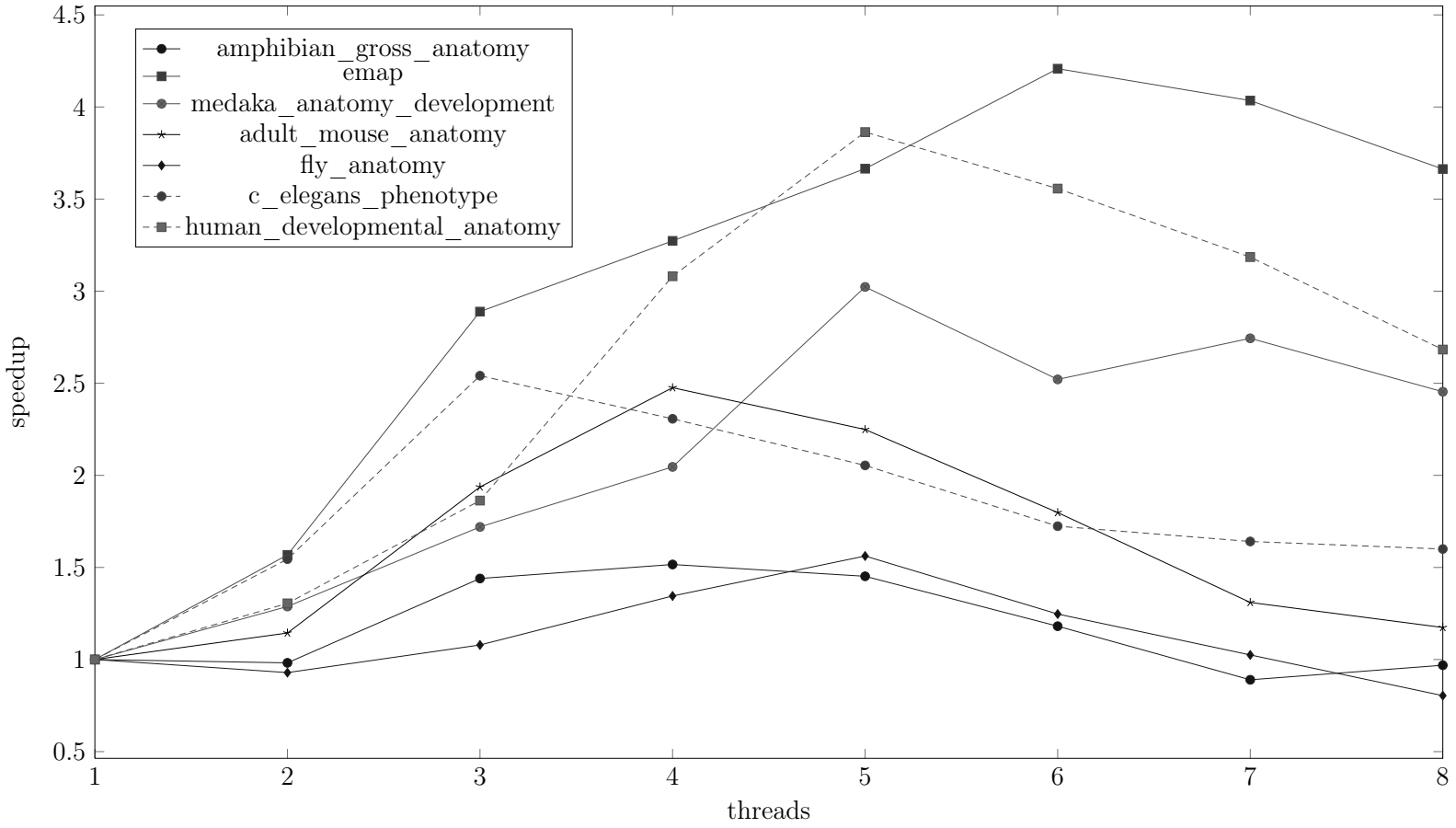


Figure 6.13: The performance of parallelized merge-classification—I.

