DESIGN AND EVALUATION OF ALGORITHMS FOR PARALLEL CLASSIFICATION OF ONTOLOGIES

MINA ASLANI

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

APRIL 2013

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Mina Aslani**
Entitled: **Design and Evaluation of Algorithms for Parallel Classification of Ontologies**

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.

Signed by the final examining commitee:

_____ Chair

_____ External Examiner
Dr. Bruce Spencer

_____ Examiner
Dr. Hovhannes Harutyunyan

_____ Examiner
Dr. Ferhat Khendek

_____ Examiner
Dr. Nematollaah Shiri

_____ Supervisor
Dr. Volker Haarslev

Approved _____
Chair of Department or Graduate Program Director

_____ 20 _____ _____

Dr. Robin Drew, Dean
Faculty of Engineering and Computer Science

Abstract

Design and Evaluation of Algorithms for Parallel Classification of Ontologies

Mina Aslani, Ph.D.

Concordia University, 2013

Description Logics are a family of knowledge representation formalisms with formal semantics. In recent years, DLs have influenced the design and standardization of the Web Ontology Language OWL. The acceptance of OWL as a web standard has promoted the widespread utilization of DL ontologies on the web. One of the most frequently used inference services of description logic reasoners classifies all named classes of OWL ontologies into a subsumption hierarchy. Due to emerging OWL ontologies from the web community consisting of up to hundreds of thousand of named classes and the increasing availability of multi-processor and multi- or many-core computers, the need for parallelizing description logic inference services to achieve a better scalability is expected.

The contribution of this thesis has two aspects. On a theoretical level, it first presents algorithms to construct a TBox in parallel, which are independent of

a particular DL logic, however they sacrifice completeness. Then, a sound and complete algorithm for TBox classification in parallel is presented. In this algorithm all the subsumption relationships between concepts of a partition assigned to a single thread are found correctly, in other words, correctness of the TBox subsumption hierarchy is guaranteed. Thereafter, we provide an extension of the sound and complete algorithm which is used to handle TBox classification concurrently and more efficiently. This thesis also describes an optimization technique suitable for better partitioning the list of concepts to be inserted into the TBox.

On a practical level, a running prototype, Parallel TBox Classifier was implemented for each generation of the classifier based on the above theoretical foundations, respectively. The Parallel TBox Classifier is used to evaluate the practical merit of the proposed algorithms as well as the effectiveness of the designed optimizations against existing state-of-the-art benchmarks. The empirical results illustrate that Parallel TBox Classifier outperforms the Sequential TBox Classifier on real world ontologies with a linear or superlinear speedup factor. Parallel TBox Classifier can form a basis to develop more efficient parallel classification techniques for real world ontologies with different sizes and DL complexities.

Acknowledgments

I wish to express my deepest gratitude to my supervisor, Dr. Volker Haarslev, whose patience, motivation, continuous support, and immense knowledge encouraged me throughout the years of my study. His confidence in my ability empowered me to pursue my research goals, and overcome many obstacles. This thesis would have not been possible without him.

I would like to thank Dmitry Tsarkov from University of Manchester and Thomas Schneider from University of Bremen for their help during the final stages of this PhD.

I am grateful to my friends, classmates and colleagues for their emotional support, and friendship when I needed the most.

I am always indebted to my parents and my brother for being there for me and inspiring me to follow my dreams.

I dedicate this thesis to the love of my life, my husband.

To

My Beautiful Daughter

Kiana

Contents

List of Figures

List of Tables

List of Algorithms

LIST OF ABBREVIATIONS

ABox Assertion Box (assertions about individuals)

AI Artificial Intelligence

ALC Attributive Concept Language with Complements

AL Attributive Language

DAG Directed Acyclic Graphs

DL Description Logic

FL Frame based description language

FOL First Order Logic

GALEN Generalized Architecture for Languages, Encyclopedia and Nomenclatures

GCI General Concept Inclusion

GO Gene Ontology

KR Knowledge Representation

OWL2 Web Ontology Language 2

OWL The Web Ontology Language

RACER Renamed Abox and Concept Expression Reasoner

RDF Resource Description Framework

SOTA State-of-the-art

TBox Terminological Box (axioms about class definitions)

UMLS The Unified Medical Language System

W3C The World Wide Web Consortium

WWW World Wide Web

XML Extensible Markup Language

Chapter 1

Introduction

Ontologies are the knowledge infrastructures of Description Logics and many intelligent systems. Ontologies require that an extensive knowledge about the world be represented and stored in a knowledge base. Among the things that need to be represented are: objects, properties, and relations between objects. A complete representation of ”what exists” in a given domain forms an ontology. In order to support the vision of the Description Logics, ontology classification needs to be highly scalable and efficient.

Description logic (DL) is a family of first-order logic formalisms allowing the representation of knowledge in the form of ”concepts” (class, unary predicate), ”roles” (object property, binary predicate), and ”individuals” (class instance). The ability of DL languages to define concepts and relationships between concepts in a systematic and formal manner, makes them ideal to capture the complex relationships and semantics that are often part of many domains (e.g., medical domain). DL is becoming very popular in Knowledge Representation and modelling as it provides the logical foundation for the Web Ontology Language (OWL), defined

by the World Wide Web Consortium (W3C) as a standard for representing semantic links and knowledge on the Semantic Web.

Due to the recent popularity of OWL ontologies in the web one can observe a trend toward the development of large OWL-DL ontologies. For instance, well known examples from the bioinformatics or medical community are SNOMED, UMLS, GALEN, and FMA. Some (versions) of these ontologies consist of more than hundreds of thousands of named concepts and have become challenging even for the most advanced and optimized description logic (DL) reasoners. Although specialized DL reasoners for certain sublogics (e.g., CEL for EL++) and OWL-DL reasoners such as FaCT++, Pellet, HermiT, or RacerPro could demonstrate impressive speed enhancements due to newly designed optimization techniques, one can expect the need for parallelizing description logic inference services in the near future in order to achieve a web-like scalability where we have to consider hundreds of thousands of concepts that challenges subsumptions tests.

Parallel algorithms for description logic reasoning were first explored in the FLEX system [9] where various distributed message-passing schemes for rule execution were evaluated. The reported results seemed to be promising but the research suffered from severe limitations due to the hardware available for experiments at that time. Another approach on parallelizing description logic reasoning [30] reported promising results using multi-core/processor hardware, where the parallel treatment of disjunctions and individual merging (due to number restrictions) is explored. In [39] an approach on distributed reasoning for $\mathcal{ALCHIQ}$ is presented that is based on resolution techniques but does not address optimizations for TBox (set of axioms) classification. There also exists work on parallel distributed RDF inferencing (e.g., [49]) and parallel reasoning in first-order the-

orem proving but due to completely different proof techniques (resolution versus tableau) and reasoning architectures this is not considered as relevant here.

Our research is strongly motivated by recent trends in computer hardware where processors feature multi-cores (2 to 8 cores) or many-cores (tens or even hundreds of cores). These processors promise significant speed-ups for algorithms exploiting so-called thread-level parallelism. This type of parallelism is very promising for DL reasoning algorithms that can be executed in parallel but might share common data structures (e.g. parallelism in classification of TBoxes, ABox realization or query answering).

Although multi-processor, multi-core systems have become widespread but the vast majority of OWL reasoners can process ontologies only sequentially, therefore only one core is utilized. Imagine the most powerful DL reasoner, using one core reasoning for 600 minutes. If we use 10 cores, with linear scalability to the number of utilized cores, then the reasoning will take only 60 minutes. Hence, the wall-clock time is significantly reduced.

The problem of the classification of large ontologies has existed for the past decade in the field of knowledge representation and artificial intelligence and despite some efforts in this area, there are no widely accepted algorithms available and also there is not any clear sign of progress in the attempts to solve the problem of concurrent ontology classification. These observations motivated us that there is a need to direct our attention to propose a set of information able to reveal and solve these fundamental problems.

1.1 Thesis Objectives

The research presented in this thesis is focused on designing a parallel classification approach for TBoxes as well as implementing it. The main objectives of adopting such a classification approach can be described as follows:

- Improving computationally expensive TBox classification time: The classification approach must ensure that the classification time using parallel computation is less than the sequential case.

- Correctness: Show that a classification procedure working in parallel ensures *soundness and completeness*. Soundness in the sense that every ”yes” answer for an inference test is a valid answer. Completeness in the sense that every ”no” answer for an inference test is a valid answer.

- Use of algorithms designed for sequential execution in parallel classification: Examine that Enhanced Traversal optimization algorithms designed for sequential execution, are also efficient for parallel classification, if extended correspondingly.

- Usability: The parallel classification procedure outperforms the sequential scenario on real world ontologies. The usability of the approach with state of the art ontologies will be assessed by an empirical analysis.

- Scalability: The classification procedure is able to be used with real world benchmarks including ontologies of different complexity and size, particularly large size ontologies with hundreds of thousands of concepts. The prototype needs to be able to tackle such a complexity.

1.2 Thesis Contributions

The work presented in this thesis is of interest to the DL community and should be of value to designers and developers of TBox classifiers as well as DL reasoners, who will be able to work into or utilize the classification approach together with their implemented procedures. The main contributions of this thesis are as follows:

- The first contribution is the design of the first parallel TBox classification algorithm. This algorithm which is independent of a particular DL logic, is represented in three generations :
  - The first generation of the algorithm sacrifices completeness and was published in [1]. The algorithm which is the first addressing parallel TBox classification will be covered in Section 5.1.1
  - The second generation of the algorithm is sound and complete. This algorithm is the first sound and complete parallel TBox classification algorithm and was proposed and published in [2] and [3]. The thorough explanation of the design and corresponding algorithm will be elaborated in Section 5.1.2.
  - The third generation of the algorithm is also sound and complete. This algorithm is an optimized concurrent extension of the second generation. The algorithm was proposed and published in [4], and will be explained in Section 5.1.3.
- The second contribution relies on utilizing optimization algorithms in parallel which were mainly designed for sequential execution.

- The third contribution is the design and implementation of a prototype for each generation of parallel TBox classifier. The architecture of the prototype is described through Section 5.2.

- The last contribution relies on the performance of the proposed algorithms. As will be explained in Chapter 6, the empirical evaluations show that the first generation of the prototype as well as the second generation are limited to classify ontologies with only 10000 concepts. However, the third generation is able to construct the ontologies of more than 379000 concepts in parallel. Therefore, for the last generation, the scalability of more than one order of magnitude is achieved. The assessment of the empirical results also supports a speedup factor that is linear to the number of utilized processors/cores.

1.3 Thesis Outline

The rest of this thesis can be outlined as follows:

- Chapter 2 introduces a formal definition of Description Logics, its syntax and semantics with a main focus on the $\mathcal{AL}$ family. It also gives an overview of DL inference services as well as state-of-the-art DL reasoners.

- Chapter 3 gives a background review of research closely related to this thesis.

- Chapter 4 presents the parallel processing and the programming model in MapReduce as well as an overview of the research work which are loosely related to our work.

- Chapter 5 consists of two main sections. In the first section, it discusses an algorithm for parallel classification of TBoxes as well as its evolution through three generations. In the second section, it presents the design of a prototype for parallel TBox classifier and its underlying architecture.

- Chapter 6 provides the evaluation of the prototype, and also explains the scalability of the Parallel TBox Classifier by using ontologies of different complexity and size.

- Chapter 7 concludes this thesis with highlighting the scientific contributions as well as a list of open problems for future work.

Chapter 2

Preliminaries

This chapter introduces preliminary information relevant to the work presented in this thesis. Section 2.1 explains the difference between Logic based system and Non-logic based system. Section 2.2 describes what description logics is. Section 2.3 presents basic definitions, syntax, semantics of Description Logics as well as the tableau algorithms as the most widely used reasoning procedures adopted by most DL reasoners. In this section, there will also be explanation of DL inferences services as well as an overview of the state-of-the-art DL reasoners.

2.1 Logic based versus Non-logic based systems

The field of knowledge representation started to gain popularity in 1970s and it was divided into two main categories: logic-based formalism, which used predicate calculus for their unambiguous representation, and non-logic based formalism. Since first order logic provides very powerful and general machinery, logic-based systems were more general-purpose from the very start. In logic-based sys-

tems the representation language is often a variant of first-order predicate calculus and reasoning is the result of verifying logical consequences. On the other hand, non logic-based systems were usually task specific, yet they were used successfully as general purpose tools. In these approaches, graphical interfaces are used for representing knowledge. Semantic networks and frames fall in this category of representation [32].

Initially, most of the systems were developed on the network based approach and were used in many applications. As more and more systems were developed, the need for a technique was felt which can precisely characterize the meaning of theses structures. This need resulted in the introduction of Description Logics. In Description Logics, precise semantic of the structures can be given by defining the language for the elements of the structures and by providing the interpretation of the strings of the language [32].

It has been accepted that representing knowledge is one of the most important and basic tasks in any knowledge base reasoning system, however the importance of the reasoning system, which is on top of the represented knowledge can not be denied. There are two important features which have to be considered while designing a reasoning system: Sound and complete reasoning. There exists a trade-off between the soundness and the completeness of a reasoning system.

2.2 Why Description Logic?

Description Logics is the most recent name for a family of knowledge representation formalisms unifying and giving a logical basis to the well-known traditions of frame-based systems, semantic networks and KL-ONE-like [38] lan-

guages, object-oriented representations, semantic data models and type systems. It also expresses knowledge about concepts, concept hierarchies, roles and individuals. Description logic systems emphasize the use of classification and subsumption reasoning as their primary mode of inference. Description Logics are descended from so-called ”structured inheritance networks” [Brachman, 1977b; 1978]. Loom [31] and Classic [10], [12], [11] were two early examples of knowledge representation systems that implemented description logics. In other words, description logics is a formalism that represents the knowledge of an application domain (”world”) by first defining the relevant concepts of the domain (its taxonomy), and then using those concepts to specify properties of the objects and individuals in the domain. As its name indicates, they are equipped with formal logic-based semantics. Another distinguished feature is reasoning which is being used as central service: reasoning is to infer implicit knowledge from the knowledge explicitly contained in the knowledge base. The classification of concepts into terminologies means to compute the superconcept/subconcept (parent/child) relationship and is called subsumption relationship in DL. The classification of individuals determines whether a given individual is an instance of a concept. There are two features of DL that are not shared by most data description formalisms: DL does not assume the Unique Name Assumption (UNA) or the Close World Assumption (CWA). UNA means that individuals with different names are assumed to be distinct. Not having CWA, or in other words having Open World Assumption (OWA) means that absence of knowledge in the KB does not imply that it is false. The OWA assumes incomplete information, which will be explained in Section 2.4.

2.3 Basic Definitions in Description Logics

A Description Logic system sets up knowledge bases to do reasoning and manipulation of its content. Figure 2.1 shows the architecture of such systems.

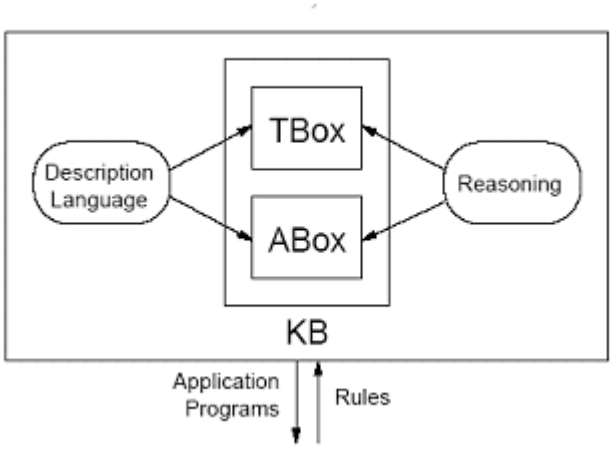


Figure 2.1: Architecture of Knowledge Representation System based on Description Logic [7].

As it is shown in Figure 2.1, a Knowledge Base (KB) consists of a TBox and an ABox. A finite set of axioms is called a TBox or terminology which introduces the vocabulary of the application domain. On the other hand, an ABox is a collection of assertional axioms, which contains assertions about individuals.

Definition 2.3.1 (Interpretation) *An interpretation is a pair* $\mathcal{I} = (\Delta^I, .^I)$, Δ^I *is a non-empty set called the domain of interpretation, and* $.^I$ *is the interpretation function. The interpretation function maps each atomic concept* $A \in N_C$ *to a subset of* Δ^I, *each atomic role* $R \in N_R$ *to a subset of* Δ^I *x* Δ^I, *and each individual a* inN_I *to an element of* Δ^I.

Definition 2.3.2 (Concept Inclusion Axiom) *TBox axioms have an expression of the form:*

- *Definitions*
  - *$C \sqsubseteq D$ which is called Concept subsumption axiom (Concept inclusion axiom), or*
  - *$C \equiv D$ which is called Concept equivalence axiom*

  Where C is a concept name.
- *General Concept Inclusion axioms (GCIs)*
  - *$C \sqsubseteq D$ where C is an arbitrary concept.*

2.3.1 TBox - Terminologies

Terminologies (Terminological Axioms) make statements about how concepts or roles are related to each other [7]. We can define our TBox using axioms in the form shown in Table 2.1. The first type of axioms is called inclusions and the second one equalities, where C and D are concepts, and R and S are roles.

$C \sqsubseteq D$	$R \sqsubseteq S$
$C \equiv D$	$R \equiv S$

Table 2.1: Different forms of axioms.

Now we can say that an interpretation $\mathcal{I}$ satisfies an inclusion $C \sqsubseteq D$ if $C^I \subseteq D^I$ and it will satisfy an equality $C \equiv D$ if $C^I \equiv D^I$. If $\mathcal{T}$ is a set of axioms, then an interpretation $\mathcal{I}$ satisfy $\mathcal{T}$ iff it satisfies all the axioms in $\mathcal{T}$. If $\mathcal{I}$ satisfies an axiom/set of axioms, then $\mathcal{I}$ is a model for that axiom/set of axioms. Two axioms/set of axioms are equivalent if they have the same models.

As mentioned earlier, equality axiom whose left hand side is an atomic concept, is called a *definition*. For example, we can define a concept (symbolic name) Mother as below:

Mother $\equiv$ Woman $\sqcap \exists$ hasChild.Person

2.3.2 ABox - Assertions about individuals

ABox is a set of assertions. In the ABox we define individuals by giving them names and also defining their properties. In an ABox, we have two kinds of assertions:

- *Concept assertion* which asserts an individual for a concept, and is the form of *a*:C.
- *Role assertion* which we introduce a role for two individuals, and is the form of (*a*,*b*):R.

Considering MARY, PETER, and PAUL as individual names, if one wants to assert concept PETER as a father in the ABox, the assertion *PETER*:Father can be used, and (*MARY*,*PAUL*):hasChild which is a role assertion means that MARY has a child named PAUL.

2.4 Open-world semantics vs. closed-world semantics

In order to explain open-world and closed-world semantics, databases on one hand should be compared to DL on the other hand. The schema of a database can be compared to a TBox and the database instances (data) to an ABox although the semantics of ABoxes differ from the semantics of data in databases. While the

data in a DB represents one interpretation, namely the relations in the schema are interpreted by tuples in the data, an ABox can represent several interpretations, namely all its models. In a simplified view, an ABox can be seen as an instance of a relational database with only unary or binary relations. However, contrary to the closed-world semantics of classical databases, the semantics of ABoxes is an open-world semantics, since normally knowledge representation systems are applied in situations where one cannot assume that the knowledge in the KB is complete. Moreover, the TBox imposes semantic relationships between the concepts and roles in the ABox that do not have counterparts in database semantics [36]. An absence of information in databases usually means negative information, however an absence of information in an ABox is interpreted as lack of information. For example, if the only assertion about MARY is (*MARY*,*PAUL*):hasChild, then in a database it can be deduced that PAUL is the only child of MARY. In an ABox, it can be understood that MARY has a child who is named PAUL. However, the ABox might have several models, some in which PAUL is the only child, and others in which he has brothers or sisters. If it is asserted that PAUL is a male, one cannot deduce from this alone that all MARY's children all male. The only way to state that PAUL is the only child of MARY is by adding the assertion ≤ 1 hasChild.*PAUL*. Therefore, while the information in databases is always considered as complete, the information in ABoxes is incomplete. The semantics of ABoxes is characterized as an "open-world" semantics, while the traditional semantics of databases is characterized as a "closed-world" semantics.

2.5 Description Languages

Description Languages are distinguished by the constructors they provide. In this section, we will discuss about the family of $\mathcal{AL}$-Languages (Attributive Languages). $\mathcal{AL}$ is a minimal language of practical interest and its sequels are just extensions of $\mathcal{AL}$ languages.

2.5.1 Basic Description Language $\mathcal{AL}$

In $\mathcal{AL}$, negation can be applied only to atomic concepts and only the top concept can be used over the existential quantification. The following are examples of axioms expressed in $\mathcal{AL}$.

Person $\sqcap$ Female

Person $\sqcap \neg$Female

Person $\sqcap \exists$hasChild.$\top$

Person $\sqcap \forall$hasChild.Female

Person $\sqcap \forall$hasChild.$\bot$

The sub-language of $\mathcal{AL}$, which disallows limited negation, is $\mathcal{FL}^-$ and the one, which disallows limited existential quantification, is named $\mathcal{FL}^0$. In Figure 2.2, the syntax of $\mathcal{AL}$ is presented.

$\mathcal{AL}$-concept $\rightarrow$	A $\mid$	(atomic concept)
	$\top$ $\mid$	(universal concept)
	$\bot$ $\mid$	(bottom concept)
	$\neg$ A $\mid$	(atomic negation)
	C $\sqcap$ D $\mid$	(intersection)
	$\forall$ R.C $\mid$	(value restriction)
	$\exists$ R.$\top$ $\mid$	(limited existential quantification)

Figure 2.2: $\mathcal{AL}$ Syntax [7].

As described earlier, we can extend the $\mathcal{AL}$ language by adding more constructs.

2.5.2 The family of $\mathcal{AL}$-Languages

We obtain more expressive languages if we add further constructors to $\mathcal{AL}$ [7]. And as it will be shown, various letters forming names of DLs indicate other extension, e.g. [27]:

- $\mathcal{H}$ for role hierarchy (hasDaughter $\sqsubseteq$ hasChild)
- $\mathcal{I}$ for inverse roles (isChildOf $\equiv$ hasChild^{-})
- $\mathcal{N}$ for number restrictions (of form $\leq$nR, $\geq$nR)
- $\mathcal{O}$ for nominals/singleton classes ($\{$Canada$\}$)
- $\mathcal{Q}$ for qualified number restrictions (of form $\leq$n R.C, $\geq$n R.C e.g. $\leq$2 hasChild.Doctor)
- $\mathcal{S}$ often used for $\mathcal{ALC}$ with transitive roles (R^{+})

Therefore, using this scheme we can introduce the $\mathcal{SHIQ}$ language as $\mathcal{ALC}$ + transitive roles + role hierarchy + inverse roles + qualified number restriction. An example for $\mathcal{SHIQ}$ follows:

”Individuals all of whose children are human and have 2 parents who are human”
Human $\sqsubseteq$ $\forall$hasChild.Human $\sqcap$ =2 hasParent.Human

”HumanParent is a human who is a parent too”

HumanParent $\equiv$ Human $\sqcap$ Parent

”hasChild is the inverse of hasParent”

hasChild $\equiv$ hasParent^{-}

if (MARY,PAUL):hasChild then

(PAUL,MARY):Parent

2.5.3 Description Language $\mathcal{ALC}$

$\mathcal{ALC}$, is the smallest propositionally closed DL, which includes $\mathcal{AL}$. In $\mathcal{ALC}$, negation and existential quantification are not limited, concepts are constructed using ($\sqcap$, $\sqcup$, $\neg$), existential value restriction ($\exists$), and universal restriction($\forall$).

$\mathcal{ALC}$-concept $\rightarrow$	A |	(atomic concept)
	$\top$ |	(universal concept)
	$\bot$ |	(bottom concept)
	C $\sqcap$ D |	(intersection)
	C $\sqcup$ D |	(disjunction)
	$\neg$ C |	(negation)
	$\forall$ R.C |	(value restriction)
	$\exists$ R.C |	(existential quantification)

Figure 2.3: $\mathcal{ALC}$ Syntax [27].

Let N_C, N_R, and N_I be non-empty and pair-wise disjoint sets of concept names, role names, and individual names respectively. A is used to denote an atomic concept (A $\in N_C$), and R an atomic role (R $\in N_R$). Figure 2.3 expresses the syntax of the $\mathcal{ALC}$ language, where C, D are $\mathcal{ALC}$-concepts and $\top$ and $\bot$ are used to abbreviate (C $\sqcup$ $\neg$ C) and (C $\sqcap$ $\neg$ C) respectively.

2.5.4 A Tableau Algorithm for $\mathcal{ALC}$

Tableau algorithms are used to test concept satisfiability (consistency). Given an $\mathcal{ALC}$-concept description C_0, a tableau algorithm tries to find a finite interpretation to satisfy C_0 [8]. It will be convenient to assume that all concept descriptions are in NNF (Negation Normal Form).

Definition 2.5.1 (Negation Normal Form (NNF)) *A concept expression is said to be in NNF if the negation sign only appears in front of a concept name. Using De Morgan Laws and usual rules for quantifiers, an $\mathcal{ALC}$-concept description can be transformed in linear time into an equivalent one in NNF such that negation appears only in front of the atomic concept.*

$\neg(C \sqcap D) \iff \neg C \sqcup \neg D$

$\neg(C \sqcup D) \iff \neg C \sqcap \neg D$

$\neg(\forall R.C) \iff \exists R.\neg C$

$\neg(\exists R.C) \iff \forall R.\neg C$

$\neg(\neg C) \iff C$

To check the satisfiability of C, the algorithm starts with $A_0 := \{C_0(x_0)\}$ and applies the completion rules in Figure 2.2.

Definition 2.5.2 *An ABox A is called complete iff none of the transformation rules of Figure 2.2 applies to it. The ABox A contains a clash iff $\{P(x), \neg P(x)\} \subseteq A$ for some individual name x and some concept name P. An ABox is called closed if it contains a clash, and open otherwise[8].*

The $\rightarrow \sqcap$ - rule
Condition : A contains x:($C_1 \sqcap C_2$), but not both x:C_1 and x:C_2.
Action : A' := A $\cup$ {x:C_1, x:C_2}.

The $\rightarrow \sqcup$ - rule
Condition : A contains x:($C_1 \sqcup C_2$), but neither x:C_1 nor x:C_2.
Action : A' := A $\cup$ {x:C_1} , A" := A $\cup$ {x:C_2}.

The $\rightarrow \exists$ - rule
Condition : A contains (x:$\exists$ r.C), but there is no individual name z
such that x:C and x,z:r are in A.
Action : A' := A $\cup$ {y:C, (x,y):r} where y is an individual name not occurring in A.

The $\rightarrow \forall$ - rule
Condition : A contains (x:$\forall$ r.C) and (x,y):r, but it does not contain y:C.
Action : A' := A $\cup$ {y:C}

Table 2.2: Tableau Rules of Satisfiability Algorithm for $\mathcal{ALC}$ [8].

2.6 DL Inference Services

The goal of reasoning in Description Logics is to reason about a KB σ = ($\mathcal{T}$, $\mathcal{A}$), where $\mathcal{T}$ is TBox and $\mathcal{A}$ is ABox, which are expressed in the concept language $\mathcal{L}$ [15]. Therefore, we can categorize reasoning into two groups: *TBox Reasoning* and *ABox Reasoning* and the main task of each category is listed below:

2.6.1 TBox Reasoning

- **Satisfiability/Consistency** When using a DL satisfiability/consistency reasoner, various reasoning problems can be transformed into knowledge base satisfiability problems. For TBox satisfiability, we check for a model for the TBox and if there exists one, we say that the TBox is satisfiable.

- **Subsumption** Represents an is-a relation, and one has to check whether a concept C is subsumed by a concept D or if every instance of C is also a D.

- **Classification** Computation of atomic concept hierarchy based on subsumption.

Note that all reasoning services can be reduced to concept satisfiability.

2.6.2 ABox Reasoning

- **Satisfiability/Consistency** Checks if all the assertions in an ABox are consistent with the TBox-Axioms.

- **Instance Checking** Checks if an individual is an instance of a given concept.

- **Realization** Finds the most specific atomic concept of an individual by traversing the TBox subsumption. Starting from the top node, repeatedly calls the instance checking procedure.

2.7 DL Reasoners

Most DL reasoners implement tableau-based algorithms with a set of optimization techniques. The practical DL reasoners are KAON2[1], FaCT++[2], JFact[3], RacerPro[4], Pellet [5] and HermiT[6] which will be described below:

[1]http://kaon2.semanticweb.org/
[2]http://owl.man.ac.uk/factplusplus/
[3]http://jfact.sourceforge.net/
[4]http://www.racer-systems.com/
[5]http://clarkparsia.com/pellet/
[6]http://www.hermit-reasoner.com/

2.7.1 HermiT

HermiT is a DL reasoner based on the hypertableau algorithm supporting OWL 2 [19]. It is equipped with optimization techniques for classification and reasoning discussed in [17], [16], [34].

2.7.2 KAON2

Contrary to most currently available DL reasoners, such as FACT++, JFact, RacerPro, or Pellet, KAON2 [13] does not implement the tableau calculus. Rather, reasoning in KAON2 is implemented by novel algorithms which reduce a $\mathcal{SHIQ}(\mathcal{D})$ knowledge base to a disjunctive datalog program.

2.7.3 Pellet

Pellet [44] is a highly optimized open-source tableau-based DL reasoner supporting OWL 2. It incorporates optimizations for nominals, conjunctive query answering, and incremental reasoning [43].

2.7.4 FaCT++

FaCT++ [46] is a tableau highly optimized OWL 2[7] DL reasoner and the new generation of the well-known FaCT OWL-DL[8] reasoner. FaCT++ uses the established FaCT algorithms, but with a different internal architecture. Additionally, the implementation language C++ was chosen in order to create a more efficient

[7] http://www.w3.org/TR/owl2-overview/

[8] OWL DL is a sub language of OWL which places a number of constraints on the use of the OWL language constructs. See http://www.w3.org/TR/owl-ref/ for more details.

software tool, and to maximize portability. During the implementation process, new optimizations were also introduced, and some new features were added.

2.7.5 JFact

JFact [47] is the Java implementation of the FaCT++ OWL 2 DL reasoner, that has extended data types support. JFact is a port of FaCT++, so it uses the same techniques and contains the same optimizations.

2.7.6 RacerPro

The RacerPro [21] system, the successor of the Racer system [20], is a knowledge representation system that implements a highly optimized tableau calculus for a very expressive description logic. It is implemented in LISP and offers reasoning services for multiple TBoxes and for multiple ABoxes as well. The system implements the description logic SRIQ(D-) [18]. This is the basic logic $\mathcal{ALC}$ augmented with qualifying number restrictions, role hierarchies, inverse roles, transitive roles and role chains. In addition to these basic features, RacerPro also provides facilities for algebraic reasoning including concrete domains for dealing with:

- Min/max restrictions over the integers
- Linear polynomial (in-)equations over the reals or cardinals with order relations
- Nonlinear multivariate polynomial (in-)equations over complex numbers
- Equalities and inequalities of strings

RacerPro supports the specification of general terminological axioms. Multiple definitions or even cyclic definitions of concepts can be handled by RacerPro. Given a TBox, various kinds of queries can be answered, which are listed below:

- Concept consistency with respect to a TBox: Is the set of objects described by a concept not empty?
- Concept subsumption with respect to a TBox: Is there a subset relationship between the set of objects described by two concepts?
- Find all inconsistent concepts mentioned in a TBox. Inconsistent concepts might be the result of modeling errors.
- Determine the parents and children of a concept with respect to a TBox: The parents of a concept are the most specific concept names mentioned in a TBox which subsume the concept. The children of a concept are the most general concept names mentioned in a TBox that the concept subsumes. Considering all concept names in a TBox, the parents (or children) relations defines a graph structure which is often referred to as taxonomy.

This chapter described a formal definition of Description Logics, its syntax and semantics with a main focus on $\mathcal{AL}$ family, DL languages, DL inference services and also explanation on some of the DL reasoners. The following chapter will discuss the background and related work.

Chapter 3

Background and Related Work

In the last couple of years, there has been a rapid growth in providing information from large knowledge bases and due to limitations in performance gain in conventional processors, parallelization has become an important area of research. Hence, when it comes to a large and/or distributed knowledge base, we have to face different kinds of issues such as

- How we do reasoning with homogeneous/heterogeneous reasoners?
- What kind of topology we use for theorem proving in distributed systems?
- Which algorithms are more efficient for parallel reasoning?
- What are the algorithms that provide a good partitioning? and so on.

One of the attempts in parallelizing reasoning is Octopus [37]. Octopus is a parallel ATP (Automated Theorem Proving) system,which is an improved version of the singled processor ATP system THEO [33]. Inference rules used by Octopus include binary resolution, binary factoring, instantiation, demodulation, and

hash table resolutions. Octopus performs 3000-10000 inferences/second on each processor. It runs on a network of 20-40 PCs and the processors communicate using the PVM (Parallel Virtual Machine). Octopus combines learning and theorem proving together.

PVM is a software package that permits a heterogeneous collection of Unix and/or Windows computers hooked together by a network to be used as a single large parallel computer. Thus, large computational problems can be solved more cost effectively by using the aggregate power and memory of many computers.

In the rest of this chapter, the algorithms for some optimization techniques which guided my work, will be explained.

3.1 Algorithms for Computing Concept Hierarchy

The subsumption relation induces the computation of concept hierarchy[1].

In this part, we assume that subsumption costs are dominating the classification costs and are considerably higher than the costs incurred by extra operations. The following methods identify the computation of concept hierarchy, and are taken from [6]:

- Brute Force Method
- Simple Traversal Method
- Enhanced Traversal Method

[1] We assume that our concept hierarchy always contain a top element $\top$ and a bottom element $\bot$.

3.1.1 Brute Force Method

The top search for Brute Force method, computes for every atomic concept c its predecessors and can be explained as follows:

Blindly tests c $\sqsubseteq$ x for all x $\in$ X_i, where X_i is atomic concept in the TBox.

Definition 3.1.1 (Top Search) *Top Search returns a set of immediate predecessors in X_i for a given element c.*

The bottom search explained in definition 3.1.2, computes for every atomic concept its successors and is done in a dual way.

Definition 3.1.2 (Bottom Search) *Bottom Search returns a set of immediate successors of c.*

This method uses 2 * | X_i | comparisons for the step of inserting c in X_i. Summing over all steps leads to n * (n-1) comparison operations to compute the representation of a partial ordering for n elements. This is not only the worst-case complexity but also the best-case complexity of this method.

3.1.2 Simple Traversal Method

In order to avoid many of the ”brute force” method’s comparisons and instead of testing the new element c blindly with all elements; in the top search phase of ”simple traversal” method, c is pushed down the tree, and in the bottom search, phase c is pushed up, stopping when immediate predecessors or successors have been determined.

- **Top Search**

Algorithm 1 Top search phase of the "simple traversal" method [6].

```
1:  top_search(c,x) =
2:  mark(x,'visited')
3:  for all y ∈ successors(x) do
4:     if simple_top_subs(y,c) then
5:        pos-succ ← pos-succ ∪ {y}
6:  if pos-succ = ∅ then
7:     result ← x
8:  else
9:     for all y ∈ pos-succ do
10:       if y not marked as 'visited' then
11:          result ← result ∪ top_search(c,y)
```

Algorithm 2 Simple top subsumption of the "simple traversal" method [6].

```
1:  simple_top_subs?(y,c) =
2:  if y marked as 'positive' then
3:     result ← true
4:  else if y marked as 'negative' then
5:     result ← false
6:  else if subs?(y,c) then
7:     mark(y,'positive')
8:     result ← true
9:  else
10:    mark(y,'negative')
11:    result ← false
```

Starting at the top of the hierarchy, for each concept x of X_i (x is child of top), it is determined whether x has an immediate successor y satisfying c $\sqsubseteq$ y. If there are such successors, they are considered as well. Otherwise, x is added to the result list of the top search. To avoid multiple visits of elements and multiple comparison of the same element with c, this algorithm employs one label to indicate whether a node was "visited" and another label to indicate whether the subsumption test was "positive", "negative", or has not been made. The top-search is depicted in Figure 3.1. The figure

only shows the details for the upper half of the tree, as traversal is done from top. The top-search procedure is illustrated in Algorithm 1, and will be explained next. The algorithm gets two arguments as its input:

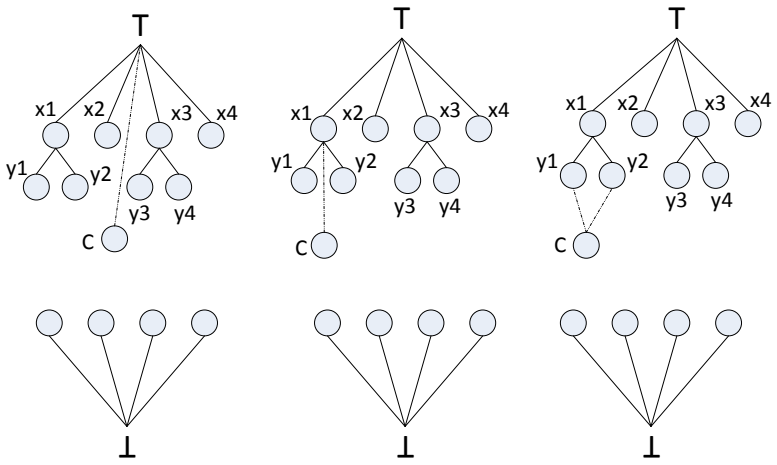


Figure 3.1: Insert a concept c through top search [6].

- Concept c which should be inserted
- Element x of X_i, which is under consideration

For the concept x, we already know that c $\sqsubseteq$ x holds; and *top-search* looks at its direct successors. Initially, the procedure is called with x = $\top$.

To check whether any direct successor y of x subsumes c we will call "*simple-top-sub?*" shown in Algorithm 1, line 4. Since our hierarchy does not need to be a tree, y may have been checked before and if we memorized the result of the previous test, we do not need to invoke the expensive subsumption procedure "subs?(y,c)"(see Algorithm 2, line 6). The direct successor for which the test was positive is located in a list called Pos-Succ.

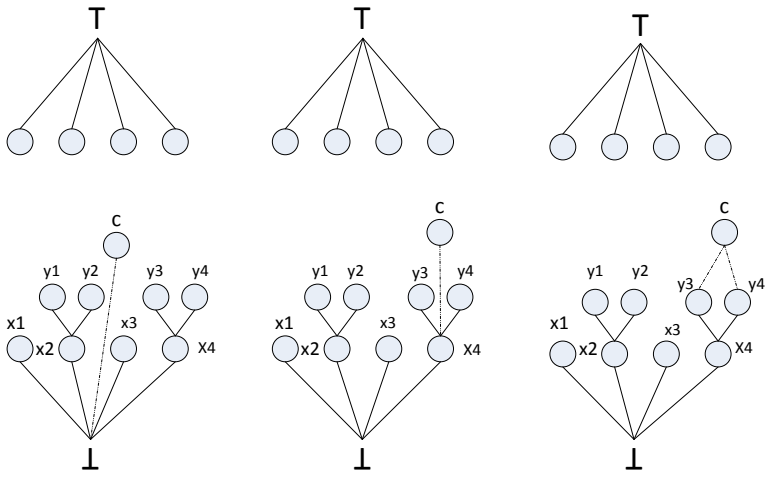


Figure 3.2: Insert a concept c through bottom search [6].

If it remains empty, x is added to the result list; otherwise top-search is called for each positive successor, but only if the concept has not been visited before.

- **Bottom Search**

  Bottom Search is symmetric to Top Search, therefore it is performed in a dual way. This means that if the tree is rotated upside down, then bottom is like top. Bottom Search is illustrated in Figure 3.2. The figure shows only the lower half of the tree, as bottom search traversal is done bottom-up.

  Starting at the bottom of the hierarchy, for each concept x of X_i (x is parent of bottom), it checks whether x has an immediate predecessor z which satisfies $c \sqsubseteq z$. If there are such predecessors, they are considered as well. The bottom-up traversal is done till the immediate successors of c are found.

3.1.3 Enhanced Traversal Method

The "simple traversal" method performs much better than the "brute force" method but it does not use all the available information. Therefore in the "enhanced traversal" method, first we can take advantage of the tests that have been performed during the top search; second, in the bottom search we can use all the information collected during the top search.

- **Top Search**

  To exploit all the information, which has been recorded during the top search, we either focus on *negative information* (i.e. a subsumption test did not succeed) or on *positive information* (i.e. a subsumption test succeeded).

  - **Using negative information**

    We check whether for any predecessor z of y the test c $\sqsubseteq$ z has failed. If this is the case (see line 6 of Algorithm 3), we can conclude that c $\not\sqsubseteq$ y) without performing the costly subsumption test. To gain maximum advantage, all predecessors of y should have been tested before the subsumption test will be done on y. Algorithm 3 shows "enhanced-top-sub" procedure.

  - **Using positive information**

    Before checking c $\sqsubseteq$ y, one can look for a successor z of y that has passed the test c $\sqsubseteq$ z. If there is such a successor one can conclude that c $\sqsubseteq$ y without any subsumption test. Although we are interested in minimizing the number of comparisons, it is more efficient to propagate positive information up through the subsumption hierarchy in-

Algorithm 3 Top search phase of the "enhanced traversal" method. The procedure top-search is the same as the "simple traversal" method, but instead of the "simple-top-sub?" procedure, it calls the "enhanced-top-sub?" procedure [6].

```
1:  enhanced_top_subs?(y,c) =
2:  if y marked as 'positive' then
3:      result ← true
4:  else if y marked as 'negative' then
5:      result ← false
6:  else if for all z ∈ predecessors(y)
                always enhanced_top_subs(z,c)
            and subs?(y,c) then
7:      mark(y,'positive')
8:      result ← true
9:  else
10:     mark(y,'negative')
11:     result ← false
```

stead of searching for a successor that has passed the search. This can be done by an easy modification of the "simple-top-sub?" procedure. When the call "subs?(y,c)" yields true, not only y is marked "positive" but also all of y's predecessors. This technique cannot be combined with the "enhanced-top-subs?" in Algorithm 3 since it reduces the number of tests if there are predecessors that have not been tested yet; however the enhanced top search tests all predecessors before making a subsumption test. None of the alternatives is better than the other one and we can see it in the Figure 3.3 and Figure 3.4.