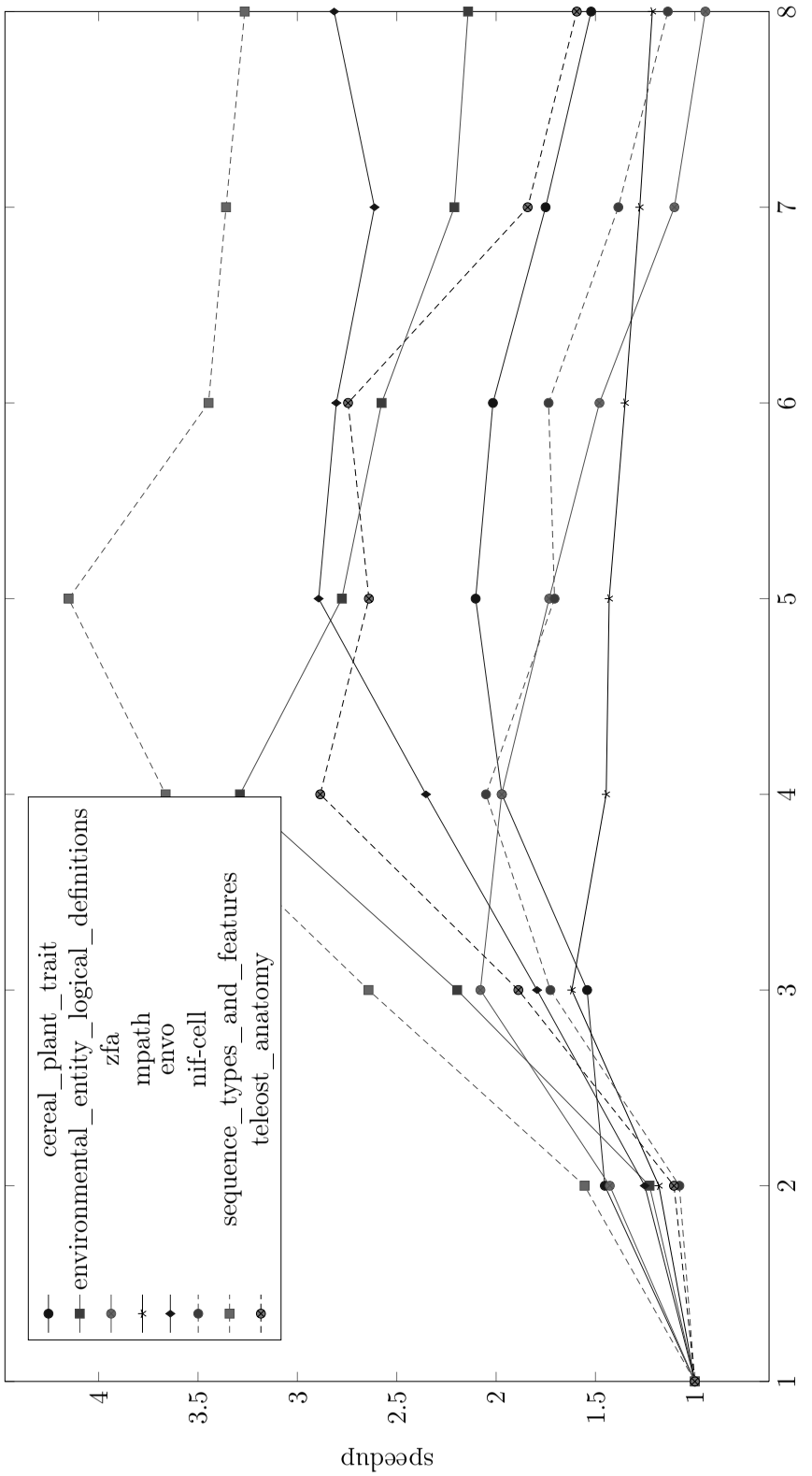


Figure 6.14: The performance of parallelized merge-classification—II.

Also, the non-determinism of the merging schedule, i.e., the unpredictable order of merging divides, needs to be effectively solved in the implementation, and racing conditions between merging workers as well as the introduced overhead may decrease the performance. In addition, the limited performance is caused by the experimental environment: Compared with a single chip architecture, the 4-chip-distribution of the 32 processors requires extra computational overhead, and the memory and thread management of JVM may decrease the performance of our program.

6.6 Summary

The approach presented in this research has been motivated by the observation that: *(i)* multi-processor/core hardware is becoming ubiquitously available but standard OWL reasoners do not yet make use of these available resources; *(ii)* although most OWL reasoners have been highly optimized and impressive speed improvements have been reported for reasoning in the three tractable OWL profiles, there exist a multitude of OWL ontologies that are outside of the three tractable profiles and require long processing times even for highly optimized OWL reasoners. Concurrent computing has emerged as a possible solution for achieving a better scalability in general and especially for such difficult ontologies.

One of the most important obstacles in successfully applying concurrent computing is the management of overhead caused by concurrency. An important factor is that the load introduced by using concurrent computing in DL reasoning is usually remarkable. Concurrent algorithms that cause only a small overhead seem to be the key to successfully apply concurrent computing to DL reasoning.

Our merge-classification algorithm uses a *divide and conquer* scheme, which is potentially suitable for low overhead concurrent computing since it rarely requires communication among divisions. The empirical tests show that the merge-classification algorithm can improve reasoning performance in a number of cases. At present our

work adopts a heuristic *partitioning* scheme at the divide phase. Different divide schemes may produce different reasoning performances. Future work may investigate better divide methods.

The advantage of merge-classification method is that reasoning performance can be effectively improved without a *large* number of processors involved. The disadvantage of merge-classification is that it needs elaborate *dividing* schemes in some cases. Furthermore, The implementation of merge-classification algorithm is complex: *(i)* the canonical classification methods such as *top-search & bottom-search* can not be used directly and must be adapted; *(ii)* merge-classification method needs designing and implementing efficient concurrent algorithms for managing the merging procedure.

Chapter 7

Conclusion

7.1 Future Work

The objective of this research is to use *concurrent computing* to get *scalability* in DL reasoning. *Concurrent computing* can hardly play a role in such a sophisticated area as automated reasoning unless computing components are decoupled elaborately. So, future work of this research should endeavor to search for easily decoupled computing components. For example, exploring more effective ways of dividing knowledge bases for merge-classification algorithm. Such research may even happen on reforming the fundamental syntax and semantics of DL in order to obtain independent reasoning components. Novel logics that are concurrent-computing-oriented may even be invented—anyway, DL and almost all other computational logics are rooted in *mathematical logic* where such practical factors as *concurrency* are not emphasized.