Considering the fact that the new element c is a direct successor of y_1 but not a successor of y_2 and having the same hierarchy as Figure 3.3, two tests should be performed in order to place c in the right place in the hierarchy while using negative information in the top search

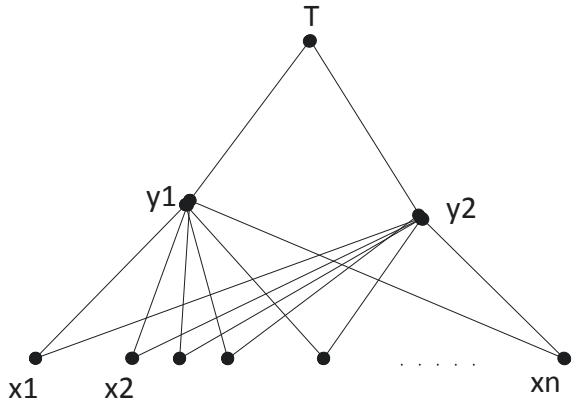


Figure 3.3: The new element c is a direct successor of y_1, but not a successor of y_2, x_1,..., x_n. Taken from [6].

algorithm: first it tests y_1, then checks x_1 but before testing x_1, its direct predecessor y_2 is tested. The negative result of this test prevents x_1,..., x_n from being tested. However, the top search using positive information tests n+2 nodes in order to place c in the hierarchy: first y_1 and then its successors x_1,..., x_n and finally y_2.

In Figure 3.4, using negative information, n+1 tests should be done: it first checks the subsumption relation between c and x_1, then checks y, but before testing y, it checks all its direct predecessors, i.e, $x_2 \ldots x_n$. However, using positive information, two tests will be performed: first x_1 and then y; and the positive result of the test is propagated to x_2, ..., x_n. There is a significant difference in performance between the two different top search methods.

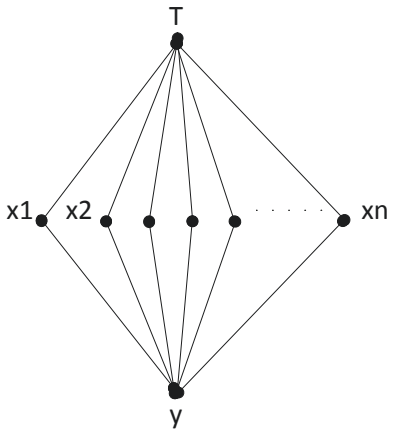


Figure 3.4: The new element c is a direct successor of y. Taken From [6].

- **Bottom Search**

  Bottom search optimizations can be applied dual to top search. This optimization is achieved by a modification of the ”enhanced-bottom-search” procedure (which is dual to ”enhanced-top-search”).

3.2 Preprocessing optimization techniques to simplify a knowledge base

The axioms that constitute a DL KB may contain considerable redundancy and may make unnecessary use of general axioms (general axioms are costly to reason with due to the high degree of non-determinism that they have) [25]. Hence, it is useful to preprocess a KB, applying syntactic simplifications and manipulations. These optimizations are as follows:

- **Normalization**

  Tries to simplify the KB by identifying syntactic equivalences, contradictions and tautologies. For example, (D $\sqcap$ C) could be simplified to $\sqcap\{$C,D$\}$.

The formal definition of normalization and simplification functions for ALC has been explained in [26]. It can be extended to more expressive DLs. Additional simplifications would clearly be possible. For example, $\forall$R.C $\sqcap$ $\forall$R.D could be simplified to $\forall$R.*Norm*(C$\sqcap$D).

Normalizations help the subsumption-testing algorithm to detect contradictions like $\sqcap\{C,D\}$,$\neg(\sqcap\{C,D\})$ directly without the need for further expansion.

Definition 3.2.1 (Lazy Unfolding) *In the lazy unfolding technique, concept names are unfolded, using the concept definitions in TBox, but as required by the progress of the tableau completion rules. The advantage of this technique is that the tableau algorithm can find contradictions between concept names before adding expressions derived from TBox (for further information see [25] and [26]).*

- **Advantages**

  It is easy to be implemented and can be used with most logics and algorithms. Normalization simplifies or even avoids subsumptions/ satisfiability problems by detecting syntactically obvious satisfiability and unsatisfiability. It complements lazy unfolding and improves easy clash detection. While normalizing, the elimination of redundancies and the sharing of syntactically equivalent structure may lead to a KB that can be more compactly stored [26].

- **Disadvantages**

The overhead involved in the procedure, although this is relatively small. For very unstructured KBs there may be no benefit, and it might even slightly increase size of KBs [26].

- **Absorption**

  Since general axioms are costly to reason with due to the high degree of non-determinism that they can introduce, it makes sense to eliminate them from a KB whenever it is possible. The basic idea is that a general axiom of the form C $\sqsubseteq$ D, where C may be a non-atomic concept, is manipulated (using the equivalence in Table 3.1 so that it has the primitive definition of A $\sqsubseteq$ D. where A is an atomic concept name.

$$C_1 \sqcap C_2 \sqsubseteq D \Longleftrightarrow C_1 \sqsubseteq D \sqcup \neg C_2$$
$$C \sqsubseteq D_1 \sqcap D_2 \Longleftrightarrow C \sqsubseteq D_1, C \sqsubseteq D_2$$

Figure 3.5: Axiom equivalences used in absorption [26].

$C_1 \sqcap C_2 \sqsubseteq D \iff C_2 \sqsubseteq D \sqcup \neg C_1 \iff C_1 \sqsubseteq D \sqcup \neg C_2$

$C \sqsubseteq D_1 \sqcap D_2 \iff C \sqsubseteq D_1$ and $C \sqsubseteq D_2$

Table 3.1: Axiom equivalences used in the absorption technique [26].

  - **Advantages**

    It can lead to a dramatic improvement in performance. It is logic and algorithm independent [26].

  - **Disadvantages**

    The disadvantage is the overhead required for the pre-processing, although this is generally small compared to classification times [26].

3.3 Optimization techniques to avoid subsumption tests

The classification optimization described in Section 3.1 helps to reduce the number of subsumption tests that are performed when classifying a KB and the combination of normalization, simplification, and lazy unfolding facilitates the detection of obvious subsumption relationships. However, detecting obvious non-subsumption (satisfiability) is more difficult in tableau algorithms and this is unfortunate. In this section, we will review some related works, which describe the optimization techniques to avoid subsumption tests. These techniques are applied to different parts of the classification process and can be done by using relations, which are obvious when we are looking at the concept definition.

- **First technique: Told (non.) subsumers**

  Assume that we are inserting concept c whose description mentions x or $\neg$x explicitly as a conjunct, then it is obviously the case that c $\sqsubseteq$ x or c disjoint to x (x is a told subsumer of c). The information that c is subsumed by its told subsumers can be propagated through the existing hierarchy prior to the top search, e.g. by pre-setting the markers to positive for told subsumers and all its predecessors [6].

- **Second technique is applicable if concepts are conjunctive and are inserted in the subsumption hierarchy following the definition-order**

  In this case, the bottom search case can be completely avoided if a primitive concept has to be classified. Such a concept C can only subsume the bottom concept and concepts for which C is a told subsumer. Since the second type of subsumees are not present in the actual hierarchy when inserting

according the definition order, the result of the bottom search is just the bottom concept $\perp$. In this case, there are no GCIs in the TBox [6].

- **Third Optimization technique can be used as a pre-test before calling the subsumption algorithm**

  By extracting and caching all the primitive components of all concepts, it becomes possible to check whether there is a subsumption relation. C can only be subsumed by D if the set of primitive components of D is a subset of the set of primitive components of C. Thus if the subset test gives a negative result, the subsumption algorithm need not be called. Obviously, this is faster than a subsumption test [6].

- **Final Optimization technique is caching partial tableau expansion trees**

  In this technique, we try to use cached results from previous tableau tests to prove non-subsumption without performing a new satisfiability test. For example, given the two concepts A and B defined by the following axioms [26]:

  $A \equiv C \sqcap \exists R1.C1 \sqcap \exists R2.C2$

  $B \equiv \neg D \sqcup \forall R3.\neg C3$

  Then B does not subsume A if the concept $A \sqcap \neg B$ is satisfiable. If the tableau expansion trees for A and B have already been cached, then the satisfiability of the conjunction can be shown by a tree consisting of the trees for A and $\neg B$ that joined at their root node, as shown in Figure 3.6.

  Figure 3.6 show that the union of root node labels does not contain a clash and that no tableau expansion rules are applicable to the new tree. The

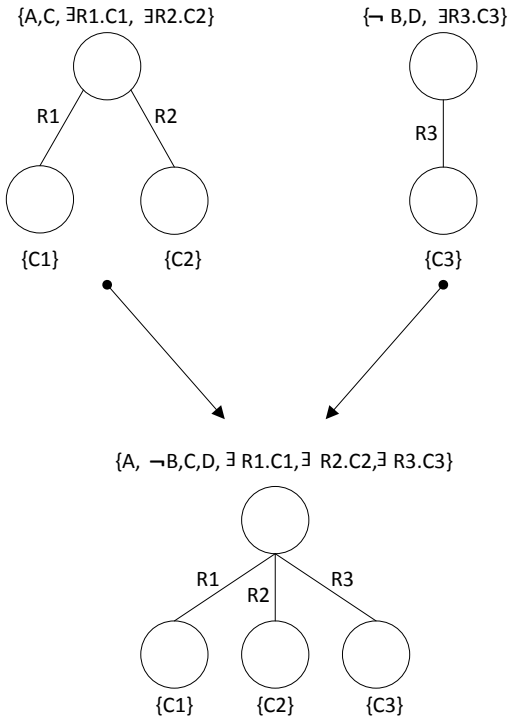


Figure 3.6: Joining expansion trees for A and ¬B [26].

caching technique can be expanded in order to avoid construction of obviously satisfiable and unsatisfiable sub-trees during the tableau expansion. For example, if a leaf node x is about to be expanded and L(x) = $\{A\}$, unfolding and expansion of L(x) is unnecessary if A has already been cached as satisfiable ($L_c(x) \neq \bot$) or unsatisfiable ($L_c(x) = \bot$).

– **Disadvantages**

One of the disadvantages of this technique is to have the overhead of sorting partial trees. It also has the overhead of satisfiability tests on concepts and their negation in order to create the partial trees that will be cached. To determine if the cached partial trees can be merged, this

technique puts overhead on the system; which is wasted if they can not be merged. This optimization technique can be used in classifying a KB or performing many similar satisfiability tests, however it does not have a value for performing single tests.

3.4 Optimization techniques for speeding up subsumption tests

In this section, optimization techniques of the subsumption algorithm will be considered. These techniques show how we can take advantage of the already computed and stored subsumption relationships during the classification process. The subsumption relationships between concepts can be determined by using satisfiability algorithms. A satisfiability algorithm may detect a contradiction during model generation if the previous subsumption relationships are taken into account. Suppose we have A subsumes B, if during the model generation a concept is defined to be both instance an of ¬A and B, then a contradiction will be detected without expanding the definition of A and B. This approach only works if we do not expand the concept definitions before the satisfiability check. If an expansion is done ”by need” during the satisfiability test (lazy unfolding), then the order of concept name expansion may have a considerable impact on the runtime behavior. To make the expansion process more efficient, we expand the concept names according to the inverse of their definition order, however, this means that for each expansion operation one has to go through the list of all expandable names, look for the maximal one with respect to the definition order.

To avoid searching for a maximal name there is another approach, which expands the concept definitions in arbitrary order. In this approach when a name is

expanded it is not removed, but just marked as expanded and it takes more memory. To optimize the satisfiability algorithm employed in KRIS[2], three versions have been implemented:

- **The first one** *takes expanded concept descriptions as input*. Since these descriptions do not contain names of defined concepts, contradictions can only appear between ”primitive concepts”, i.e., concepts names which are not defined in the TBox.

- **The second one** *successively expands the concept description during model generation* but keeps the names as described above. This allows the algorithm to detect a contradiction not only between primitive concepts but also between names of defined concepts.

- **The third one** *is the refinement of the second one*, in a way that already computed subsumption relationships are taken into account when looking for obvious contradictions.

After having explained the optimization techniques for computing the concept hierarchy, avoiding the subsumption tests, and speeding up the subsumption test in this section, there will be an overview of atomic decomposition in the next section.

[2]It is an implemented prototype of a KL-ONE system where all reasoning facilities are realized by sound and complete algorithms. KRIS provides a concept language, and an assertional language [5].

3.5 Atomic Decomposition

In recent years modules have frequently been used for ontology development and understanding. This happens because a module captures all the knowledge an ontology contains in a given area, and often is much smaller than the whole ontology. One useful modularization technique for expressive ontology languages is locality-based modularization, which allows for fast (polynomial) extraction of modules. In order to better understand the modular structure of an ontology, a technique called Atomic Decomposition can be used [45]. It efficiently builds the structure for all possible modules of an ontology, which can be used for quick extraction of modules, or investigate dependencies between modules, and so on.

In this section, we will present an overview of the work of other researchers for Atomic Decomposition. The details of the work as well as the corresponding algorithms are described in [45].

In [45], it is assumed that the reader is familiar with the notion of OWL 2 axiom, ontology and entailments. An *entity* is a named element of the signature of an ontology. For an axiom α, the signature of that axiom is denoted by $\widetilde{a}$, i.e. the set of all entities in α.

Definition 3.5.1 (Signature) *Let N_C be be a set of concept names, and N_R a set of role names. A signature Σ is a set of terms, i.e., $\Sigma \sqsubseteq N_C \cup N_R$. We can think of a signature as specifying a topic of interest. Axioms using only terms from Σ are on-topic. For instance, if Σ = {Animal,Duck,Grass,eats} then Duck $\subseteq \exists$eats is on-topic, while Duck $\sqsubseteq$ Bird is off-topic. Given an ontology O (axiom a), its signature is denoted with $\widetilde{O}(\widetilde{a})$ [50].*

Definition 3.5.2 (Module) *Let O be an ontology and* Σ *be a signature. A subset M of the ontology is called a module of O w.r.t.* Σ *if* $M \models \alpha \iff O \models \alpha$ *for every axiom* α *with* $\widetilde{a} \subseteq \Sigma$.

One of the ways to build modules is to use locality of axioms.

Definition 3.5.3 (Semantic Locality) *An axiom* α *is called* $\top(\bot)$*-local w.r.t a signature* Σ *if replacing all named entities in* $\widetilde{a} \setminus \Sigma$ *with* $\top$*(resp.* $\bot$*) makes that axiom a tautology. An axiom* α *is called a tautology if it is local w.r.t.* $\widetilde{a}$*. An axiom* α *is called global if it is non-local w.r.t.* $\emptyset$.

It is assumed that the locality checker provides a method isNonLocal(α) that returns true iff the axiom α is non-local.

Definition 3.5.4 (Atomic Decomposition) *A set of axioms A is an atom of an ontology O, if for every module M of O, either* $A \subseteq M$ *or* $A \bigcap M = \emptyset$*. An atom A is dependent on B (written* $B \preceq A$*) if* $A \subseteq M$ *implies* $B \subseteq M$*, for every module M. An Atomic Decomposition of an ontology O is a graph* $G = (S, \preceq)$*, where S is the set of all atoms of O.*

In other words, Atomic Decomposition is a method to partition an ontology into pieces called atoms. An atom is a set of axioms that appear in every module all together (or none of them). They could be viewed as a smallest pieces of modules.

In our research, Atomic Decomposition is considered as a black box. Hence, we decompose ontologies using a jar file provided by Manchester University.

In this section, we briefly explained atomic decomposition of ontologies; in the next section, we will provide an overview of concurrent classification of EL Ontologies.

3.6 Concurrent Classification of EL Ontologies

Many works have focused on the development of techniques to reduce classification times. Numerous approaches have been proposed for optimizing the underlying (mostly tableau-based) procedures by reducing the number of redundant inferences and making the computation more goal-directed. Another way of reducing the classification time, which is studied in this section, is to perform several inferences at the same time, i.e., concurrently. Concurrent algorithms and data structures have gained substantial practical importance due to the widespread availability of multi-core and multi-processor systems [28]. In [28] an optimized consequence-based procedure for classification of ontologies expressed in a polynomial fragment *ELHR+* of the OWL 2 *EL* profile. As mentioned earlier, the procedure can take advantage of multiple processors/cores, which increasingly prevail in computer systems. The solution is based on a variant of the given clause saturation algorithm for first-order theorem proving, where derived axioms are assigned to contexts within which they can be used and which can be processed independently.

In this section, we will briefly present the implementation of the procedure within the Java-based reasoner ELK, which is developed by another researcher. The thorough explanation of the implementation as well as the corresponding algorithms are described in [28].

The implementation is light-weight in the sense that an overhead of managing concurrent computations is minimal. This is achieved by employing lock-free data structures and operations such as compare-and-swap [28].

The key idea of the system design is based on the notion of active context. A

context is active if the scheduled queue for this context is not empty. The algorithm maintains the queue of active contexts to preserve this invariant. For every input axiom, the algorithm takes every context assigned to this axiom and adds this axiom to the queue of the scheduled axioms for this context). Because the queue of scheduled axiom becomes non-empty, the context is activated by adding it to the queue of active contexts. Afterwards, each active context is repeatedly processed in the loop. The conclusions of computed inferences are inserted into (possibly several) sets of scheduled axioms for the contexts assigned to this conclusion, in a similar way as it is done for the input axiom. Once the context is processed, i.e., the queue of the scheduled axioms becomes empty and the loop quits, and the context is deactivated.

This chapter described the background review on the closely related works. In the following chapter, parallel processing in general will be discussed.

Chapter 4

Toward Parallel Processing With Semantic Web

This chapter presents the parallel processing and the programming model in MapReduce. It also gives an overview of the related research work.

4.1 MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key [14].

MapReduce is inspired by the map and reduce primitives present in Lisp and other functional languages. In MapReduce, a map operation is applied to each logical ”record” in the input in order to compute a set of intermediate key/value pairs, and

then applying a reduce operation to all values that shared the same key, in order to combine the derived data properly.

4.1.1 Programming Model

The computation takes a set of input key/value pairs and produces a set of output key/value pairs. It consists of two functions, Map and Reduce.
The Map function, takes an input key/value pair and produces a set of intermediate key/value pairs. MapReduce combines all the intermediate values associated with the same intermediate key $\mathcal{I}$ and passes them to the Reduce function.
The Reduce function, accepts an intermediate key $\mathcal{I}$ and all the values for that key. It merges those values to form a smaller set of values. Zero or one output value is produced per Reduce function. An iterator provides value from intermediate values to the Reduce function.

Map (k1,v1)	$\rightarrow$	list (k2,v2)
Reduce (k2,list(v2))	$\rightarrow$	list (v2)

Figure 4.1: Map-Reduce [14].

The input keys and values are drawn from different domain than output keys and values; however, the intermediate keys and values are from the same domain as output keys and values.

4.1.2 Execution

First, MapReduce splits the input into $\mathcal{M}$ pieces and many copies of the program runs on a cluster of machines. The special copy of the program is called Master

and the rest are workers. Master picks the idle worker and assigns a M map task or a R reduce task.

A worker which gets a map task, reads the input and parses the key/value pair and passes to the Map function. The intermediate key/value pair results from Map function is buffered into memory. Periodically, the buffered pairs are stored on a local disk, partitioned into R region by partitioning function. The locations of buffered pairs on disk are passed to Master which is forwarding them to the Reduce workers.

As soon as a reduce worker is notified by master about the locations, through RPC, it gets the data from local disk of map workers. It reads all intermediate data and sorts based on the key so that all occurrences of the key are grouped together. Sorting is done as typically many different keys are mapped to the same reduce task.

Reduce worker then iterates over the data and for each key, it passes the key and the corresponding set of values to the Reduce function. The output is appended to the output file of this reduce partition.

When execution is completed, master worker returns the handle to the program [14].

4.1.3 Examples

There are some researches about MapReduce which are loosely related to our work. Some examples of MapReduce in Semantic Web follows:

A distributed technique is introduced in [49] to materialize the closure of an RDF graph based on MapReduce. Another research, [35], explains a MapReduce

Algorithm for EL+. Another work in MapReduce, is named WebPIE [48], which is a distributed technique, introduced to reason under RDFS and OWL with Horst Semantics using MapReduce.

In this section, we mentioned some examples of MapReduce. In the following section, we will note some examples for Distributed Reasoning/Inferencing in Semantic Web.

4.2 Distributed Reasoning/Inferencing

In this section, we will present some research on Distributed Reasoning. These works loosely relate to our work. SAOR is an optimizations of rule-based materialization approaches for reasoning over large static RDF datasets [24]. Distributed reasoning over large scale RDF datasets, and a solution to speeddating in elastic regions is described in [29]. Another distributed reasoning method is proposed in [40] that preserves soundness and completeness of reasoning under the original OWL import semantics. The method is based on resolution methods for ALCHIQ ontologies that is modified to work in a distributed setting. A partitioning scheme for abox partitioning is defined in [51]. This research also introduces classes of rules which can be used to perform complete parallel inferencing on abox partitions.

In this chapter, we had an overview of parallel processing in Semantic Web, and we browsed some researches on MapReduce as well as Distributed Reasoning. In the next chapter, Parallel TBox Classification will be presented in detail.

Chapter 5

Parallel TBox Classification

This chapter presents the main contributions of the thesis on parallel TBox classification and the evolution of its underlying algorithms. The content of this chapter is divided into two sections.

First, in Section 5.1, which contains the theoretical contributions of the thesis, the algorithms for Parallel TBox Classifier and their evolution will be explained as follows:

- *First Generation* : First generation of Parallel Classifier is a set of sound but incomplete algorithms. In Section 5.1.1, we will explain that in this generation, we explored how parallel classification was done with deliberate sacrifice in completeness.

- *Second Generation* : Second generation of Parallel Classifier is a set of sound and complete algorithms. The algorithms are thoroughly explained in Section 5.1.2. This section will also elaborate the scenarios which can cause incompleteness, and discuss the corresponding algorithms to achieve

completeness.

- *Third Generation* : The third generation of Parallel TBox Classifier, which also can be referred to as Concurrent TBox Classifier, is where the employed algorithms are sound and complete, and it classifies the TBox concurrently, and more efficiently utilizing a benchmark with much bigger ontologies. The algorithms for third generation of Parallel TBox Classifier are described in Section 5.1.3.

Then, in Section 5.2, the design of a running prototype for Parallel TBox Classifier will be demonstrated.

The prototype is independent of a particular logic or reasoner. This architecture was deliberately designed to facilitate our experiments by using existing OWL reasoners to generate auxiliary information as well as making the TBox Classifier independent of particular DLs. Racer is only used for generating the input files for our prototype.

This prototype was utilized for the assessment of the algorithms in Section 5.1, and its main components follow:

- *Ontology Loader* : The Ontology Loader is responsible for loading an ontology selected by the user. The Ontology Loader is described in Section 5.2.1.

- *Configuration Manager* : The Configuration Manager is responsible for checking and utilizing user preferences about how the Parallel TBox Classifier should be configured. The Configuration Manager is described in Section 5.2.2.

- *Parser* : The Parser is responsible for parsing the ontology input file which is provided by Racer. The Parser is described in Section 5.2.3.

- *Preprocessor* : The Preprocessor is responsible for preprocessing of the parsed information and computing a Topological Sort Order. The Preprocessor is described in Section 5.2.4.

- *Partition Manager* : The Partition Manager is responsible for partitioning the list of concepts to be inserted. The list can be a Topologically Sorted list which is generated by Preprocessor component or a non-sorted list. The details of Partition Manager are described in Section 5.2.5.

- *Serializer* : The Serializer is responsible for serialization of logs as well as the statistics. The Serializer is described in Section 5.2.6.

- *TBox Classifier* : The TBox Classifier is the main component and responsible for classification of TBox in parallel. The description of how TBox Classifier works, is explained in Section 5.2.7.

5.1 Algorithms for Parallel TBox Classifier

5.1.1 First Generation

This section describes the algorithms for the first generation of Parallel TBox Classifier. In contrast to starting to implement a parallel TBox classifier, we decided to first conduct a field study with the goal to evaluate the impact of two control parameters, number of threads and partition size[1], on the completeness of

[1]Number of concepts assigned to every thread and are expected to be inserted by a corresponding thread.

the classifier if one assumes a type of parallelization that deliberately sacrifices completeness. By using such a strategy, we wanted to get some experience about the quality of a sound but incomplete parallel classifier, or, in other words, we wanted to find out how many nodes are misplaced during TBox classification.

To manage concurrency in the system, at least two shared-memory approaches could be taken into account by using either (i) sets of local trees (so-called ParTree approach) or (ii) one global tree. In the ParTree algorithm [42] a local tree would be assigned to each thread, and after all the threads have finished the construction of their local hierarchy, the local trees need to be merged into one global tree. TBox classification through a local tree algorithm would not need any communication or synchronization between the threads. ParTree is well suited for distributed systems which do not have shared memory. The global tree approach was chosen because it implements a shared space which is accessible to different threads running in parallel, avoids the large scale overhead of ParTree on synchronizing local trees and also we were not planning to design a distributed system.

To ensure data integrity, a locking mechanism for single nodes is used. This allows a proper lock granularity and helps to increase the number of simultaneous write accesses to the subsumption hierarchy under construction.

To ensure avoiding unnecessary tree traversals and tableau subsumption tests when computing the subsumption hierarchy, the parallel classifier adapts the enhanced traversal method, explained in Section 3.1.3, which is an algorithm that was designed for sequential execution (see Algorithms 6 and 7). Algorithm 6 is used in Algorithm 5.

The procedure parallel_tbox_classification is sketched in Algorithm 4. It is called with a list of named concepts, which are sorted in topological order w.r.t. to

the initial taxonomy created from the already known predecessors and successors of each concept (using the told subsumer information). Alternatively, the procedure parallel_tbox_classification can be also executed with a random order of the given concept list. The classifier assigns partitions with a fixed[2] or dynamically[3] increased size from the concept list to idle threads and activates idle threads with their assigned partition using the procedure insert_ partition sketched in Algorithm 5. All threads work in parallel, and there exists either a fixed number of threads or the number of threads grows dynamically.

Algorithm 4 parallel_tbox_classification(*concept_list,shuffle_flag*)

```
topological_order_list ← topological_order(concept_list)
if shuffle_flag then
   topological_order_list ← random_shuffle(topological_order_list)
repeat
   assign each idle thread t_i a partition p_i from topological_order_list
   run idle thread t_i with insert_partition(p_i)
until all concepts in topological_order_list are inserted
compute missing subsumptions and ratio
print statistics
```

The procedure insert_ partition inserts all concepts of a given partition into the global taxonomy. For updating a concept or its predecessor or successors, it locks the corresponding nodes. It first performs for each concept the top-search phase (starting from the top concept $\top$) and afterwards the bottom-search phase (starting from the bottom concept $\bot$).

In the First Generation of Parallel TBox Classifier, the degree of incompleteness caused by classifying partitions of concepts in parallel was tested, which is documented in Section 6.2. For a variety of ontologies it turned out that a sur-

[2]If Partition Size *n* is initialized to 5, then each partition contains n^1 (e.g. 5) concepts.

[3]If Partition Size *n* is initialized to 5, then each partition contains n^i, where i={1,2,3,..}, (e.g. 5,25,125,...) concepts.

Algorithm 5 insert_ partition(*partition*)

```
for all new ∈ partition do
  parents ← top_search(new,⊤)
  lock(new)
  set predecessors of new to parents
  for all pred predecessor of new do
    lock(pred)
    add new to successors of pred
    unlock(pred)
  unlock(new)
  children ← bottom_search(new,⊥)
  lock(new)
  set successors of new to children
  for all succ successor of new do
    lock(succ)
    add new to predecessors of succ
    unlock(succ)
  unlock(new)
```

Algorithm 6 top_search(*new*,*current*)

```
mark(current,'visited')
pos-succ ← ∅
for all y ∈ successors(current) do
  if enhanced_top_subs(y,new) then
    pos-succ ← pos-succ ∪ {y}
if pos-succ = ∅ then
  return {current}
else
  result ← ∅
  for all y ∈ pos-succ do
    if y not marked as 'visited' then
      result ← result ∪ top_search(new,y)
  return result
```

Algorithm 7 enhanced_top_subs(*current*,*new*)

if *current* marked as 'positive' **then**
  return *true*
else if *current* marked as 'negative' **then**
  return *false*
else if for all $z \in$ predecessors(*current*)
      always enhanced_top_subs(*z*,*new*)
    and subsumes(*current*,*new*) **then**
  mark(*current*,'positive')
  return *true*
else
  mark(*current*,'negative')
  return *false*

Algorithm 8 subsumes(*subsumer*,*subsumee*)

Checks whether subsumer subsumes subsumee using computed subsumer information provided by Racer.

prisingly few number of subsumptions were missed. This motivated the work to improve the Parallel TBox Classifier which will be explained in next section. The scenarios for the incompleteness as well as the algorithms for its resolution will also be described in the next section.

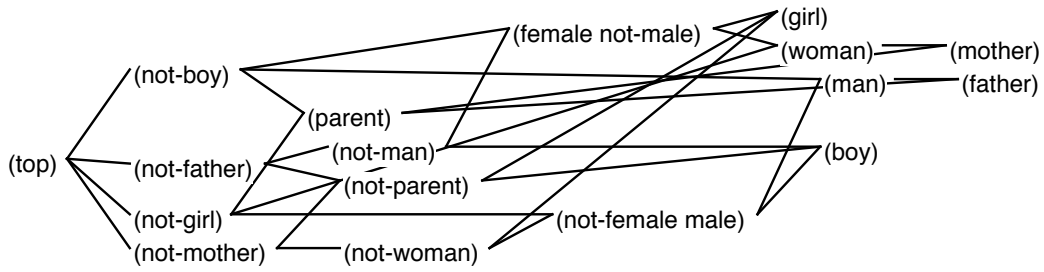


Figure 5.1: Complete subsumption hierarchy for yaya-1

5.1.2 Second Generation

In this section, first two scenarios will be illustrated which may cause that a concept is misplaced in the taxonomy due to parallel classification, and therefore causes incomplete classification.

Then, underlying *sound and complete* algorithm for parallel classification of DL ontologies is described.

Example Scenario

We use a very small ontology named yaya-1 with 16 concepts (see Figure 5.1 and 5.2).

For this example, the system is configured so that it runs with 4 threads and partition size 3 (e.g. number-of-tasks-per-thread). First, the list of concepts which is preprocessed to the form of Topological Sort Order, is partitioned using fixed-size partitioning (See Sections 5.2.4 and 5.2.5). Then, the partitions are assigned to the threads (e.g. round-robin). For instance, in Figure 5.3 a list of concepts allocated to each thread is shown. The two possible scenarios that may lead to a situation where the correct place of a concept in the hierarchy is overlooked, are described as follows.

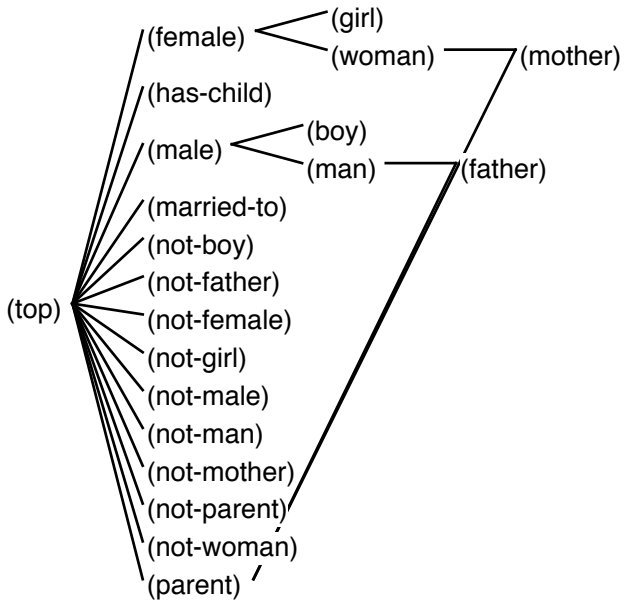


Figure 5.2: Told subsumer hierarchy for yaya-1

Scenario I: In top search, as the new concept is pushed downward, right after the children of the current concept have been processed, at least one new child is added by another thread. In this scenario, the top search for the new concept is not aware of the recent change and this might cause missing subsumptions if there is any interaction between the new concept and the added children. The same might happen in bottom search if the bottom search for the new concept is not informed of the recent change to the list of parents of the current node.

Scenario II: Between the time that top search has been started to find the location of a new concept in the taxonomy and the time that its location has been decided, another thread has placed at least one other concept into the hierarchy, which the new concept has an interaction with. Again, this might cause missing subsumptions and is analogously also applicable to bottom search.

In our example (yaya-1), due to the small size of the taxonomy, scenario I

thread#1 $\longrightarrow$ (female not-male), girl, parent
thread#2 $\longrightarrow$ woman, mother, (male not-female)
thread#3 $\longrightarrow$ man, boy, father
thread#4 $\longrightarrow$ not-boy, not-father, not-girl
thread#1 $\longrightarrow$ not-man, not-mother, not-parent, not-woman

Figure 5.3: Concept assignments to each thread for classifying yaya-1

was not encountered, however, scenario II occurred in our experiments because thread#1 inserted (female not-male)[4] and thread#2 added woman independently into the taxonomy and due to the parallelism each thread did not have any information regarding the latest concept insertion by other threads (see Figure 5.3). Hence, both (female not-male) and woman were initially placed under the top concept although woman should be a child of (female not-male) (see Figure 5.1). This was discovered and corrected by executing lines 6-7, 16-17, and 25-36 in Algorithm 10 as shown below.

The procedure parallel_tbox_classification is sketched in Algorithm 9. It is called with a list of named concepts and then preprocessing of the list results in topological order w.r.t. to the initial taxonomy created from the already known told ancestors and descendants of each concept (using the told subsumer information). Then, the topological sort order list is partitioned into fixed-size partitions, which are assigned in a round-robin manner to idle threads. These threads are activated with their assigned partition using the procedure insert_partition outlined in Algorithm 10. All threads work in parallel with the goal to construct a global subsumption tree (taxonomy). They also share a global array *inserted_concepts* indexed by thread identifications. Nodes in the global tree as well as entries in the array are locked for modification.

[4]This notation indicates that the concepts female and not-male are synonyms for each other.

Algorithm 9 parallel_tbox_classification(*concept_list*)

```
topological-order-list ← topological_order(concept_list)
repeat
   wait until an idle thread t_i becomes available
   select a partition p_i from topological-order-list
   run thread t_i with insert_partition(p_i, t_i)
until all concepts in topological-order-list are inserted
compute ratio and overhead
print statistics
```

The procedure insert_partition inserts all concepts of a given partition into the global taxonomy. For updating a concept or its parents or children, it locks the corresponding nodes. In order to avoid unnecessary tree traversals and tableau subsumption tests when computing the subsumption hierarchy, the parallel classifier adapts the enhanced traversal method, explained in Section 3.1.3. (see Algorithms 7 and 11, which outline the traversal procedures for the top-search phase).

Therefore, the classifier for each concept *new* performs the top-search phase (starting from the top concept) and possibly repeats the top-search phase for *new* if other threads updated the list of children of its parents. Then, it sets the parents of *new* and adds *new* for each parent to its list of children (line 5 to 14). Afterwards the bottom-search phase (starting from the bottom concept) is performed. Analogously to the top-search phase the bottom search is possibly repeated and sets the children of *new* and updates the parents of the children of *new* (line 15 to 24). After finishing the top and bottom search for *new* it is checked again whether other threads updated its entry in *inserted_concepts* and the top and/or bottom search needs to be repeated (line 26 to 36). Finally, *new* is added to the entries in *inserted_concepts* of all other busy threads. Hence, other threads are notified of added concept (line 37 to 40).

Algorithm 10 insert_partition(*partition,id*)

```
1:  lock(inserted_concepts(id))
2:  inserted_concepts(id) ← ∅
3:  unlock(inserted_concepts(id))
4:  for all new ∈ partition do
5:     parents ← top_search(new,⊤)
6:     while ¬ consistent_in_top_search(parents,new) do
7:        parents ← top_search(new,⊤)
8:     lock(new)
9:     predecessors(new) ← parents
10:    unlock(new)
11:    for all pred ∈ parents do
12:       lock(pred)
13:       successors(pred) ← successors(pred) ∪ {new}
14:       unlock(pred)
15:    children ← bottom_search(new,⊥)
16:    while ¬ consistent_in_bottom_search(children,new) do
17:       children ← bottom_search(new,⊥)
18:    lock(new)
19:    successors(new) ← children
20:    unlock(new)
21:    for all succ ∈ children do
22:       lock(succ)
23:       predecessors(succ) ← predecessors(succ) ∪ {new}
24:       unlock(succ)
25:    check ← check_if_concept_inserted(new, inserted_concepts(id))
26:    if check ≠ 0 then
27:       if check = 1 ∨ check = 3 then
28:          new_predecessors ← top_search(new,⊤)
29:          lock(new)
30:          predecessors(new) ← new_predecessors
31:          unlock(new)
32:       if check = 2 ∨ check = 3 then
33:          new_successors ← bottom_search(new,⊥)
34:          lock(new)
35:          successors(new) ← new_successors
36:          unlock(new)
37:    for all busy threads t_i ≠ id do
38:       lock(inserted_concepts(t_i))
39:       inserted_concepts(t_i) ← inserted_concepts(t_i) ∪ {new}
40:       unlock(inserted_concepts(t_i))
```

Algorithm 11 top_search(*new*,*current*)

```
mark(current,'visited')
pos-succ ← ∅
captured_successors(new)(current) ← successors(current)
for all y ∈ successors(current) do
  if enhanced_top_subs(y,new) then
    pos-succ ← pos-succ ∪ {y}
if pos-succ = ∅ then
  return {current}
else
  result ← ∅
  for all y ∈ pos-succ do
    if y not marked as 'visited' then
      result ← result ∪ top_search(new,y)
  return result
```

To resolve the possible incompleteness caused by parallel classification explained earlier, we utilize Algorithms 12 and 13.

The procedure consistent_in_bottom_search is not shown as it is symmetric to consistent_in_top_search.

Algorithm 12 consistent_in_top_search(*parents*,*new*)

```
for all pred ∈ parents do
  if successors(pred) ≠ captured_successors(new)(pred) then
    diff ← successors(pred) \ captured_successors(new)(pred)
    for all child ∈ diff do
      if subsumption_possible(child,new) then
        return false
return true
```

Algorithms 12 illustrates the solution for *Scenario I*, described in Section 5.1.2 on page 57. As already described, in top search we start traversing from the top concept to locate the concept *new* in the taxonomy. At time *t1*, when top_search is called, we capture the children information "captured_successors" of the con-

cept *current*; the children information is stored relative[5] to the concept *new* being inserted (we use an array of arrays) and captures the successors of the concept *current* (see Algorithm 11). As soon as top_search is finished at time *t2*, and the parents of the concept *new* have been determined, we check if there has been any update on the children list of the computed parents for *new* between *t1* and *t2* (e.g., see Algorithm 12 on how this is discovered). If there is any inconsistency and also if there is a possible subsumption[6] between *new* and any concept newly added to the children list, we rerun top_search until there is no inconsistency (see line 6 in Algorithm 10).

The same process as illustrated in Algorithm 12 happens in bottom search. The only difference is that parents information is captured when bottom search starts; and when bottom search finishes, the inconsistency and interaction is checked between the parent list of the computed children for *new* and the "captured_predecessors".

Algorithm 13 describes the solution for *Scenario II*, explained in Section 5.1.2 on page 57. Every time a thread inserts a concept in the taxonomy, it notifies the other threads by adding the concept name to their "inserted_concepts" list. Therefore, as soon as a thread finds the parents and children of the *new* concept by running top_search and bottom_search; it checks if there is any interaction between *new* concept and the concepts located in the "inserted_concepts" list. Based on the interaction, top_search and/or bottom_search need to be repeated accordingly.

The fact that algorithms are sound is obvious as every "yes" answer for a subsumption test is a valid answer. Therefore, soundness is preserved.

[5]Otherwise a different thread could overwrite captured_successors for node *current*. This is now prevented because each concept (*new*) is inserted by only one thread.

[6]This is checked by subsumption_possible using pseudo model merging [22], where a sound but incomplete test for non-subsumption on pseudo models of named concepts and their negation is utilized (pseudo model information is provided in the input file).

Algorithm 13 check_if_concept_inserted(*new,inserted_concepts*)

```
The return value indicates whether and what type of re-run needs to be done:
0 : No re-run in needed
1 : Re-run TopSearch because a possible parent could have been overlooked
2 : Re-run BottomSearch because a possible child could have been overlooked
3 : Re-run TopSearch and BottomSearch because a possible parent and child
could have been overlooked
if inserted_concepts = ∅ then
   return 0
else
   for all concept ∈ inserted_concepts do
      if subsumption_possible(concept,new) then
         if subsumption_possible(new,concept) then
            return 3
         else
            return 1
      else if subsumption_possible(new,concept) then
         if subsumption_possible(concept,new) then
            return 3
         else
            return 2
   return 0
```

Proposition 1 (Completeness of Parallel TBox Classifier) *The proposed algorithms are complete for TBox classification.*

TBox classification based on top search and bottom search is complete in the sequential case. This means that the subsumption algorithms will find all subsumption relationships between concepts of a partition assigned to a single thread. The threads lock and unlock nodes whenever they are updating the information about a node in the global subsumption tree. Thus, we need to consider only the scenarios where two concepts C and D are inserted in parallel by different threads (e.g., thread#1 inserts concept C while thread#2 inserts concept D). In principle, if top (bottom) search pushed a new concept down (up), the information about children (parents) of a traversed node E could be incomplete because another thread might later add more nodes to the parents or children of E that were not considered when determining whether the concept being inserted subsumes or is subsumed by any of these newly added nodes. This leads to two scenarios that need to be examined for incompleteness.

W.l.o.g. we restrict our analysis to the case where a concept C is a parent of a concept D in the complete subsumption tree (CT). Let us assume that our algorithms would not determine this subsumption, i.e., in the computed (incomplete) tree (IT) the concept C is not a parent of D.

Case I: top_search incomplete for D: After D has been pushed down the tree IT as far as possible by top search (executed by thread#2) and top search has traversed the children of a concept E and E has become the parent of D, C is inserted by thread#1 as a new child of E. In line 6 of Algorithm 10 top_search is iteratively repeated for the concept *new* as long as consistent_in_top_search finds

a discrepancy between the captured and current successors of the parents of the newly inserted concept *new*. After finishing top and bottom search, Algorithm 10 checks again in lines 27-28 whether top search needs to be repeated due to newly added nodes. If any of the newly added children of D would subsume C and become a parent of C, the repeated execution of top_search would find this subsumption. This contradicts our assumption.