Furthermore, instead of tableau-based algorithms, reasoning methods that is more suitable for concurrent computing may be introduced into DL. So far, tableau-based algorithms have been shown as the most efficient techniques in DL reasoning. But, this conclusion is drawn in *sequential computing* context; maybe, some other reasoning techniques will surpass tableau-based ones in *concurrent computing* context.

After obtaining theoretical methods using concurrent computing, researchers en-

counter the practical engineering issue: How should the methods be effectively implemented?

An obstacle of exploring *concurrent computing* algorithms is implementation. Future works of this research should adopt data structures, algorithms, languages, tools, platforms, libraries, and patterns that are more suitable and more efficient for concurrent computing. This requires the researchers to be familiar with the techniques of implementing *concurrent computing*.

Some implementation technologies may largely increase performance of concurrent computing algorithms. This research attempts to obtain scalable DL reasoning performance by using concurrent computing, which includes much more practical factors than theoretical ones. Among those practical factors, the most significant two are *efficiently implementing* and *efficient implementation*.

Efficiently implementing such a complex software system as a DL reasoner is a challenging task. After having implemented the basic tableaux, a number of optimization techniques should be added; otherwise, the reasoner's performance is unacceptable. What's more, the functionalities implementing a variety of novel optimization techniques and more powerful expressivity need to be added into the reasoner. It is the software architecture that determines how possible and how easy it is to add those new functionalities. We rarely agree that a reasoner which can not further evolve to reason with a more expressive language is efficiently implemented. This involves how flexible and extensible implementation architectures are. Future work may explore reasoner implementation architectures that have a small degree of component coupling.

On the other hand, given the same algorithm, different implementations have different performance. An *efficient implementation* involves a number of details of organizing and operating data, which are hidden behind algorithm designs. The implementation details greatly influence the final performance of a reasoner: An efficient implementation of an algorithm may produce more surprising performance

improvement than inefficient ones. For future reasoner developers, a good method to accomplish an efficient implementation is to study those well-known systems' source code, besides mastering the general data structure and algorithm knowledge.

Both topics are not easy, and when concurrent computing is involved, the problematic issues triple. Besides designing an elegant architecture and selecting suitable design patterns, an efficient weapon that should be taken into account for reducing the gap between a concurrent algorithm design and its implementation is *functional programming*.

Functional programming has been considered to be suitable for concurrent computing. The *immutability* of functional programming makes state-changing be reduced to the minimum, as can largely improve the robustness of concurrent programs. A number of problems of concurrent programs are generated by concurrent state-changing, and it is not easy even for experts to find out and to fix this sort of problems. Functional programming will persist computing objects' states, and therefore such bugs can be avoided to a great extent. Furthermore, the *immutability* of functional programming is also suited to tableaux, the preliminary method used by DL reasoning: States of branches can be saved without much extra work. The advantages of functional programming can help implement more robust and flexible reasoning systems that adopt concurrent computing. Thus, functional programming may play a role in concurrent DL reasoning research in the future.

7.2 Summary

DL reasoning is a hard topic. A number of optimization techniques have been extensively researched. Now, practicable DL reasoning is not feasible unless those optimization techniques are taken into account. With the progress of semantic web technology, knowledge bases are becoming more and more complex. It seems that this tendency will continue. DL reasoners should evolve corresponding reasoning

abilities, one of which is *reasoning scalability*. A means to get scalability is using *concurrent computing*.

Concurrent computing can not be easily adopted into such a complex area as automated reasoning. The primary reasoning algorithms and optimization techniques are not suitable for concurrent processing. A reasoning algorithm generally operates on data that is difficult to be decoupled, as is an important obstacle hindering performance improvement of concurrent reasoning. Concurrent reasoning can not play a role unless either reasoning data or reasoning operations are finely decoupled.

This research investigated a collection of algorithms that can reason about DL knowledge bases in parallel.

DL TBox classification calculates all concept subsumption relationships entailed in a knowledge base. Each subsumption calculation is independent of the others, and thus classification can be computed in parallel. Our research worked on this idea, and has shown that *reasoning scalability* can be gained by enrolling a growing number of processors. This optimization combines concurrent computing and tableau-based TBox classification.

Besides classification, this research investigated how general tableau-based DL reasoning algorithms can get benefits from concurrent computing. Tableau-based algorithms have been shown as pretty efficient DL reasoning techniques. Before this research, there have been researchers who tried to parallelize manipulating disjunctive branches of a tableau expansion tree. This research filled a gap by processing conjunctive branches of a tableau expansion tree in parallel. Reasoning scalability can be observed in the corresponding experiments. This research combines concurrent computing with general tableau-based DL reasoning algorithms.

This research further investigated a more elaborate concurrent TBox classification method. *Divide and conquer* strategy was used to decouple operations and data of *top-search & bottom-search* algorithm, which is more efficient and more popularly used than brute-force testing. In this method, both *divide* and *conquer* stages can

be processed in parallel. It has been shown that scalable performance can be gained in a number of cases. This work combines concurrent computing with *top-search & bottom-search* algorithm by *divide and conquer* strategy.

DL has progressed for twenty years or so, and is playing a growing important role in semantic web era. This research is expected to contribute some knowledge to developing the next generation of reasoners that are endowed with reasoning scalability.

Bibliography

[1] Resource Description Framework (RDF). http://www.w3.org/RDF/ (2 2004)

[2] OWL 2 web ontology language structural specification and functional-style syntax. http://www.w3.org/TR/owl2-syntax (10 2009)

[3] Web Ontology Language (OWL). http://www.w3.org/OWL/ (12 2012)

[4] RIF Overview (Second Edition). http://www.w3.org/TR/rif-overview/ (2 2013)

[5] SPARQL 1.1 Overview. http://www.w3.org/TR/sparql11-overview/ (3 2013)

[6] W3C Semantic Web Activity (12 2013), http://www.w3.org/2001/sw/

[7] Alba, E.: Parallel evolutionary algorithms can achieve super-linear performance. Information Processing Letters 82(1), 7–13 (2002)

[8] Aslani, M., Haarslev, V.: Towards parallel classification of TBoxes. In: Proceedings of the 2008 International Workshop on Description Logics. vol. 353 (2008)

[9] Aslani, M., Haarslev, V.: Parallel TBox classification in description logics—first experimental results. In: Proceedings of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence. pp. 485–490 (2010)

[10] Baader, F., Brandt, S., Lutz, C.: Pushing the EL envelope. In: 2005 International Joint Conference on Artificial Intelligence. vol. 19, p. 364 (2005)

[11] Baader, F., Bürckert, H., Heinsohn, J., Hollunder, B., Müller, J., Nebel, B., Nutt, W., Profitlich, H.: Terminological knowledge representation: A proposal for a terminological logic. Tech. rep., Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI) (1992)

[12] Baader, F., Hollunder, B., Nebel, B., Profitlich, H.J., Franconi, E.: An empirical analysis of optimization techniques for terminological representation systems. Applied Intelligence 4(2), 109–132 (1994)

[13] Baader, F., Nutt, W.: Basic description logics. In: Baader, F., Calvanese, D., D., M., Nardi, D., Patel-Schneider, P. (eds.) The description logic handbook: theory, implementation, and applications, chap. 2. Cambridge University Press (2003)

[14] Baader, F., et al.: The description logic handbook: theory, implementation, and applications. Cambridge University Press (2003)

[15] Baumgartner, P., Furbach, U., Niemelä, I.: Hyper tableaux. Logics in Artificial Intelligence 1126, 1–17 (1996)

[16] Bock, J.: Parallel computation techniques for ontology reasoning. In: Proceedings of the 7th International Conference on the Semantic Web. pp. 901–906 (2008)

[17] Brachman, R.J.: What's in a concept: Structural foundations for semantic networks. International Journal of Man-Machine Studies 9(2), 127–152 (1977)

[18] Brachman, R.J., Schmolze, J.G.: An overview of the KL-ONE knowledge representation system. Cognitive Science 9(2), 171–216 (1985)

[19] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The DL-Lite family. Journal of Automated Reasoning 39(3), 385–429 (2007)

[20] Dentler, K., Guéret, C., Schlobach, S.: Semantic web reasoning by swarm intelligence. In: The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009). p. 1 (2009)

[21] Ding, Y.: Tableau-based reasoning for description logics with inverse roles and number restrictions. Ph.D. thesis, Concordia University (2008)

[22] Donaldson, V., Berman, F., Paturi, R.: Program speedup in a heterogeneous computing network. Journal of Parallel and Distributed Computing 21(3), 316–322 (1994)

[23] Donini, F.M., Lenzerini, M., Nardi, D., Nutt, W.: The complexity of concept languages. Information and Computation 134(1), 1–58 (1997)

[24] Dumontier, M., Villanueva-Rosales, N.: Towards pharmacogenomics knowledge discovery with the semantic web. Briefings in Bioinformatics 10(2), 153–163 (2009)

[25] Faddoul, J.: Reasoning algebraïcally with description logics. Ph.D. thesis, Concordia University (2011)

[26] Fang, Q., Zhao, Y., Yang, G., Zheng, W.: Scalable distributed ontology reasoning using DHT-based partitioning. In: The Semantic Web: 3rd Asian Semantic Web Conference, ASWC 2008. pp. 91–105 (2008)

[27] Fitting, M.: Introduction. In: D'Agostino, M. (ed.) Handbook of tableau methods, chap. 1. Kluwer Academic Publishers (1999)

[28] Ghosh, S.: Distributed systems: an algorithmic approach, vol. 13. CRC press (2006)

[29] Glimm, B., Horrocks, I., Motik, B., Shearer, R., Stoilos, G.: A novel approach to ontology classification. Web Semantics: Science, Services and Agents on the World Wide Web 14, 84–101 (2012)

[30] Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: A logical framework for modularity of ontologies. In: Proceedings International Joint Conference on Artificial Intelligence. pp. 298–304 (2007)

[31] Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: Modular reuse of ontologies: Theory and practice. Journal of Artificial Intelligence Research 31(1), 273–318 (2008)

[32] Grau, B.C., Parsia, B., Sirin, E., Kalyanpur, A.: Modularizing OWL ontologies. In: K-CAP 2005 Workshop on Ontology Management (2005)