Case II: bottom_search incomplete for C: After C has been pushed up the tree IT as far as possible by bottom search (executed by thread#1) and bottom search has traversed the parents of a concept E and E has become a child of C, D is inserted by thread#2 as a new parent of E. In line 16 of Algorithm 10 bottom_search is iteratively repeated for the concept *new* as long as consistent_in_bottom_search finds a discrepancy between the captured and current predecessors of the children of the newly inserted concept *new*. After finishing top and bottom search, Algorithm 10 checks again in lines 32-33 whether bottom search needs to be repeated due to newly added nodes. If C would subsume any of the newly added parents of D and it would become a child of C, the repeated execution of bottom_search would find this subsumption. This contradicts our assumption.

This section explained the *sound and complete* algorithms for the Parallel TBox Classifier. The third generation of algorithms, Concurrent TBox Classifier, which is more efficient will be described in the next Section.

5.1.3 Third Generation

In Section 5.1.2, we introduced our algorithms for *sound and complete* parallel classification. In this section, the enhanced concurrent version of those algorithms are introduced. (i.e., Algorithms 15, 17 and 18); others are similar.

In this generation of Parallel TBox Classifier, similar to previous generations, to manage concurrency and multi-threading in the system, a single-shared global tree approach is used. Also, to avoid unnecessary tree traversals and tableau subsumption tests when computing the subsumption hierarchy, the parallel classifier adapts the enhanced traversal method in Section 3.1.3, Algorithms 7 and 11 outline the traversal procedures for the top-search phase, and bottom search is symmetric to top search.

Algorithm 14 informed_partitioning(*topological_sort_list*)

```
1:  atoms ← get-atomic-decomposition-atoms(owlfile)
2:  partition-counter ← 0
3:  ignore-list ← 0
4:  for all list ∈ topological_sort_list do
5:    for all c ∈ list do
6:      if ignore-list not contain c then
7:        ad-partition ← get-ad-partition-no(c, atoms)
8:        set informed-partitions(partition-counter) to ad-partition
9:        add ad-partition to ignore-list
10:       increment partition-counter
```

The procedure parallel_tbox_classification is also similar to the one sketched in Algorithm 9, and is called with a list of named concepts. After preprocessing of the list, topological order is generated with respect to the initial taxonomy created from already known told ancestors and descendants of each concept (using the told subsumer information). The sorted list is partitioned using either fixed-size partitions, or atomic decomposition (explained in Section 3.5.4), or informed

Algorithm 15 insert_partition(*partition,id*)

```
for all new ∈ partition do
  rerun ← 0
  finish_rerun ← false
  parents ← top_search(new,⊤)
  while ¬ consistent_in_top_search(parents,new) do
    parents ← top_search(new,⊤)
  predecessors(new) ← parents
  children ← bottom_search(new,⊥)
  while ¬ consistent_in_bottom_search(children,new) do
    children ← bottom_search(new,⊥)
  successors(new) ← children
  for all busy threads t_i ≠ id do
    located_concepts(t_i) ← located_concepts(t_i) ∪ {new}
  check ← check_if_concept_has_interaction(new, located_concepts(id))
  while (check ≠ 0) and ¬finish_rerun do
    if rerun < 3 then
      if check = 1 then
        new_predecessors ← top_search(new,⊤)
        rerun ← rerun + 1
        predecessors(new) ← new_predecessors
    else
      if check = 2 then
        new_successors ← bottom_search(new,⊥)
        rerun ← rerun + 1
        successors(new) ← new_successors
      check ←
      check_if_concept_has_interaction(new, located_concepts(id))
    else
      finish_rerun ← true
      for all busy threads t_i ≠ id do
        located_concepts(t_i) ← located_concepts(t_i) \ {new}
  if ¬finish_rerun then
    insert_concept_in_tbox(new, predecessors(new), successors(new))
```

partitioning (see Algorithm 14); (the partitioning methods will be explained in Section 5.2.5 and the corresponding evaluation of each partitioning method is documented in Section 6.4.6).

To improve the partitioning, Informed Partitioning algorithm was designed. Using Informed Partitioning, we had the hypothesis that number of re-run will be reduced, as the concepts which have interactions are placed in the same partition (Section 6.4.6 shows the preliminary evaluation of using Informed Partitioning).

As it is shown in Algorithm 14, a topological sorted list is passed as the input. In line 1 of the algorithm, the atomic decompositions are computed using a library provided by Manchester University (explained in Section 3.5.4). In Atomic Decomposition, the collections of signatures of atoms are usually overlapping. In other words, the atoms' signatures in Atomic Decomposition are not generally pairwise disjoint. Therefore, to avoid adding a concept redundantly into different informed-partitions, as it is shown in line 6 of Algorithm 14, a list named ignore-list is utilized. In line 6, everytime a concept c from Topological Sort Order list is going to be processed, it is checked if is already placed in ignore-list (e.g. added to previous partitions). If it is the first time that concept c is to be processed, then the first AD partition which contains c is returned (line 7). Then, the AD partition is added to the informed-partitions as a new partition (line 8). As it is shown in line 9, all the concepts of the new partition, are added to the ignore-list. The iteration continues till all the concepts in the Topological Sort Order list are processed and therefore, all the partitions for the informed-partitions are generated.

Then, classifier assigns in a round-robin manner partitions to idle threads and activates these threads with their assigned partition using the procedure insert_partition outlined in Algorithm 15. All threads work in parallel with the goal to construct a global subsumption tree (taxonomy). They also share a global array *located_concepts* indexed by thread identifications. The *located_concepts* array is used when a thread has inserted a concept into the subsumption tree and wants to

notify other threads about the concept.

The procedure insert_partition inserts all concepts of a given partition into the global taxonomy. However, for updating a concept or its parents or children, no locking mechanism is used. Therefore, all the assignments in Algorithm 15 are lock-free assignments.

Algorithm 15 first performs for each concept *new* the top-search phase (starting from the top concept ($\top$)) and possibly repeats the top-search phase for *new* if other threads updated the list of children of its parents. Then, it sets the parents of *new*. Afterwards the bottom-search phase (starting from the bottom concept ($\bot$)) is performed. Analogously to the top-search phase, the bottom search is possibly repeated and sets the children of *new*. After finishing the top and bottom search for *new*, the node *new* is added to the entries in *located_concepts* of all other busy threads; it is also checked whether other threads updated the entry in *located_concepts* for this thread. If this was the case, the top or bottom search need to be repeated correspondingly.

To prevent redundant re-runs, it only runs twice. If the concept *new* is still not ready to be inserted; e.g., there is any interaction between *new* and a concept in *located_concepts*; it will be added to the partition list of concepts (to be located later), and also eliminated from the other busy threads' *located_concepts* list, otherwise, *new* can be inserted into the taxonomy using Algorithm 18. Hence, for cases that we need more re-runs, as there are lots of interaction, after re-running for the second time, we postpone it and the concept is added to the partition list of concepts, in order to be inserted later.

The scenarios explained in Section 5.1.2, are properly addressed in Algorithm 15 to ensure completeness. Every time a thread locates a concept in the

Algorithm 16 consistent_in_top_search(*parents*,*new*)

```
for all pred ∈ parents do
  if successors(pred) ≠ captured_successors(new)(pred) then
    diff ← successors(pred) \ captured_successors(new)(pred)
    for all child ∈ diff do
      if found_in_ancestors(child,new) then
        return false
return true
```

taxonomy, it notifies the other threads by adding this concept name to their "located_concepts" list. Therefore, as soon as a thread finds the parents and children of the concept *new* by running top_search and bottom_search; it checks if there is any interaction between concept *new* and the concepts located in the "located_concepts" list. Based on the interaction, top_search or bottom_search needs to be repeated accordingly. If no possible situations for incompleteness are discovered anymore, Algorithm 18 is called. To resolve the possible incompleteness we utilize Algorithms 16 and 17. Algorithm 17 is the optimized version of Algorithm 13. The procedure consistent_in_bottom_search is not shown here because it mirrors consistent_in_top_search.

As mentioned earlier, beside using fixed-partition size, Concurrent TBox Classifier (e.g. Parallel TBox Classifier - Third generation) also utilizes atomic decomposition partitions as well as informed partitioning.

In this section, the *sound and complete* algorithms for the Concurrent TBox Classifier were explained and the evaluation will be described in Section 6.4. In the next section, the implemented prototype will be elaborated.

Algorithm 17 check_if_concept_has_interaction(*new,located_concepts*)

The return value indicates whether and what type of re-run needs to be done:
0 : No re-run in needed
1 : Re-run TopSearch because a possible parent could have been overlooked
2 : Re-run BottomSearch because a possible child could have been overlooked

```
if located_concepts = ∅ then
  return 0
else
  for all concept ∈ located_concepts do
    if interaction_possible(new,concept) then
      if found_in_ancestors(new,concept) then
        return 2
      else
        return 1
    else if interaction_possible(concept,new) then
      if found_in_ancestors(new,concept) then
        return 2
      else
        return 1
  return 0
```

Algorithm 18 insert_concept_in_tbox(*new,predecessors,successors*)

for all *pred* ∈ *predecessors* **do**
  successors(*pred*) ← successors(*pred*) ∪ {*new*}
for all *succ* ∈ *successors* **do**
  predecessors(*succ*) ← predecessors(*succ*) ∪ {*new*}

Algorithm 19 interaction_possible(*new*,*concept*)

Uses pseudo model merging information [23], pre-computed by Racer, to decide whether a subsumption is possible between *new* and *concept*.

Algorithm 20 found_in_ancestors(*new*,*concept*)

Checks if *concept* is an ancestor of *new*.

5.2 A Prototype for Parallel TBox Classifier

In this section, the implemented prototype for Parallel TBox Classifier will be explained.

The input of the prototype is a file generated by Racer. The file contains a list of concept names to be classified and information about them. The per-concept information available in the file includes its name, parents (in the complete taxonomy), told disjoints, told subsumers, and pseudo model [23] information. The information about parents is used to compute the set of ancestors and descendants of a concept. Told information consists of disjoints and subsumers.

The told disjoints' information is not utilized due to time constraints. The information was provided by Racer in order to be used for optimization techniques in pruning the classification tree. However, the information is not instrumental to parallelization procedure.

The told subsumers information is used for creating the initial taxonomy for generating a topological sort list.

The ancestors' (descendants') information is only used for (i) emulating a tableau subsumption test, i.e., by checking whether a possible subsumer (subsumee) is in the list of ancestors (descendants) of given concept, and (ii) in order to verify the completeness of the taxonomy computed by the parallel classifier. This information substitutes for an implemented tableau reasoning procedure. In other words, using the pre-computed parents information by Racer was deliberately designed to avoid implementing the tableau subsumption tests.

The output of the prototype is an XML file, which includes the computed statistics during Parallel TBox Classification.

The prototype has seven main components as follows.

5.2.1 Ontology Loader

The Ontology Loader lets Parallel TBox Classifier accept an ontology file generated by Racer. One can load an ontology by selecting a file through setting the corresponding name and path in the configuration file.

5.2.2 Configuration Manager

The Configuration Manager allows Parallel TBox Classifier to get user preferences before running classification. The name of the ontology to be processed as well as the control parameters are set in a configuration file. The Control Parameters are:

- **Number of Threads**
- **Partition Size** Number of concepts assigned to every thread and are expected to be inserted by corresponding thread.
- **Number of Processors**

Users can also select the strategies that will be used to partition a given set of concepts (e.g. Atomic Decomposition) as well as the logging level to be utilized by serializer.

5.2.3 Parser

The Parser parses the ontology input file generated by Racer.

5.2.4 Preprocessor

The Preprocessor utilizes told subsumer information and generates a topological order list (e.g., using a depth first traversal). In the topological order list, from left to right, parent concepts precede their children concepts.

5.2.5 Partition Manager

Partition Manager gets a concept list and partitions it. Based on the partitioning strategies defined in the configuration file, the following scenarios can be chosen for partitioning.

Fixed-size Partitioning

In this scenario, Partition Manager divides the Topological Sorted List of concepts from left to right into fixed-size partitions. The partition size is provided by Configuration Manager.

Dynamic-size Partitioning

Using Dynamic-size partitioning, Partition Manager partitions the Topological Sorted List of concepts from left to right, using partition size is provided by Configuration Manager. The partition size is increased dynamically.

Atomic Decomposition Partitioning

In Atomic Decomposition, Partition Manager uses a library provided by Manchester University, which applies Atomic Decomposition and therefore, creates partitions based on the computed atoms. The information about Atomic Decompo-

sition was explained in Section 3.5.4. Using Atomic Decomposition partitioning, no preprocessing is done and the partition size is not fixed, as the partition size is mandated by the atom size.

Informed Partitioning

Partition Manager uses Algorithm 14 to generate Informed Partitioning partitions. Here, preprocessing is done and the order in the Topological Sorted List of concepts is utilized for ordering the computed atoms, generated by Manchester University's library (see Atomic Decomposition explanation in Section 3.5.4).

5.2.6 Serializer

Serializer allows the information to be logged into the files. The level of logging is provided by Configuration Manager. Serializer also serializes the statistics computed during the classification.

5.2.7 TBox Classifier

TBox Classifier is the core of the prototype and it classifies the TBox in parallel using a multi-threaded architecture. The classifier utilizes the partitions generated by Partition Manager and assigns them to threads (See Algorithms 15, 16, 17, and 18).

In the third generation of TBox Classifier, we use Concurrent collections from the java.util.concurrent package. This package supplies Collection implementations which are thread-safe and designed for use in multi-threaded contexts. Using the Concurrency package in Java, synchronization on the nodes of the global tree

as well as the entries in the global array have been eliminated. Therefore, for updating a concept or its parents or children, no locking mechanism for the affected nodes of the global tree is needed anymore.

Chapter 6

Performance Evaluation

The implementation of Parallel Classification of TBox has the following main objectives :

1. To show that efficient classification is possible in parallel.

2. The optimization techniques explained in Section 3.1.3 and designed for sequential execution, are also efficient for parallel classification, if extended correspondingly.

The objectives can be met by evaluating the Parallel TBox Classifier using real world ontologies.

The scalability of the Parallel TBox Classifier, is tested by using ontologies of different complexity and size. Note that the ontologies of generally large size (hundreds of thousands of concepts) are also considered, although the size greatly affects the complexity of classification, the prototype is able to tackle such a complexity.

6.1 Benchmarks

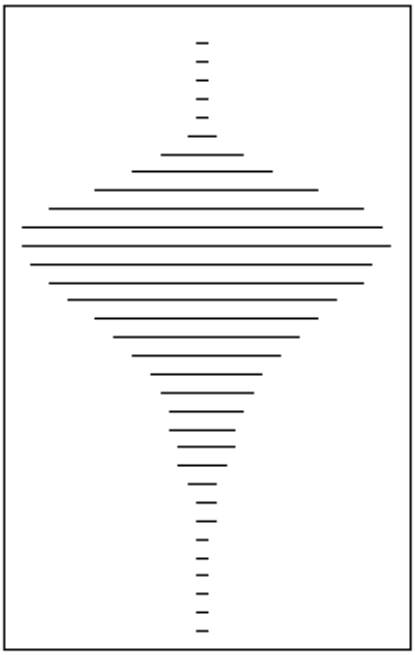
The benchmarks used are publicly available real world ontologies. These are the typical benchmarks utilized by Racer and other reasoners on a regular basis. Note that the chosen test cases exhibit different sizes, structures, and DL complexities. In this section, the state of the art ontologies as well as their Xmas trees[1] will be explained.

6.1.1 Cyc

The Cyc knowledge base (KB)[2] is a formalized representation of a vast quantity of fundamental human knowledge: facts, rules of thumb, and heuristics for reasoning about the objects and events of everyday life. The KB consists of terms and assertions which relate those terms. These assertions include both simple ground assertions and rules. Cyc is not a frame-based system: the Cyc team thinks of the KB instead as a sea of assertions, with each assertion being no more ”about” one of the terms involved than another. The Cyc KB is divided into ”microtheories”, each of which is essentially a bundle of assertions that share a common set of assumptions; some microtheories are focused on a particular domain of knowledge, a particular level of detail, a particular interval in time, etc. Figure 6.1 shows the taxonomy. In the figure, the length of the line per level is proportional to the number of concepts in this tree level.

[1] Xmas trees were designed to show the shape of taxonomies. In the tree, from top to bottom the number of concepts on each level is represented by a line with a width scaled to the maximum number of concepts over all level. All levels of the taxonomy are shown in the tree.

[2] http://www.cyc.com/cyc/technology/technology/whatiscyc_dir/whatsincyc



DL Expressivity	# of Concepts
$\mathcal{ALH}(D)$	25,566

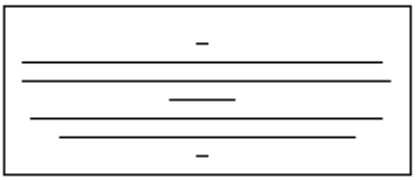
Figure 6.1: CYC Ontology.

6.1.2 eCl@ss

The EClass ontology[3] is part of eClassOWL. eClassOWL is an OWL ontology for describing the types and properties of products and services on the Semantic Web (also known as the ”Web of Linked Data”). eClassOWL is meant to be used in combination with the GoodRelations ontology for e-commerce, which covers the commercial aspects of offers and demand, e.g. prices, payment, or delivery options.

eClassOWL is a project that has been initiated by Martin Hepp in 2003 and is now being hosted and maintained by the E-Business and Web Science Research Group at the Universität der Bundeswehr München.

[3]http://notes.3kbo.com/eclassowl



DL Expressivity	# of Concepts
$\mathcal{AL}(D)$	76,975

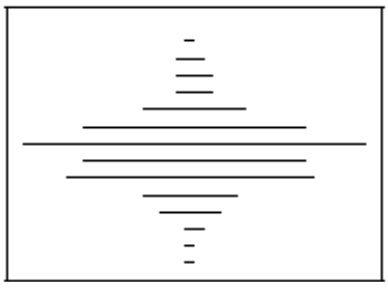
Figure 6.2: ECLASS Ontology.

eCl@ss covers a wide range of products and services. However, when we want to use such categorization standards for the Semantic Web, we have to make sure that we properly understand the meaning and structure of the existing standard and create a consistent and correct transformation. This work has already been done in the case of eCl@ss. Figure 6.2 shows the taxonomy.

6.1.3 Embassi

EMBASSI [4] is in the framework of the program MTI (interaction in the knowledge society), which during the four-year term, 7 research and university institutes as well as 12 companies involved in the projects. Overall, EMBASSI (Multimodal operator and service assistant) is used for the development of an intelligent user assistance that will allow people a natural, intuitive control of electronic devices. The aim is to achieve a flexible, extensible and configurable architecture so that, for example a user equipment from different manufacturers can always operate in the same manner. In 2001, significant improvements to structure, stability, maintainability and performance of the scenario-demonstrators in private household and automotive applications have been achieved. Figure 6.3 shows the taxonomy.

[4] http://univis.uni-erlangen.de/formbot/dsc_3Danew_2Fresrep_view_26rprojs _3Dtech_2FIMMD_2FIMMD8_2Fembass_26dir_3Dtech_2FIMMD_2FIMMD8_26ref_3Dresrep



Variations	DL Expressivity	# of Concepts
EMBASSI-1	$\mathcal{ALCF}$	313
EMBASSI-2	$\mathcal{ALCF}$	731
EMBASSI-3	$\mathcal{ALCF}$	1,178

Figure 6.3: EMBASSI Ontology.

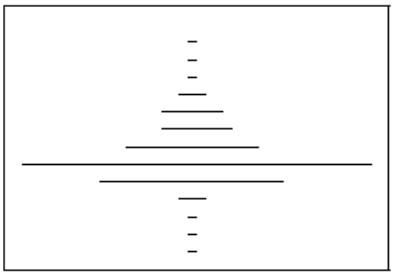
6.1.4 Fungal Web

The FungalWeb Ontology [41] is a formal ontology in the domain of fungal genomics, which provides a semantic web infrastructure for sharing knowledge using four distinct sub-ontologies: enzyme classification based on their reaction mechanism, fungal species, enzyme substrates and industrial applications of enzymes. The ontology was developed in OWL-DL by integrating numerous online textual resources, interviews with domain experts, biological database schemas and reusing some existing bio-ontologies, such as GO and TAMBIS. Figure 6.4 shows the taxonomy.

6.1.5 Generalized Architecture for Languages, Encyclopedia and Nomenclatures in medicine (GALEN)

Generalized Architecture for Languages, Encyclopedia and Nomenclatures in medicine (GALEN)[5] has been modeled to represent clinical information to support clinicians and is intended to generate a formal multilingual coding system for medicine.

[5]http://www.openclinical.org/prj_galen.html



DL Expressivity	# of Concepts
$\mathcal{ALCHOIN}(\mathcal{D})$	3,601

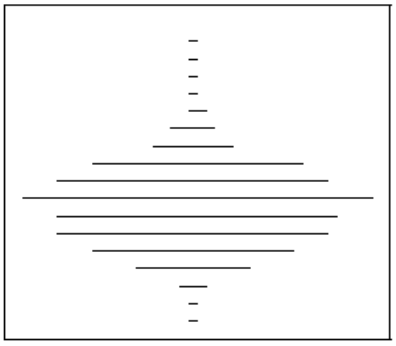
Figure 6.4: FungalWeb Ontology.

It originally evolved from the Pen&Pad electronic medical record system, which was modeled using Structured Meta Knowledge (SMK), in the way that terms were described through relationships to other terms. The core of GALEN is an ontology, the Common Reference Model, formulated in a specialized description logic, GRAIL, that does not support the use of disjunction or negation. GALEN has been employed as a basis for studying nursing terminologies, surgical vocabularies, anatomy, and decision support systems. The major strengths of GALEN are the formal representation of clinical information and the use of a formal structure based on description logic. GALEN also allows multiple views of relevant detail as needed. Figure 6.5 shows the taxonomy.