[33] Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: Combining logic programs with description logic. In: Proceedings of the 12th international conference on World Wide Web. pp. 48–57 (2003)

[34] Haarslev, V., Möller, R.: Expressive ABox reasoning with number restrictions, role hierarchies, and transitively closed roles. In: Principles of Knowledge Representation and Reasoning-International Conference. pp. 273–284 (2000)

[35] Haarslev, V., Möller, R., Turhan, A.Y.: Exploiting pseudo models for TBox and ABox reasoning in expressive description logics. Automated Reasoning 2083/2001, 61–75 (2001)

[36] Helmbold, D.P., McDowell, C.E.: Modelling speedup (n) greater than n. IEEE Transactions on Parallel and Distributed Systems 1(2), 250–256 (1990)

[37] Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of semantic web technologies. Chapman & Hall/CRC (2009)

[38] Homola, M., Serafini, L.: Towards formal comparison of ontology linking, mapping and importing. In: Proceedings of the 2010 International Workshop on Description Logics. p. 291 (2010)

[39] Horridge, M., Bechhofer, S.: The OWL API: A Java API for OWL ontologies. Semantic Web 2(1), 11–21 (2011)

[40] Horrocks, I.: Optimising tableaux decision procedures for description logics. Ph.D. thesis, The University of Manchester (1997)

[41] Horrocks, I.: Using an expressive description logic: Fact or fiction? In: Proceedings of KR-98. pp. 636–649 (1998)

[42] Horrocks, I.: Implementation and optimization techniques. In: Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.) The description logic handbook: theory, implementation, and applications, chap. 9. Cambridge University Press (2003)

[43] Horrocks, I., Kutz, O., Sattler, U.: The even more irresistible SROIQ. In: Proceedings of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006). pp. 57–67 (2006)

[44] Horrocks, I., Patel-Schneider, P.F.: Optimizing description logic subsumption. Journal of Logic and Computation 9(3), 267 (1999)

[45] Horrocks, I., Sattler, U., Tobies, S.: Practical reasoning for expressive description logics. In: Proceedings of the 6th International Conference on Logic Programming and Automated Reasoning. pp. 161–180 (1999)

[46] Horrocks, I., Sattler, U., Tobies, S.: Reasoning with individuals for the description logic SHIQ. In: Automated deduction - CADE-17: 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 2000, Proceedings. p. 482 (2000)

[47] Horrocks, I., Sattler, U.: A tableau decision procedure for SHOIQ. Journal of Automated Reasoning 39, 249–276 (2007)

[48] Hudek, A.K., Weddell, G.: Binary absorption in tableaux-based reasoning for description logics. In: Proceedings of the 2006 International Workshop on Description Logics. vol. 189 (2006)

[49] Kazakov, Y.: Consequence-driven reasoning for horn SHIQ ontologies. In: Proceedings of the 21st international jont conference on Artifical intelligence. pp. 2040–2045 (2009)

[50] Kazakov, Y., Krötzsch, M., Simančík, F.: Concurrent classification of EL ontologies. In: Proceedings of the 10th International Semantic Web Conference (2011)

[51] Kazakov, Y., Krötzsch, M., Simančík, F.: The incredible ELK: From polynomial procedures to efficient reasoning with EL ontologies (2013), submitted to a journal

[52] Kshemkalyani, A.D., Singhal, M.: Distributed computing: principles, algorithms, and systems. Cambridge University Press (2008)

[53] Lavender, R.G., Schmidt, D.C.: Active object: an object behavioral pattern for concurrent programming. In: The Second Annual Conference on the Pattern Languages of Programs. pp. 483–499 (1995)

[54] Lea, D.: Concurrent programming in Java: design principles and patterns. Addison-Wesley Professional (2000)

[55] Liebig, T., Müller, F.: Parallelizing tableaux-based description logic reasoning. In: Proceedings of the 2007 OTM Confederated International Conference on the Move to Meaningful Internet Systems-Volume Part II. pp. 1135–1144 (2007)

[56] Lipkis, T.: A KL-ONE classifier. In: Proceedings of the 1981 KL-ONE Workshop. pp. 128–145 (1982)

[57] Meissner, A.: A simple parallel reasoning system for the ALC description logic. In: Computational Collective Intelligence: Semantic Web, Social Networks and Multiagent Systems (First International Conference, ICCCI 2009, Wroclaw, Poland, October 2009). pp. 413–424 (2009)

[58] Meissner, A., Brzykcy, G.: A parallel deduction for description logics with ALC language. Knowledge-Driven Computing 102, 149–164 (2008)

[59] Minsky, M.: A framework for representing knowledge. In: The Psychology of Computer Vision. McGraw-Hill (1975)

[60] Motik, B.: On the properties of metamodeling in OWL. Journal of Logic and Computation 17(4), 617 (2007)

[61] Motik, B., Shearer, R., Horrocks, I.: A hypertableau calculus for SHIQ. In: Calvanese, D., Franconi, E., Haarslev, V., Lembo, D., Motik, B., Tessaris, S., Turhan, A.Y. (eds.) Proceedings of the 2007 International Workshop on Description Logics. pp. 419–426. Brixen/Bressanone, Italy (June 2007)

[62] Motik, B., Shearer, R., Horrocks, I.: Optimized reasoning in description logics using hypertableaux. Automated Deduction–CADE-21 4603/2007, 67–83 (2007)

[63] Motik, B., Shearer, R., Horrocks, I.: Optimizing the nominal introduction rule in (hyper) tableau calculi. In: Baader, F., Lutz, C., Motik, B. (eds.) Proceedings of the 2008 International Workshop on Description Logics. CEUR Workshop Proceedings, vol. 353. Dresden, Germany (May 13–16 2008)

[64] Motik, B., Shearer, R., Horrocks, I.: Hypertableau reasoning for description logics. Journal of Artificial Intelligence Research 36, 165–228 (2009)

[65] Mutharaju, R., Maier, F., Hitzler, P.: A MapReduce algorithm for EL+. In: Proceedings of the 2010 International Workshop on Description Logics. p. 456 (2010)

[66] Nagashima, U., Hyugaji, S., Sekiguchi, S., Sato, M., Hosoya, H.: An experience with super-linear speedup achieved by parallel computing on a workstation cluster: Parallel calculation of density of states of large scale cyclic polyacenes. Parallel computing 21(9), 1491–1504 (1995)

[67] Nebel, B.: Reasoning and revision in hybrid representation systems, vol. 422. Springer-Verlag New York, Inc. (1990)

[68] Pérez-Urbina, H., Horrocks, I., Motik, B.: Efficient query answering for OWL 2. In: The Semantic Web-ISWC 2009. pp. 489–504 (2009)

[69] Quinn, M.J., Deo, N.: An upper bound for the speedup of parallel best-bound branch-and-bound algorithms. BIT Numerical Mathematics 26(1), 35–43 (1986)

[70] Schild, K.: A correspondence theory for Terminological Logics: Preliminary report. In: Proceedings of the 12th international joint conference on Artificial Intelligence-Volume 1. pp. 466–471 (1991)

[71] Schmidt-Schauß, M., Smolka, G.: Attributive concept descriptions with complements. Artificial Intelligence 1(48), 1–26 (1991)

[72] Shearer, R., Horrocks, I., Motik, B.: Exploiting partial information in taxonomy construction. In: Grau, B.G., Horrocks, I., Motik, B., Sattler, U. (eds.) Proceedings of the 2009 International Workshop on Description Logics. CEUR Workshop Proceedings, vol. 477. Oxford, UK (July 27–30 2009)

[73] Shearer, R., Motik, B., Horrocks, I.: Hermit: A highly-efficient OWL reasoner. In: Ruttenberg, A., Sattler, U., Dolbear, C. (eds.) Proceedings of the 5th Int.

Workshop on OWL: Experiences and Directions (OWLED 2008 EU). Karlsruhe, Germany (October 26–27 2008)

[74] Simančík, F., Kazakov, Y., Horrocks, I.: Consequence-based reasoning beyond Horn ontologies. In: Proceedings of the 22nd Int. Joint Conf. on Artificial Intelligence (International Joint Conference on Artificial Intelligence 2011) (2011)

[75] Smolka, G.: A feature logic with subsorts. Tech. rep., LILOG Report 33, IWBS, IBM Deutschland (1992)

[76] Soma, R., Prasanna, V.: Parallel inferencing for OWL knowledge bases. In: The 37th International Conference on Parallel Processing (ICPP-08). pp. 75–82 (2008)

[77] Sottile, M.J., Mattson, T.G., Rasmussen, C.E.: Introduction to concurrency in programming languages. Chapman & Hall/CRC (2010)

[78] Staab, S., Studer, R.: Handbook on ontologies. Springer, second edn. (2009)

[79] Tarski, A.: Logic, semantics, metamathematics: papers from 1923 to 1938. Hackett Publishing (1983)

[80] Toub, S.: Patterns of Parallel Programming. Microsoft Corporation (2010)

[81] Tsarkov, D., Horrocks, I.: Efficient reasoning with range and domain constraints. In: Proceedings of the 2004 International Workshop on Description Logics. p. 41 (2004)

[82] Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. Automated Reasoning 4130, 292–297 (2006)

[83] Tsarkov, D., Horrocks, I., Patel-Schneider, P.F.: Optimizing terminological reasoning for expressive description logics. Journal of Automated Reasoning 39(3), 277–316 (2007)

[84] Wu, J., Haarslev, V.: Planning of axiom absorptions. In: Proceedings of the 2008 International Workshop on Description Logics (2008)

[85] Wu, K., Haarslev, V.: A parallel reasoner for the description logic ALC. In: Proceedings of the 2012 International Workshop on Description Logics (2012)

[86] Wu, K., Haarslev, V.: Exploring parallelization of conjunctive branches in tableau-based description logic reasoning. In: Proceedings of the 2013 International Workshop on Description Logics (2013)

[87] Wu, K., Haarslev, V.: Parallel OWL reasoning: Merge classification. In: The Third Joint International Semantic Technology Conference (2013)

[88] Zolin, E.: Complexity of reasoning in Description Logics (8 2013), `http://www.cs.man.ac.uk/~ezolin/dl/`

Nomenclature

ABox A DL knowledge base component, which contains assertional descriptions on individuals (i.e. instances).