6.1.6 Gene Ontology (GO)

The Gene Ontology (GO) project[6] is a collaborative effort to address the need for consistent descriptions of gene products in different databases. The project began

[6]http://www.geneontology.org/GO.doc.shtml



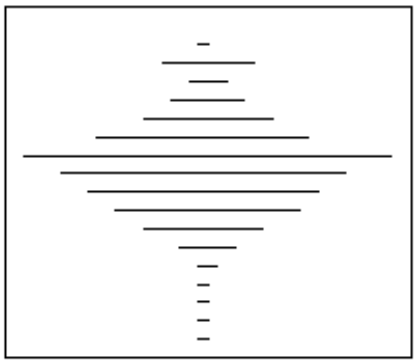
Variations	DL Expressivity	# of Concepts
GALEN	$\mathcal{SHF}$	2,730
GALEN1	$\mathcal{ALC}$	2,730
GALEN2	$\mathcal{ALE}$	3,926

Figure 6.5: GALEN Ontology.

as a collaboration between three model organism databases, FlyBase (Drosophila), the Saccharomyces Genome Database (SGD) and the Mouse Genome Database (MGD), in 1998. Since then, the GO Consortium has grown to include many databases, including several of the world's major repositories for plant, animal and microbial genomes.

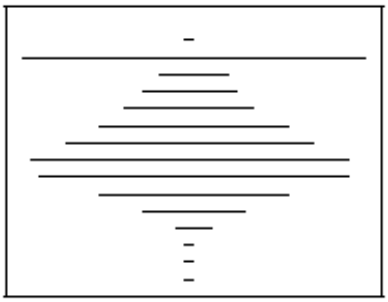
The Gene Ontology project provides an ontology of defined terms representing gene product properties. The ontology covers three domains: cellular component, the parts of a cell or its extracellular environment; molecular function, the elemental activities of a gene product at the molecular level, such as binding or catalysis; and biological process, operations or sets of molecular events with a defined beginning and end, pertinent to the functioning of integrated living units: cells, tissues, organs, and organisms.

The GO ontology is structured as a directed acyclic graph, and each term has defined relationships to one or more other terms in the same domain, and sometimes to other domains. The GO vocabulary is designed to be species-neutral, and in-



DL Expressivity	# of Concepts
$\mathcal{ALE}+$	22,566

Figure 6.6: GO Ontology.



DL Expressivity	# of Concepts
$\mathcal{ALE}+$	5,584

Figure 6.7: LargeTest Ontology.

cludes terms applicable to prokaryotes and eukaryotes, single and multicellular organisms. Figure 6.6 shows the taxonomy.

6.1.7 LargeTest Ontology

The LargeTest Ontology is a variant/extract of Gene Ontology(GO). Figure 6.7 shows the taxonomy.

6.1.8 Systematized Nomenclature of Medicine Clinical Terms (SNOMED CT)

SNOMED Clinical Terms (SNOMED CT)[7] is the most comprehensive, multilingual clinical healthcare terminology in the world.

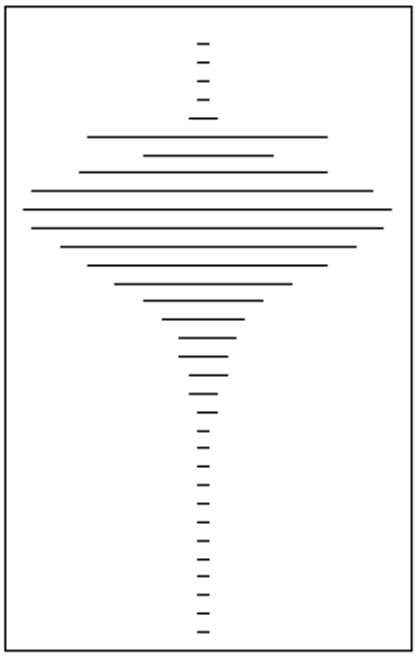
SNOMED CT contributes to the improvement of patient care by underpinning the development of Electronic Health Records that record clinical information in ways that enable meaning-based retrieval. This provides effective access to information required for decision support and consistent reporting and analysis. Patients benefit from the use of SNOMED CT because it improves the recording of EHR information and facilitates better communication, leading to improvements in the quality of care.

SNOMED CT is owned, maintained and distributed by the International Health Terminology Standard Development Organisation (IHTSDO). The IHTSDO is a not-for-profit association which is owned and governed by its national Members. In January 2012 eighteen countries were Members of IHTSDO, more countries are joining every year.

SNOMED CT is

- a clinical healthcare terminology
- a resource with comprehensive, scientifically-validated content
- essential for electronic health records
- a terminology that can cross-map to other international standards
- already used in more than fifty countries.

[7] http://www.ihtsdo.org/snomed-ct/



Variations	DL Expressivity	# of Concepts
SNOMED-2	$\mathcal{ELH}$	182,869
SNOMED-1	$\mathcal{ELH}$	223,260
SNOMED	$\mathcal{ELH}$	379,691

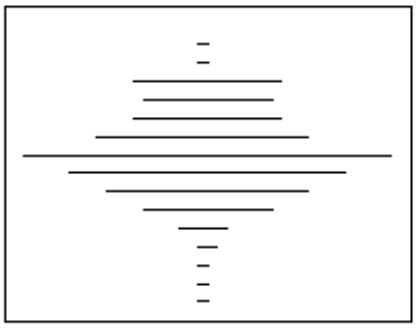
Figure 6.8: SNOMED Ontology.

SNOMED CT provides the core general terminology for the electronic health record (EHR). When implemented in software applications, SNOMED CT can be used to represent clinically relevant information consistently, reliably and comprehensively as an integral part of producing electronic health records.

6.1.9 Transparent Access to Multiple Bioinformatics Information Sources Ontology (TAMBIS)

TAMBIS[8] was a joint research project between the School of Biological Sciences and the Information Management Group, part of Computer Science in the Univer-

[8]http://www.cs.man.ac.uk/ stevensr/tambis/text/details.html



Variations	DL Expressivity	# of Concepts
TAMBIS-0	$\mathcal{ALEH}+$	340
TAMBIS-2a	$\mathcal{ALEH}+$	10,114

Figure 6.9: TAMBIS Ontology.

sity of Manchester in the UK. TAMBIS aims to aid researchers in biological science by providing a single access point for biological information sources round the world.

TAMBIS's[9] ontology enables biologists to ask questions over multiple external databases using a common query interface. The TAMBIS ontology (TaO) describes a wide range of bioinformatics tasks and resources, and has a central role within the TAMBIS system.

An interesting feature of TaO is that it does not contain any instances. The TaO only contains knowledge about bioinformatics and molecular biology concepts and their relationships - the instances they represent still reside in the external databases.

The TaO is a dynamic ontology, in that it can grow without the need for either conceptualizing or encoding new knowledge. Figure 6.9 shows the taxonomy.

[9]http://www.cs.man.ac.uk/ stevensr/onto/node10.html

Variations	DL Expressivity	# of Concepts
UMLS-1	$\mathcal{ALCHIN}$	297
UMLS-2	$\mathcal{ALCHIN}$	9,479
UMLS-3	$\mathcal{ALCHIN}$	160,035

Figure 6.10: UMLS Ontology.

6.1.10 Unified Medical Language System (UMLS)

The Unified Medical Language System (UMLS)[10] was created by the National Library of Medicine (NLM) to facilitate the development of computer systems that behave as if they ”understand” the meaning of the biomedicine/health language. To that end, the NLM produces and distributes the UMLS knowledge sources (databases) and associated software tools (programs) to system developers for use in informatics research and in building or enhancing electronic information systems that create, process, retrieve, integrate, and aggregate biomedical/health data and information. The UMLS Knowledge Sources are multipurpose, and can utilize a variety of data and information, such as patient records, scientific literature, guidelines and public health data. Figure 6.10 shows the taxonomy.

6.2 Evaluation of Parallel TBox Classifier - First Generation

To evaluate the adequacy of ”Parallel TBox Classifier - First Generation”, explained in Section 5.1.1, and also to assess the performance of the corresponding

[10]http://www.nlm.nih.gov/pubs/factsheets/umls.html

Table 6.1: Used test ontologies.

Ontology Name	DL Expressivity	No. of named concepts
Galen	$\mathcal{SHF}$	2,730
Galen1	$\mathcal{ALC}$	2,730
Umls-2	$\mathcal{ALCHIN}$	9,479

algorithms, the prototype is configured on a Intel Single Core Dell laptop using 1 GHZ memory, so that it runs various experiments over ontologies with various sizes, complexity and expressivity.

Two types of experiments were conducted. In the first experiment, Preprocessor generates a topological sort order using a corresponding algorithm. However, in the second experiment, Preprocessor randomly shuffle the generated topological order, using a custom random shuffle algorithm, and thereafter the shuffled list is partitioned. Once the shuffle order has been computed, it is saved and later reused for more tests.

The experiments were conducted on three different scenarios (will be explained in Sections 6.2.1, 6.2.2, and 6.2.3) using the ontologies listed in Table 6.1. The ontologies of the benchmark were explained in Section 6.1.

The two control parameters which influence the ”Parallel TBox Classifier - First Generation”, namely partition size, and the number of threads, vary in the following three scenarios. The results are measured on the basis of the number of missed subsumptions in the taxonomy produced by Parallel Classifier vs. the complete taxonomy.

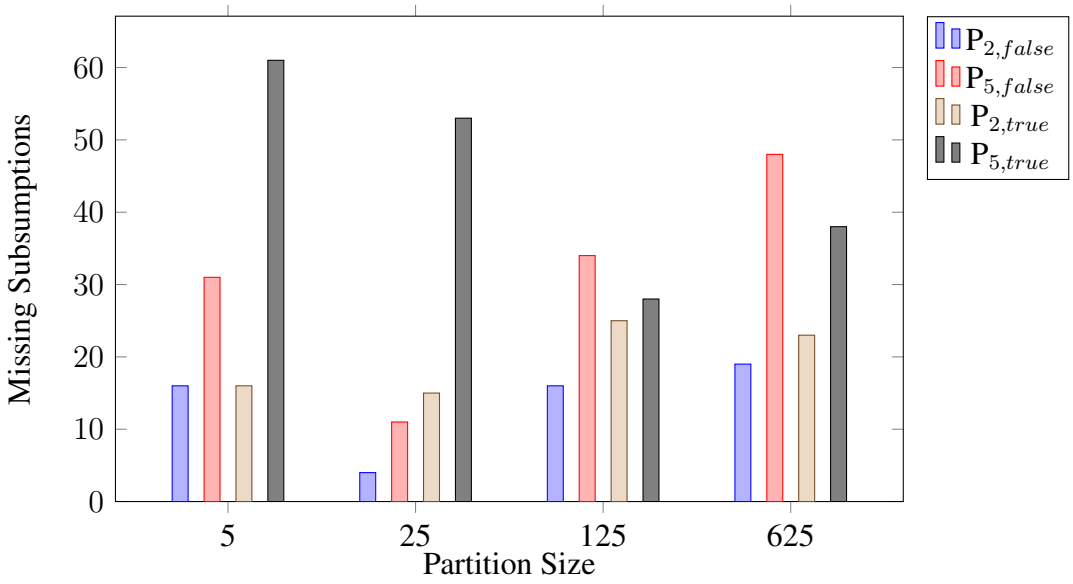


Figure 6.11: Scenario 1: Missing Subsumptions for Galen using 4 settings: $P_{2,false}$, $P_{5,false}$, $P_{2,true}$, $P_{5,true}$ ($P_{threads,shuffle}$).

6.2.1 Partition size and number of threads are constant

In this scenario, Partition Manager divides the topological order list into fixed-size partitions. After each test the partition size is increased by a factor of 5. As shown in Figure 6.11, when the system is configured to run with 2 threads, the number of missing subsumptions decreases and then slightly increases as the partition size increases. Running the simulation with 5 threads, the number of missing subsumptions decreases when the topological order is not shuffled; however, if it is shuffled the missing subsumptions decreases and then dramatically increases as the partition size increases. In the worst case, the missing subsumptions make 2% of all detected subsumptions.

Figure 6.12 shows the ratio for passed subsumption tests[11] and failed sub-

[11]Total number of passed (successful) subsumption tests in the parallel case divided by the total number of passed subsumption tests in the sequential case.

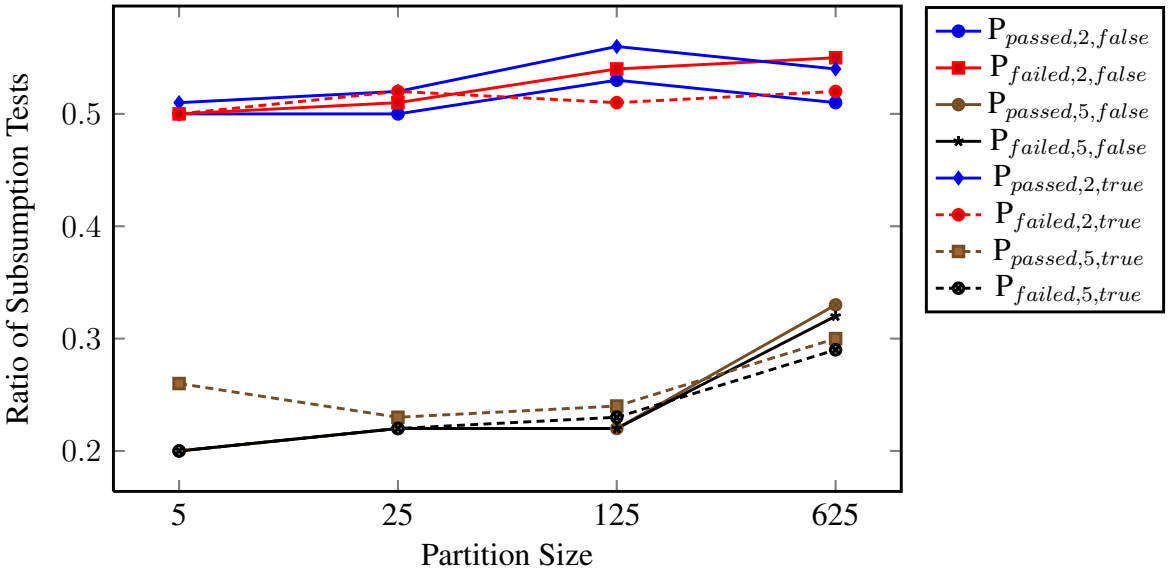


Figure 6.12: Scenario 1: Ratio of passed and missed subsumption tests for Galen using 8 settings: $P_{passed,2,false}$, $P_{failed,2,false}$, $P_{passed,5,false}$, $P_{failed,5,false}$, $P_{passed,2,true}$, $P_{failed,2,true}$, $P_{passed,5,true}$,$P_{failed,5,true}$ ($P_{subsumptiontest,threads,shuffle}$).

sumption tests.[12] The intention of the ratio is to measure the effect of parallelism on number of subsumption tests (e.g. passed, failed) vesus sequential. As shown in this figure, the increase of the number of threads results a better ratio. In other words, when more threads are added, the ratio decreases. Conducting the same tests for Umls-2 one gets at most 1% of missed subsumptions and the ratio has the same trend as for Galen.

6.2.2 Partition size is dynamic and grows exponentially and the number of threads is constant

In this scenario, the partition size grows exponentially (5^n), however the number of threads remains constant in each test run. As displayed in Figure 6.13, in all the

[12]Total number of failed (unsuccessful) subsumption tests in the parallel case divided by the total number of failed subsumption tests in the sequential case.

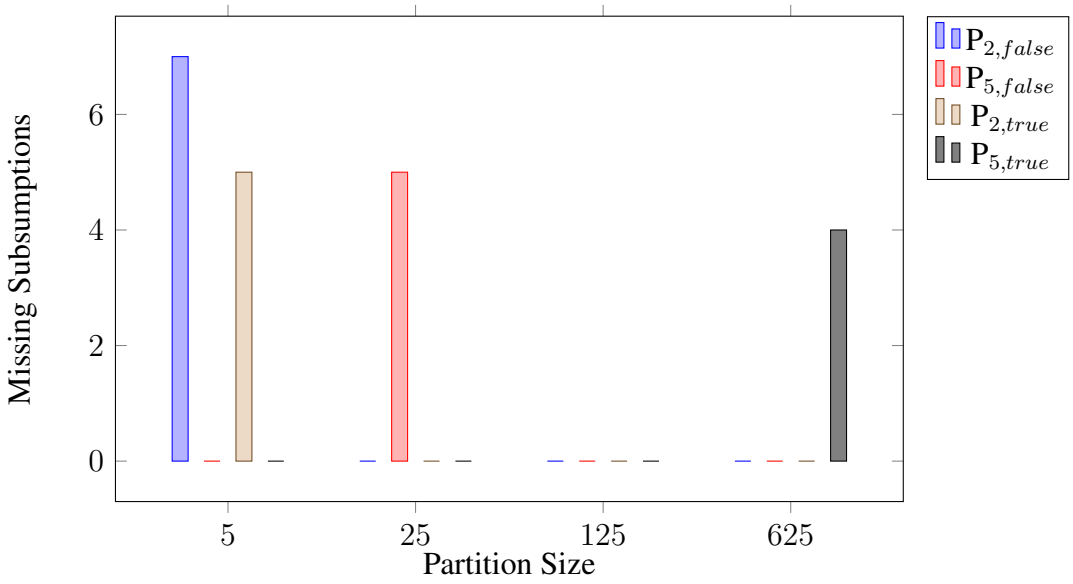


Figure 6.13: Scenario 2: Missing Subsumptions for Galen using 4 settings: $P_{2,false}$, $P_{5,false}$, $P_{2,true}$, $P_{5,true}$ ($P_{threads,shuffle}$).

cases the missing subsumptions decrease except for the case when the topological order list is shuffled and the number of threads is equal to 5. The percentage of missing subsumptions in the worst case is 0.25%. When the partition size increases, the ratio (e.g. passed, failed subsumption tests) is close to 1. This means that the number of passed and failed subsumption tests is similar to the sequential case. For Umls-2 the number of missing subsumptions is 0.02% and the ratio follows a trend close to Galen.

6.2.3 Number of threads is dynamic and grows exponentially

In this scenario, the number of threads grows exponentially (2^n) but the partition size remains constant. Figure 6.15 shows the test results for the cases when a topological and a random order is used. In this figure, it is observed that the number of missing subsumptions goes up and down based on the number of threads.

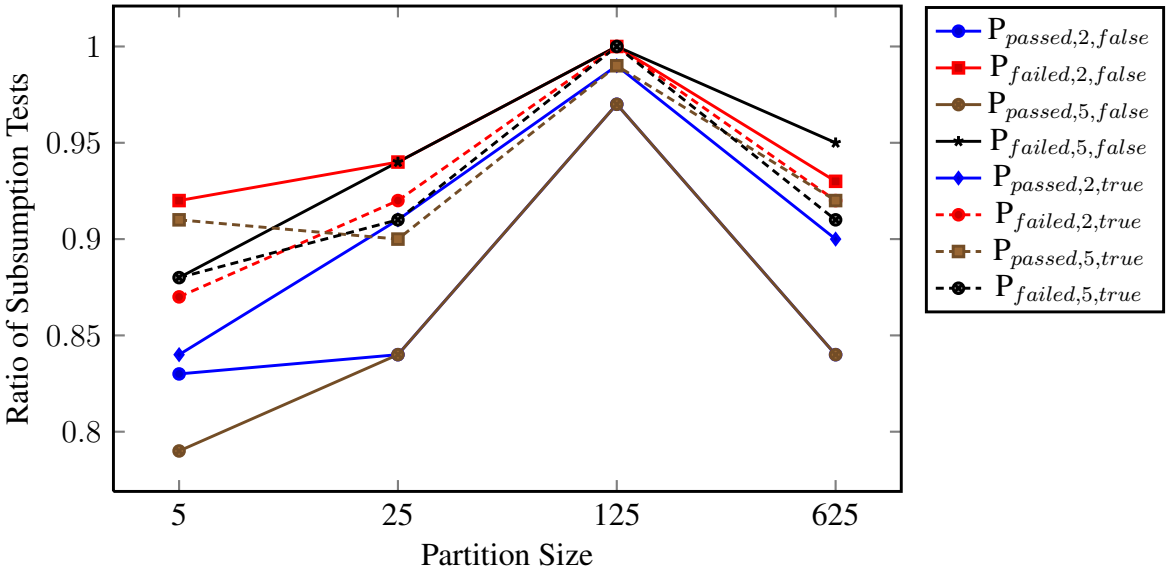


Figure 6.14: Scenario 2: Ratio of passed and missed subsumption tests in Galen using 8 settings: $P_{passed,2,false}$, $P_{failed,2,false}$, $P_{passed,5,false}$, $P_{failed,5,false}$, $P_{passed,2,true}$, $P_{failed,2,true}$, $P_{passed,5,true}$,$P_{failed,5,true}$ ($P_{subsumptiontest,threads,shuffle}$).

Figure 6.16 displays the ratio for passed and failed subsumption tests. This figure shows a smooth decrease of the ratio.

The experimental evaluation of the proposed algorithms for ”Parallel TBox Classifier - First Generation” shows that the results are very promising because the number of missed subsumptions is small. Due to missing subsumptions in classification, the algorithm is sound but incomplete. In the next section, the experimental results for the second generation of the prototype, ”Parallel TBox Classifier - Second Generation”, will be described.