AI Artificial Intelligence

$\mathcal{AL}$ A basic DL language allowing atomic concept negation, concept intersection, universal restriction, and limited existential quantification (no concept union).

$\mathcal{ALC}$ A DL language extending $\mathcal{AL}$ with full concept negation, full existential quantification, and concept union.

$\mathcal{ALCH}(\mathcal{D})$ A DL language extending $\mathcal{ALCH}$ with concrete data type.

$\mathcal{ALCH}$ A DL language extending $\mathcal{ALC}$ with role hierarchy.

$\mathcal{ALCN}$ A DL language extending $\mathcal{ALC}$ with unqualified number restriction.

$\mathcal{ALE}$ A DL language extending $\mathcal{AL}$ with full existential quantification.

$\mathcal{ALE}+$ A DL language extending $\mathcal{ALE}$ with role transitivity.

DAG directed acyclic graph

DHT distributed hash table

DL Description Logic

DLP Description Logic Programs: a knowledge representation that is used for the inter-operation between rule-based reasoning and DL.

$\mathcal{EL}$ A DL language allowing concept intersection and full existential quantification.

$\mathcal{EL}+$ A DL language extending $\mathcal{EL}$ with role transitivity.

GCI general concept inclusion axioms of the form $C \sqsubseteq D$ in which C is not necessarily an *atomic* concept name.

GUID globally unique identifier

KB knowledge base

KR Knowledge Representation and Reasoning

NNF negation normal form: Negation occurs only in front of atomic concept names.

ORE OWL Reasoner Evaluation Workshop

OWL Web Ontology Language

RBox A DL knowledge base component, which contains a set of roles (i.e. properties) descriptions, such as hierarchical or functional descriptions.

RDF Resource Description Framework

RDFS Resource Description Framework Schema

RIF Rule Interchange Format

$\mathcal{S}$ An abbreviation for $\mathcal{ALC}$ with role transitivity.

$\mathcal{SHIF}$ A DL language extending $\mathcal{S}$ with role hierarchy, role inverse, and functional role.

$\mathcal{SHIQ}$ A DL language extending $\mathcal{S}$ with role hierarchy, role inverse, and qualified cardinality restriction.

$\mathcal{SHOIN}$ A DL language extending $\mathcal{S}$ with role hierarchy, nominal, role inverse, and unqualified number restriction.

SPARQL SPARQL Protocol and RDF Query Language

$\mathcal{SROIQ}$ A DL language extending $\mathcal{S}$ with nominal, role inverse, qualified cardinality restriction, and a set of role descriptions (role intersection, role reflexivity, and so on).

TBox A DL knowledge base component, which contains terminological descriptions on concepts (i.e. classes).

URI Uniform Resource Identifier

W3C World Wide Web Consortium

Appendices

Appendix A

Tableau

A.1 Rules

Table A.1 lists the tableau rules used in $\mathcal{SHIQ}$ [45].

$T = (\mathbf{S}, \mathcal{L}, \mathcal{E})$ is a tableau for a $\mathcal{SHIQ}$-concept D in NNF, with $clos(D)$ as the smallest set of concepts that contains D and is closed under sub-concepts and $\neg D$'s NNF, $\mathcal{R}^+$ a role hierarchy, and $\mathbf{R}_D$ the set of roles occurring in D and $\mathcal{R}^+$ together with their inverses: $\mathbf{S}$ is a set of individuals, $\mathcal{L} : \mathbf{S} \rightarrow 2^{clos(D)}, \mathcal{E} : \mathbf{R}_D \rightarrow 2^{\mathbf{S}\times\mathbf{S}}$, $D \in \mathcal{L}(s)$, and $s \in \mathbf{S}$.

In Table A.1, given a tableau $T = (\mathbf{S}, \mathcal{L}, \mathcal{E})$: $\bowtie$ is a placeholder for $\leq$ and $\geq$; $S^T(s, C) = \{t \in \mathbf{S} \mid \langle s, t\rangle \in \mathcal{E}(S) \wedge C \in \mathcal{L}(t)\}$.

A node x of a tableau tree for a concept D is labelled with a set $\mathcal{L}(x) \subseteq clos(D)$, an edge $\langle x, y\rangle$ is labelled with a set $\mathcal{L}(\langle x, y\rangle)$ of roles occurring in $clos(D)$, and explicit inequalities between nodes are recorded in a symmetric binary relation $\neq$.

A node y is an R-successor of a node x iff y is a successor of x and $S \in \mathcal{L}(\langle x, y\rangle)$ for some S with $S \sqsubseteq R$. A node y is an R-neighbor of x iff y is an R-successor of x, or if x is an $Inv(R)$-successor of y.

A node x is blocked iff it is *directly* or *indirectly* blocked. A node is *directly* blocked iff none of its ancestors are blocked, and it has ancestors x', y, and y' such

name	**rule**
$\sqcap$**-rule**	If *(i)* $C \sqcap D \in \mathcal{L}(x)$, x is not indirectly blocked, and *(ii)* $\{C, D\} \not\subseteq \mathcal{L}(x)$, then $\mathcal{L}(x) \leftarrow \mathcal{L}(x) \cup \{C, D\}$.
$\sqcup$**-rule**	If *(i)* $C \sqcup D \in \mathcal{L}(x)$, x is not indirectly blocked, and *(ii)* $\{C, D\} \cap \mathcal{L}(x) = \emptyset$, then $\mathcal{L}(x) \leftarrow \mathcal{L}(x) \cup \{E\}$ for some $E \in \{C, D\}$.
$\exists$**-rule**	If *(i)* $\exists S.C \in \mathcal{L}(x)$, x is not blocked, and *(ii)* x has no S-neighbor y with $C \in \mathcal{L}(y)$, then create a new node y with $\mathcal{L}(\langle x, y\rangle) \leftarrow \{S\}$ and $\mathcal{L}(y) \leftarrow \{C\}$.
$\forall$**-rule**	If *(i)* $\forall S.C \in \mathcal{L}(x)$, x is not indirectly blocked, and *(ii)* there is an S-neighbor y of x with $C \notin \mathcal{L}(y)$, then $\mathcal{L}(y) \leftarrow \mathcal{L}(y) \cup \{C\}$.
$\forall_+$**-rule**	If *(i)* $\forall S.C \in \mathcal{L}(x)$, x is not indirectly blocked, *(ii)* there is some R with $Trans(R)$ and $R \sqsubseteq S$, and *(iii)* there is an R-neighbor y of x with $\forall R.C \notin \mathcal{L}(y)$, then $\mathcal{L}(y) \leftarrow \mathcal{L}(y) \cup \{\forall R.C\}$.
choose-**rule**	If *(i)* $(\bowtie n S.C) \in \mathcal{L}(x)$, x is not indirectly blocked, and *(ii)* there is an S-neighbor y of x with $\{C, nnf(\neg C)\} \cap \mathcal{L}(y) = \emptyset$, then $\mathcal{L}(y) \leftarrow \mathcal{L}(y) \cup E$ for some $E \in \{C, nnf(\neg C)\}$.
$\geq$**-rule**	If *(i)* $(\geq n S.C) \in \mathcal{L}(x)$, x is not blocked, and *(ii)* there are not n S-neighbors $y_1, \ldots, y_n$ of x with $C \in \mathcal{L}(y_i)$ and $y_i \neq y_j$ for $1 \leq i < j \leq n$, then create n new nodes $y_1, \ldots, y_n$ with $\mathcal{L}(\langle x, y_i\rangle) \leftarrow \{S\}$, $\mathcal{L}(y_i) \leftarrow \{C\}$, and $y_i \neq y_j$ for $1 \leq i < j \leq n$.
$\leq$**-rule**	If *(i)* $(\leq n S.C) \in \mathcal{L}(x)$, x is not indirectly blocked, and *(ii)* $\left\lvert S^T(x, C)\right\rvert > n$ and there are two S-neighbors y, z of x with $C \in \mathcal{L}(y)$, $C \in \mathcal{L}(z)$, y is not an ancestor of x, and not $y \neq z$, then *(i)* $\mathcal{L}(z) \leftarrow \mathcal{L}(z) \cup \mathcal{L}(y)$, *(ii)* if z is an ancestor of x, then $\mathcal{L}(\langle z, x\rangle) \leftarrow \mathcal{L}(\langle z, x\rangle) \cup Inv(\mathcal{L}(\langle x, y\rangle))$, else $\mathcal{L}(\langle x, z\rangle) \leftarrow \mathcal{L}(\langle x, z\rangle) \cup \mathcal{L}(\langle x, y\rangle)$, *(iii)* $\mathcal{L}(\langle x, y\rangle) \leftarrow \emptyset$, and *(iv)* set $u \neq z$ for all u with $u \neq y$.

Table A.1: Tableau expansion rules for deciding satisfiability of an $\mathcal{SHIQ}$ concept.

that: *(i)* x is a successor of x' and y is a successor of y'; *(ii)* $\mathcal{L}(x) = \mathcal{L}(y)$ and $\mathcal{L}(x') = \mathcal{L}(y')$; and *(iii)* $\mathcal{L}(\langle x', x\rangle) = \mathcal{L}(\langle y', y\rangle)$. A node y is *indirectly* blocked iff one of its ancestors is blocked, or it is a successor of a node x and $\mathcal{L}(\langle x, y\rangle) = \emptyset$.

See [45] and [47] for a full reference on $\mathcal{SHIQ}$ tableau expansion rules.

Appendix B

Pseudocode

B.1 Merge Classification

```
100 (define merge-top
101  (λ (C D)
102    (if (subs? D C)
103     (let ([P (merge-top-search C D)])
104          (taxonomy-add C P)
105          (for-each (λ (E)
106                      (for* ([c (get-children C)]
107                             [e (get-children E)])
108                        (merge-top c e)))
109                    P))
110     (for-each (λ (c)
111                 (merge-top c D))
112               (get-children C)))))
```

Listing B.1: Lisp/Scheme/Racket-style pseudocode: merge-top.

```
100 (define merge-top-search
101  (λ (C D)
102   (let ([box (filter (λ (d)
103                        (subs? d C))
104                      (get-children D))])
105     (if (empty? box)
106         (cons D (quote ()))
107         (map (λ (d)
108                (merge-top-search C d))
109              box)))))
```

Listing B.2: Lisp/Scheme/Racket-style pseudocode: merge-top-search.