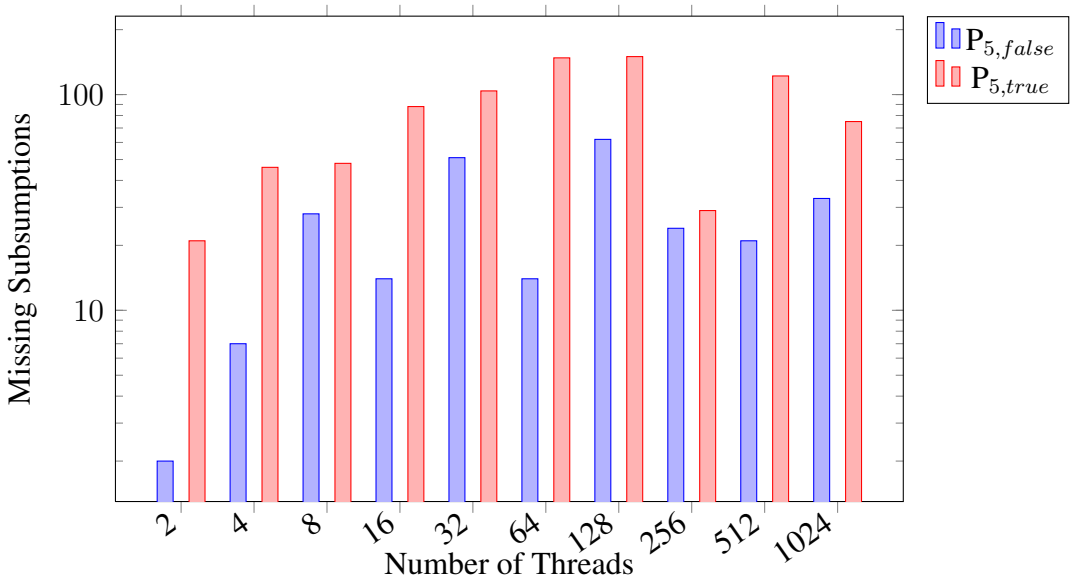


Figure 6.15: Scenario 3: Missing Subsumptions for Galen using 2 settings: $P_{5,false}$, $P_{5,true}$ ($P_{partitionsize,shuffle}$).

6.3 Evaluation of Parallel TBox Classifier - Second Generation

In this section, the adequacy of "Parallel TBox Classifier - Second Generation", explained in Section 5.1.2 will be evaluated. This generation of the prototype was tested on a high performance parallel computing cluster. The nodes in the cluster run an HP-version of RedHat Enterprise Linux for 64 bit processors, with HP's own XC cluster software stack.

The Parallel TBox Classifier has been developed to speed up the classification time especially for large ontologies by utilizing parallel threads sharing the same memory. Using Configuration Manager, the benchmark can be configured so that it runs various experiments over ontologies. The evaluation is done with a collection of 8 mostly publicly available ontologies. Their name, size in number of

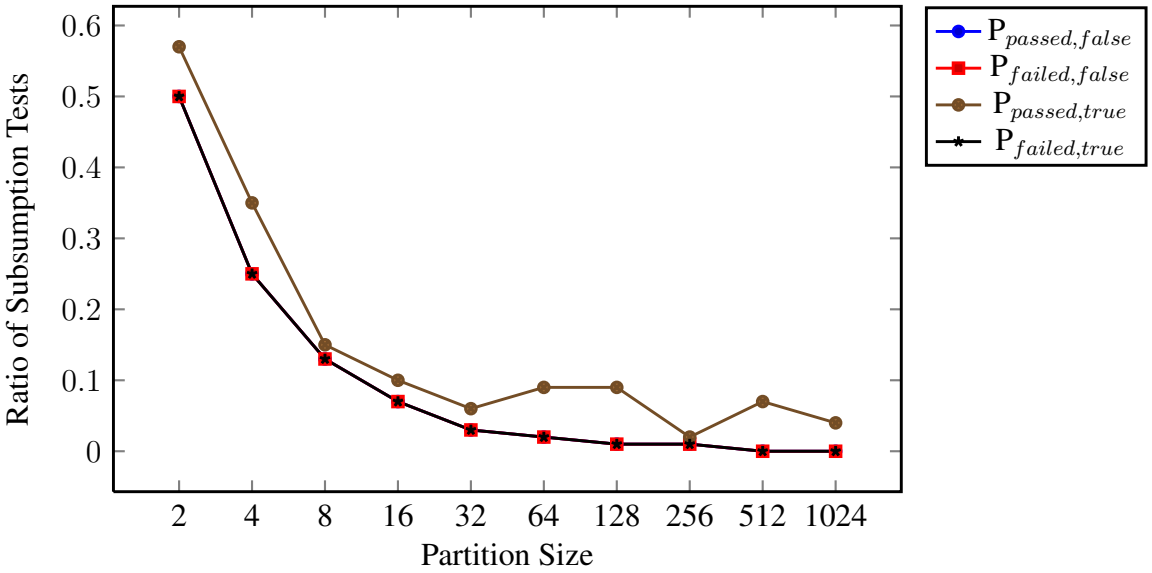


Figure 6.16: Scenario 3: Ratio of passed and missed subsumption tests in Galen using 4 settings: $P_{passed,false}$, $P_{failed,false}$, $P_{passed,true}$, $P_{failed,true}$ ($P_{subsumptiontest,shuffle}$).

Table 6.2: Characteristics of the used test ontologies.

Ontology name	DL language	No. of named concepts
Embassi-2	$\mathcal{ALCHN}$	657
Embassi-3	$\mathcal{ALCHN}$	1,121
Galen	$\mathcal{SHN}$	2,730
Galen1	$\mathcal{ALCH}$	2,730
Galen2	$\mathcal{ELH}$	3,928
FungalWeb	$\mathcal{ALCHIN}(\mathcal{D})$	3,603
Umls-2	$\mathcal{ALCHIN}(\mathcal{D})$	9,479
Tambis-2a	$\mathcal{ELH}$	10,116

named concepts, and used DL is shown in Table 6.2 (see Section 6.1 for further information).

Two control parameters, which are explained in Section 5.2.2 influence the parallel TBox classification, namely number of threads, and partition size; the partition size was set to 5 and number of threads to 2 in our empirical experiments.

To better compare the performance between the sequential and parallel case, it is assumed that every subsumption test runs in constant time *t1* and in the sequential and parallel case the same amount of time is used for an executed subsumption test. Subsumption tests can be expensive and, hence, are preferred to be avoided by optimization techniques such as pseudo model merging [22]. The ratio illustrated in Equation 6.1 uses $TotSubsTests_s$, the number of times a subsumption test was computed in the sequential case, and *MaxOfSubTestsInEachThread*, the maximum number of subsumption tests performed in all threads. Similarly, Equation 6.2 defines the overhead (where the index p refers to the parallel case).

$$Ratio = \frac{MaxOfSubTestsInEachThread}{TotSubsTests_s} \tag{6.1}$$

$$Overhead = \frac{TotSubsTests_p - TotSubsTests_s}{TotSubsTests_s} \tag{6.2}$$

Table 6.3 shows an excellent performance increase and a surprisingly small overhead when using the Parallel TBox Classifier. Using two threads the maximum of number of subsumption test for all ontologies could be reduced to roughly one half compared to the sequential case. The overhead as defined in Equation 6.2 varies between 0.13% and 2.62%. The overhead is mostly determined by the quality of the told subsumers , the imposed order of traversal within a partitioning, and the division of the ordered concept list into partitions using fixed-size partitioning. In general, one should try to insert nodes as close as possible to their final order in the tree using a top to bottom strategy.

The results for Parallel TBox Classifier, the Sound and Complete generation are promising, however, there is small overhead. As the benchmark shows, the

Table 6.3: Subsumptions tests and their ratio for the test ontologies.

	Embassi-2	**Embassi-3**	**Galen**	**Galen1**
Subs. Tests in sequent.	154,034	420,912	2,706,412	2,688,107
Subs. Tests in thread#1	76,267	217,324	1,363,321	1,367,302
Subs. Tests in thread#2	77,767	214,633	1,354,297	1,348,281
Worst Case Ratio	50.48%	51.63%	50.37%	50.86%
Overhead	1.64%	2.62%	0.41%	1.02%
	Galen2	**FungalWeb**	**Umls-2**	**Tambis-2a**
Subs. Tests in sequent.	5,734,976	4,996,932	87,423,341	36,555,225
Subs. Tests in thread#1	2,929,276	2,518,676	44,042,203	18,342,944
Subs. Tests in thread#2	2,893,716	2,490,329	44,025,988	18,261,532
Worst Case Ratio	51.07%	50.40%	50.37%	50.17%
Overhead	1.53%	0.24%	0.73%	0.13%

prototype could only classify the ontologies with up to 10,000 named concepts. To overcome this limitation, which was caused by design and architecture of the prototype, the third generation was designed and implemented, and the evaluation is documented in the following section.

6.4 Evaluation of Concurrent TBox Classifier - Third Generation

In the previous section, we explained the evaluation of algorithms used in ”Parallel TBox Classifier - Second Generation”.

In this section, the ”Concurrent TBox Classifier - Third Generation”, which the algorithms illustrated in Section 5.1.3, will be evaluated. Hence, the scalability and performance of the prototype will be examined; also, the behavior of the system when it is run in a (i) sequential or (ii) parallel multi-processor environment will be explained.

This section, also describes how the prototype performs when we have huge real-world ontologies with different DL complexities. The performance is also examined, using different schemes of partitions generated by Partition Manager (e.g. Fixed-Size partitions, Atomic Decomposition partitions, and Informed Partitioning partitions).

In the following, a description of the used platform and the implemented prototype are provided. Then, the benchmarks used to evaluate ”Concurrent TBox Classifier - Third Generation” are described and an overview of the parameters used in the experiments is explained. Finally, the results are shown and the performance of the classifier is discussed. In addition, the measured runtimes in the figures are shown in seconds using a logarithmic scale.

Platform and Implementation The experiments were conducted on a high performance parallel computing cluster. The nodes in the cluster run an HP-version of RedHat Enterprise Linux for 64 bit processors, with HP's own XC cluster software stack. The environment is same as “Parallel TBox Classifier - Second Generation”.

As explained in Section 5.2.7, Concurrent TBox Classifier has been implemented in Java using lock-free data structures from the java.util.concurrent package with minimal synchronization.

As discussed in Section 5.1.3, in Concurrent TBox Classifier no specific optimization techniques for classification have been implemented. For instance, there are well-known optimizations which can avoid subsumption tests or eliminate the bottom search for some DL languages or decrease the number of bottom searches in general (see Section 3.3). The system is not intended to be competitive compared to highly optimized DL reasoners or special-purpose reasoners designed to take

Table 6.4: Characteristics of the used test ontologies (e.g., $\mathcal{LH}$ denotes the DL allowing only conjunction and role hierarchies, and unfoldable TBoxes).

Ontology	DL language	No. of named concepts
Embassi-2	$\mathcal{ALCHN}$	657
Galen1	$\mathcal{ALCH}$	2,730
LargeTestOntology	$\mathcal{ELHR}+$	5,584
Tambis-2a	$\mathcal{ELH}$	10,116
Cyc	$\mathcal{LHF}$	25,566
EClass-51En-1	$\mathcal{LH}$	76,977
Snomed-2	$\mathcal{ELH}$	182,869
Snomed-1	$\mathcal{ELH}$	223,260
Snomed	$\mathcal{ELH}$	379,691

advantage of the characteristics of the $\mathcal{EL}$ fragment (e.g., see [28]), however, it can easily classify ontologies that are outside of the $\mathcal{EL}$ fragment.

Test Cases Table 6.4 shows a collection of 9 mostly publicly available real-world ontologies, which were described in Section 6.1. Note that the chosen test cases exhibit different sizes, structure, and DL complexities. The benchmark ontologies are characterized by their name, size in number of named concepts or classes, and used DL.

Parameters Used in Experiments The parameters used in our empirical evaluation and their meaning are described below (the default parameter value in shown in bold).

- *Number of Threads*: To measure the scalability of our system, we have performed our experiments using different numbers of threads (**1**, 2, 4, 6, 8, 16).

- *Partition Size*: The number of concepts (**5**, 25, 65, 125) that are assigned to every thread and are expected to be inserted by the corresponding thread.

Similar to the number of threads, this parameter is also used to measure the scalability of our approach.

- *Number of Processors*: For the presented benchmarks we always had 8 processors or cores[13] available.

In the following experiments, $\mathrm{P}_{threads,partition\_size}$ is used to indicate a parallel multi-core setting where the subscripts give the number of threads created, and the partitions size used (from left to right). These parameters are set exponentially to the bases of $\mathrm{P}_{2,5}$ using the exponent starting from 1 (e.g. $\mathrm{P}_{2,5}$, $\mathrm{P}_{4,25}$, etc). This setting was chosen for the historic reasons.

Performance In order to test the effect of these parameters in the system, the benchmarks are run with different parameter values. The performance improvement is measured using the speedup factor which is defined as $Speedup_p = \frac{T_1}{T_p}$, where $Speedup_p$ is the speedup factor or efficiency, and

- p is the number of threads. In the cluster environment, 8 cores are always available and in most of the experiments more than 8 threads are not used, so, each thread can be considered as mapped to one core exclusively.

- T_1 is the CPU time for the sequential run using only one thread and one single partition containing all concept names to be inserted;

- T_p is the CPU time for the parallel run with p threads as maximum over all threads.

[13]For ease of presentation we use the terms *core* and *processor* as synonyms here.

Ideally, speedup should scale linearly with p, means that doubling the number of processors/threads doubles the speed.

Superlinear speedup Superlinear speedup occurs when the speedup grows faster than the number of processors/threads. Surely, the sequential program, which is the basis for the speedup computation, could just simulate the p processes of the parallel program to achieve an execution time that is no more than p times the parallel execution time. There are reasons why a superlinear speedup occurs.

The most common reason is that the computations working set - that is, the set of pages needed for the computationally intensive part of the program - does not fit in the cache when executed on a single processor, but it does fit into the caches of the multiple processors when the problem is divided amongst them for parallel execution. In other words, with the larger accumulated cache size, more or even all of the working set can fit into caches and the memory access time reduces dramatically, which causes the extra speedup in addition to that from the actual computation. In such cases the superlinear speedup derives from improved execution time due to the more efficient memory system behavior of the multi-processor execution.

Also, a good use of cluster has to do with managing very large data sets. Hence, when the data set to be crunched is sufficiently large that it cannot reside in a single machines RAM and if the computation can be spread across multiple CPUs so a subset of the total data can fit in a single machines RAM, then a ”superlinear speedup” is possible. Therefore, a superlinear speedup can occur for large data given the effect of memory and disk swapping. In other words, if one has 8 processors in a cluster, the performance of the cluster could be more than 8 times faster

than that of a single processor due to the software overhead involved in memory swapping in the sequential case.

6.4.1 Effect of changing only the number of threads

To measure the performance of the classifier in this case, EClass-51En-1 was selected as the test case and the tests were run with fixed-size partitions (5 or 25) and a different number of threads (2 and 4), as shown in Figure 6.17. In the test cases $P_{2,5}$ and $P_{4,5}$, an ideal speedup was achieved that is proportional to the number of threads, as shown in Figure 6.18. As it is shown, doubling the number of threads from S to $P_{2,5}$ and to $P_{4,5}$, each time doubles the speedup, in other words, decreases the CPU time by the number of threads.

Comparing the test cases S, $P_{2,25}$, and $P_{4,25}$, we get an even better speedup, also shown in Figure 6.17 and 6.18. In this case, the CPU time decreases almost to $\frac{1}{10}$ compared to the sequential case (S). This speedup is due to the reasons explained in "Superlinear speedup" section. When we increase the number of threads to 4, the speedup is again proportional to the number of threads and this is what we expected. Here, by doubling the number of threads, the speedup doubles.

In order to better visualize the effect of increasing number of threads, we also examined the galen-1 ontology using a different number of threads (e.g. 1, 2, 4, 8, 16) and partition size 5 (see Figure 6.19 and Figure 6.20). As shown in Fig. 6.19, the CPU time decreases when increasing number of threads from 1 - 4; however, increasing the thread numbers more than 4 causes the CPU time to increase. Using 16 threads shows that the CPU time is higher than in the sequential case. The

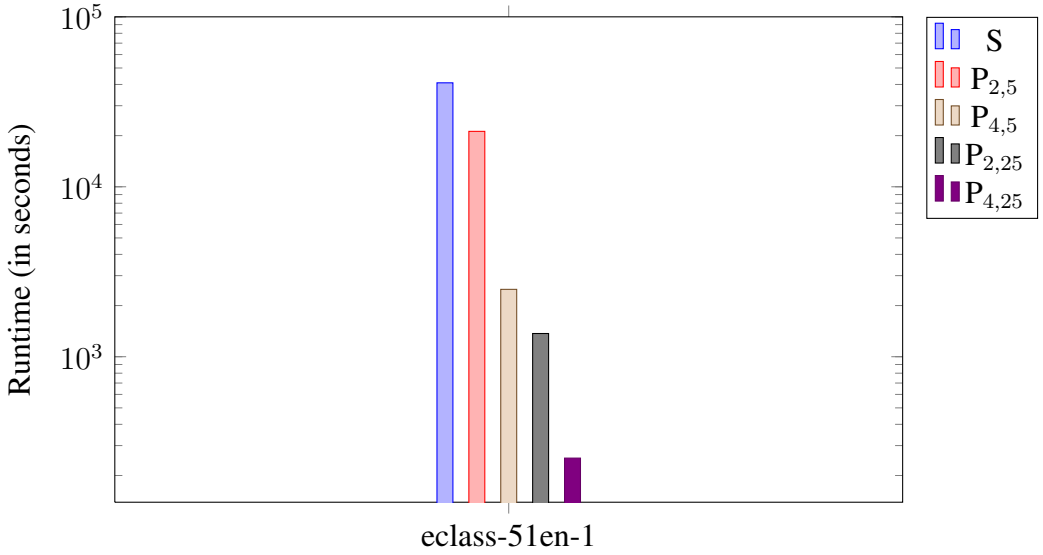


Figure 6.17: Runtimes for eclass-51en-1 using 5 settings: S (sequential), $P_{2,5}$, $P_{4,5}$, $P_{2,25}$, $P_{4,25}$ ($P_{threads,partition\_size}$).

performance degradation is because each thread does I/O and also, 16 number of threads is over the optimal number of threads that a system with 8 cores can manage.

The same explanation is for the speedup, shown in Figure 6.20. The speedup increases using 1 - 4 threads and it slows down after 4 threads. This behaviour is expected as increasing the number of threads means more re-runs or in other words, increase in number of corrections.

6.4.2 Effect of changing only partition sizes

The performance of the classifier in this case for EClass-51En-1 is also shown in Figure 6.17 with a fixed number of threads (2 or 4) but different partition sizes (5 or 25). When using 2 threads, compared to case S, we get the ideal speedup for $P_{2,5}$, as shown in Figure 6.18. As we can see, doubling the number of threads,

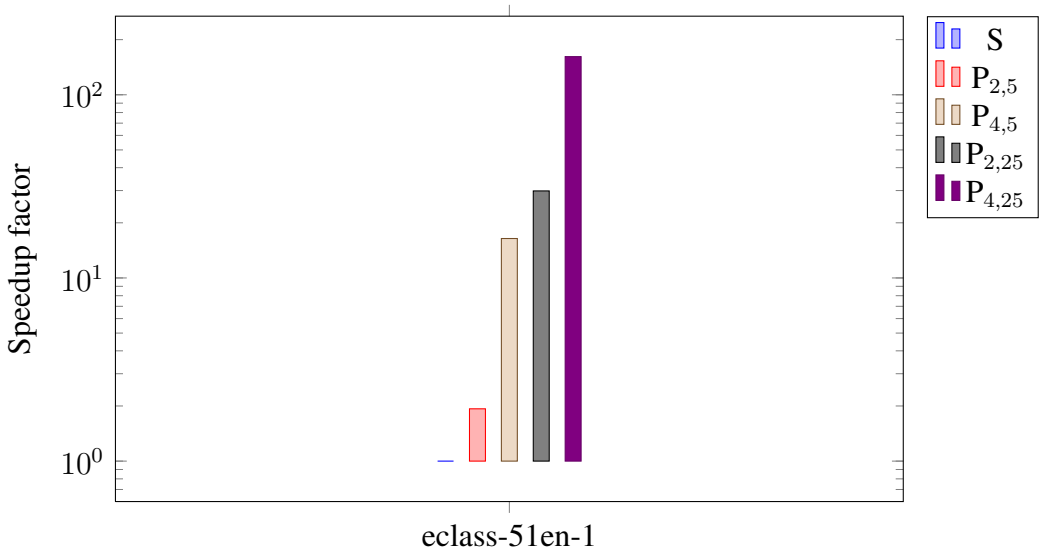


Figure 6.18: Speedup for eclass-51en-1 from Figure 6.17.

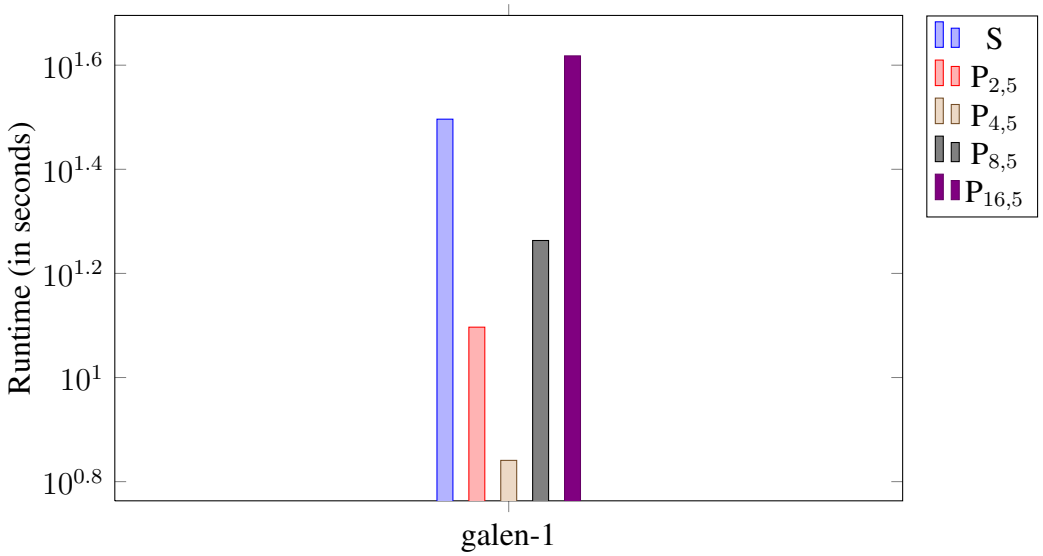


Figure 6.19: Runtimes for galen-1 using 5 settings: S (sequential), $P_{2,5}$, $P_{4,5}$, $P_{8,5}$, $P_{16,5}$ ($P_{threads,partition\_size}$).

doubles the speedup, in other words, decreases the CPU time by half. This is the ideal case which is what we were expecting to happen. Again, compared to case S if the partition size is increased to 25, it shows the same speedup as shown in

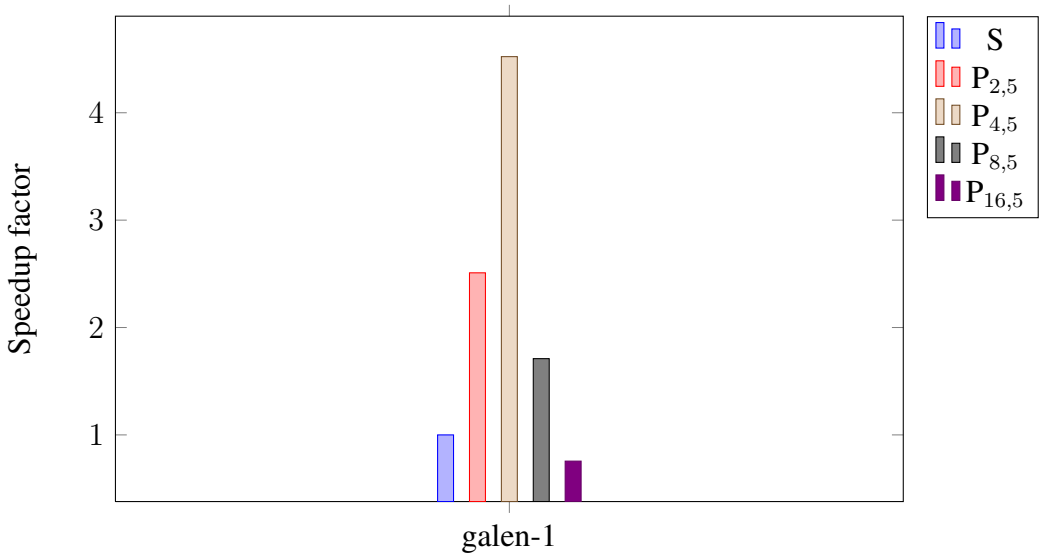


Figure 6.20: Speedup for galen-1 from Figure 6.19.

Figure 6.18. In this case, the CPU time decreases almost to $\frac{1}{5}$ compared to the previous case.

In the scenario with 4 threads, we get a corresponding speedup, as shown in Figure 6.18. In this case, the CPU time decreases to almost $\frac{1}{10}$ compared to the sequential case. This speedup is again due to the reasons mentioned in the ”Superlinear speedup” section. When we increase the partition size to 25, the speedup is what we expected. Here, by multiplying the partition size by 5, the speedup is multiplied by five too.

Increasing the partition size, means that more concepts are assigned to one thread; therefore, all the related concepts are inserted into the taxonomy by one thread. Hence, increasing the partition size, reduces the number of corrections. Comparing $P_{2,5}$ and $P_{2,25}$ shows that increasing the partition size from 5 to 25, makes the number of re-runs for $P_{2,25}$ half of $P_{2,5}$.

6.4.3 Effect of increasing both the number of threads and the partition size

In this scenario, we measured the CPU time when increasing both the number of threads and the partition size. In Figure 6.21 and 6.22, our test suite includes the ontologies Embassi-2, Galen1, LargeTestOntology, Tambis-2a, Cyc, and EClass-51En-1. The CPU time for each test case is shown in Figure 6.21 and the speedup factor for each experiment is depicted in Figure 6.22. As the results show, in the scenario with 2 threads and partition size 5, the speedup doubles compared to the sequential case and is around 2 and this is what we were expecting. When we increase the number of threads as well as the partition size, for the scenario with 4 threads and partition size 25, the CPU time decreases dramatically and therefore the speedup factor is above 20 for most test cases. This is more than a linear speedup, and it is the result of increasing the thread number as well as partition size together with the reasons previously explained in "Superlinear speedup" section.

Increasing the number of threads to 6 and the partition size to 65, speedup does not show a dramatic change for small ontologies comparing to scenario $P_{4,25}$, however, for Cyc and EClass-51en-1 the speed up is superlinear. Considering scenario with 8 threads and partition size 125, cpu time decrease more than 1/30 compared to the sequential case. The good scalability is due to combination of decreasing number of total subsumption tests as well as re-runs per threads.

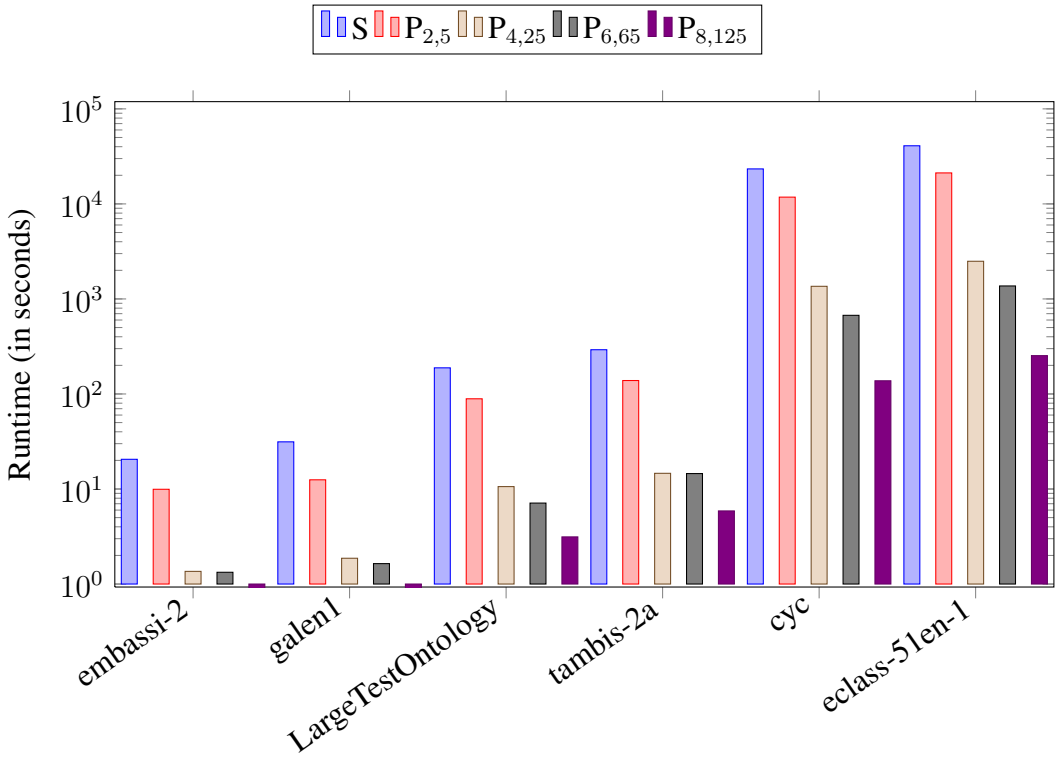


Figure 6.21: Runtimes for ontologies using 5 settings: S (sequential), $P_{2,5}$, $P_{4,25}$, , $P_{6,65}$, $P_{8,125}$.

6.4.4 Experiment on very large ontologies

We selected 3 Snomed variants as very large ontologies with more than 150,000 concepts. Snomed-2 with 182,869 concepts, Snomed-1 with 223,260 concepts, and Snomed with 379,691 concepts were included in our tests. Figure 6.25 shows an excellent improvement of CPU time for the parallel over the sequential case. In Figure 6.26, the speedup factor is almost 2, which is the expected behavior. The best speedup factor is observed for test case Snomed. Increasing the number of threads as well as partition size for Snomed family (e.g. $P_{4,25}$, $P_{8,125}$), shows a great performance. In scenario $P_{4,25}$, the speedup factor is close to 17 and for $P_{8,125}$ the speedup factor is around 35. Therefore, increasing the number of threads

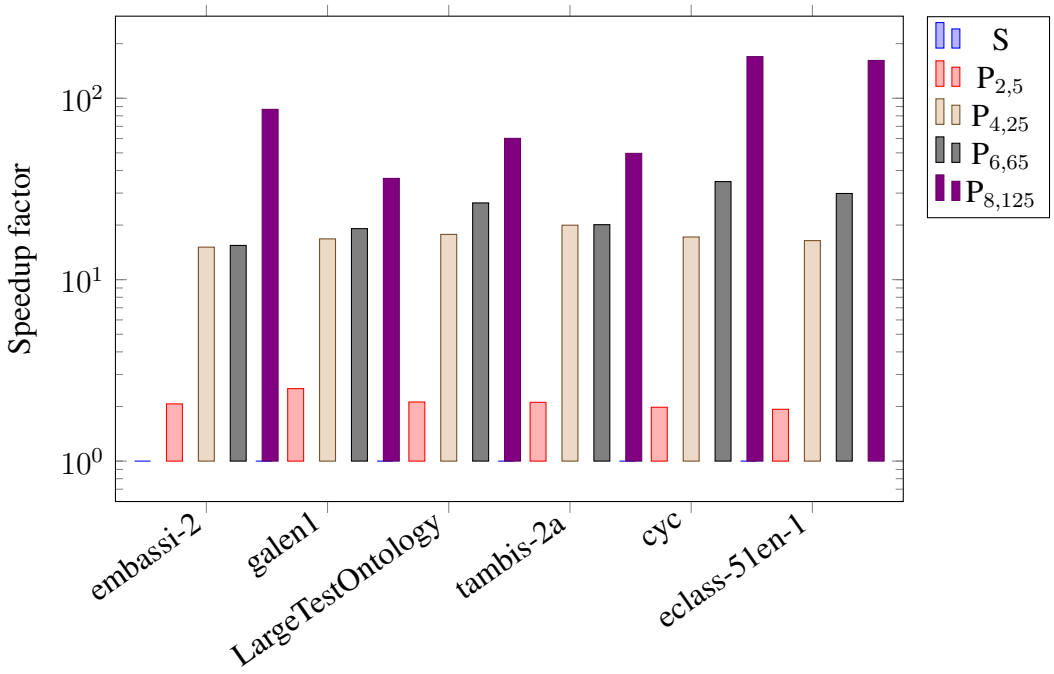


Figure 6.22: Speedup for ontologies from Figure 6.21.

as well as partition size results to a superlinear speedup.

6.4.5 Observation on the increase of size of ontologies

We chose Cyc, EClass-51en-1, Snomed-1, Snomed-2, and Snomed as test cases. Here, as shown in Figure 6.23 and 6.25, in a parallel setting with 2 threads, the CPU time is divided by 2 compared to the sequential case. The speedup, shown in Figure 6.24 is linear for $P_{2,5}$ and superlinear for $P_{4,25}$, $P_{6,65}$ and $P_{8,125}$.

In Figure 6.26, speedup is linear for $P_{2,5}$ and superlinear for $P_{4,25}$ as well as $P_{8,125}$. The result is consistent for our benchmark ontologies even when the size of the ontologies increases.

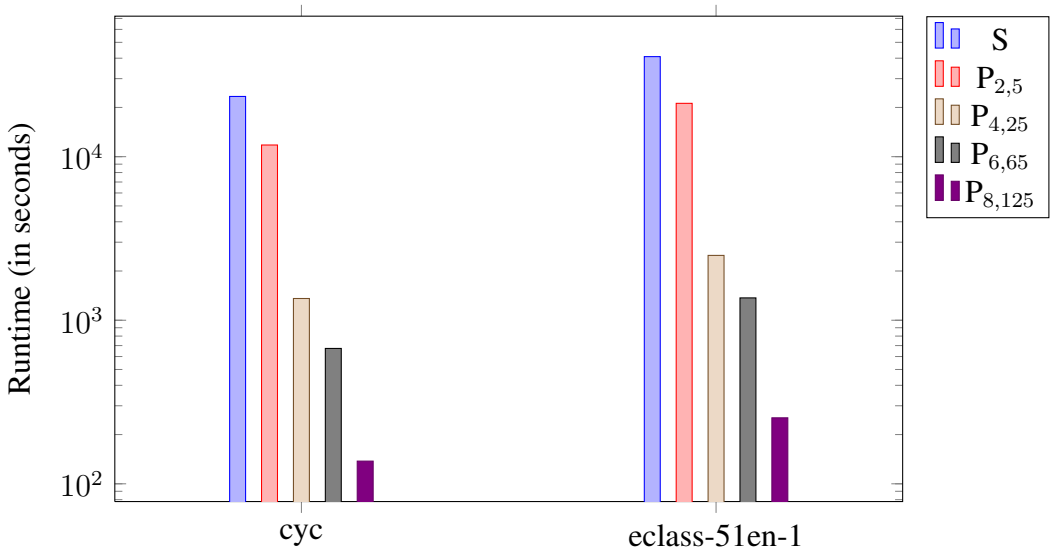


Figure 6.23: Runtimes for cyc and eclass-51en-1 using 4 settings: S (sequential), $P_{2,5}$, $P_{4,25}$, $P_{6,65}$, $P_{8,125}$.

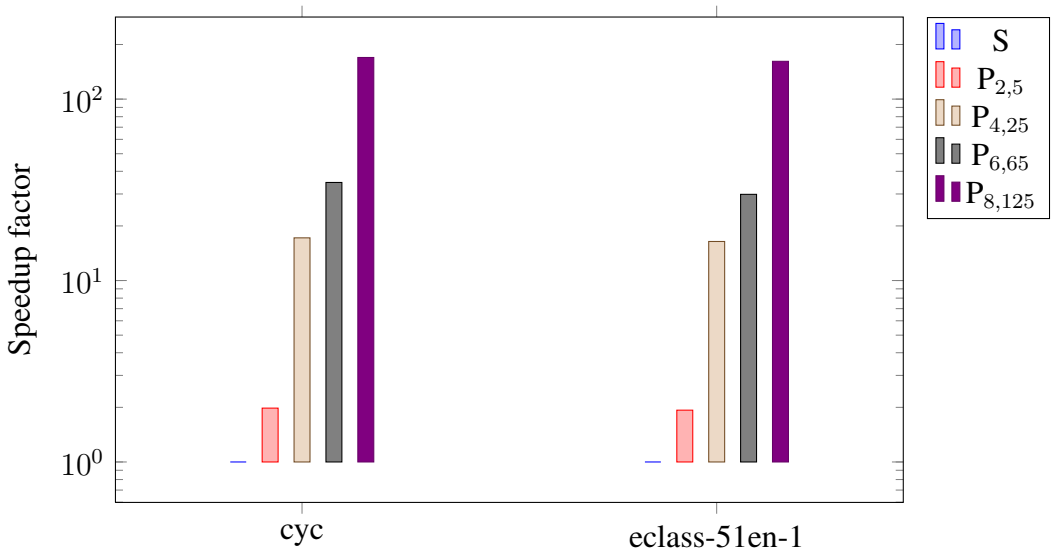


Figure 6.24: Speedup for ontologies from Figure 6.23.

6.4.6 Effect of changing partitioning scheme

To optimize the partitioning, as explained in Section 5.2.5, a new partitioning algorithm was designed. This Informed Partitioning algorithm, will open the win-

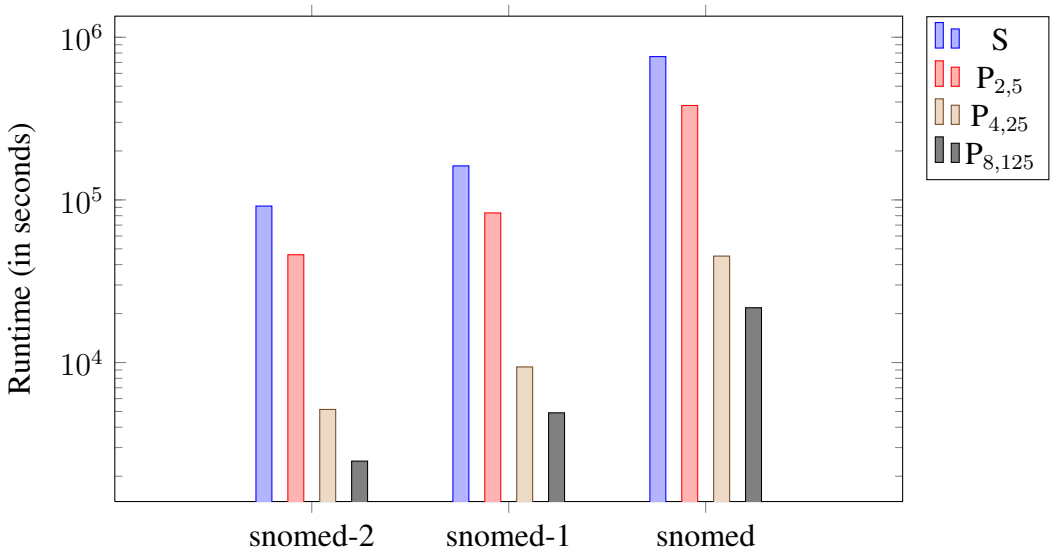


Figure 6.25: Runtimes for snomed using 3 settings: S (sequential), $P_{2,5}$, $P_{4,25}$, $P_{8,125}$.

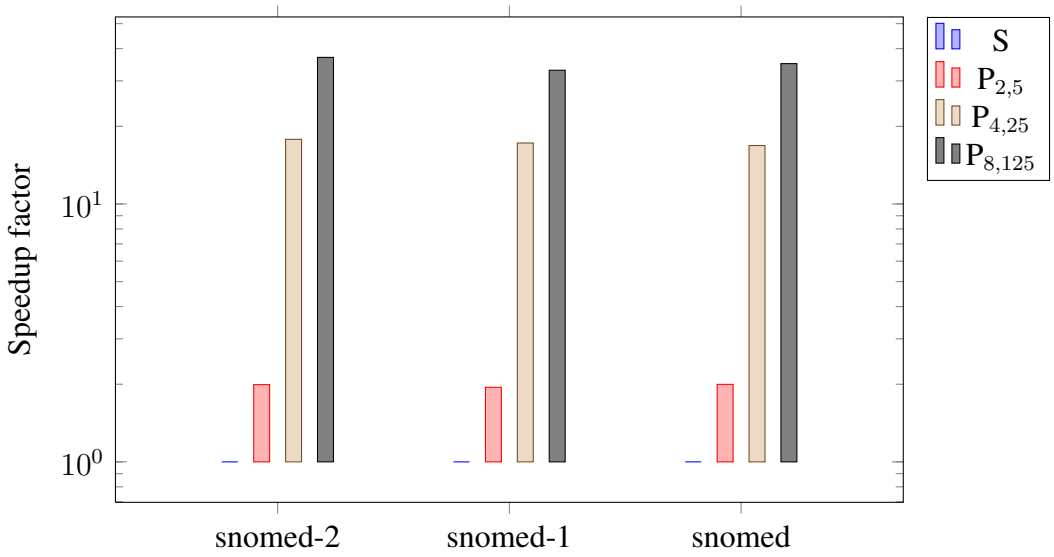


Figure 6.26: Speedup for ontologies from Figure 6.25.

dow for the future works as well. The evaluation of the algorithm is done with ontologies shown in Table 6.5, which are only proof of concept. Their name, size in number of named concepts, and used DL are also shown in Table 6.5. To mea-

sure the performance of the prototype using different partitioning, three scenarios are investigated:

Scenario I In this scenario, Partition Manager utilizes Fixed-Size Partitioning as explained in Section 5.2.5. Therefore, Partition Manager partitions the topological sort order list generated by Preprocessor into Fixed-Size partitions.

Scenario II In this scenario, Partition Manager uses Atomic Decomposition Partitioning, described in Section 5.2.5. Here, no preprocessing is done, and in other words topological sort order list is not generated. Hence, Partition Manager generates partitions based on the order of partitions using Atomic Decomposition partitioning, explained in Section 5.2.5. In this scenario, the partition size is not fixed.

Scenario III In this scenario, Partition Manager exploits Informed Partitioning, described in Section 5.2.5. Here, preprocessing is done, and a topological sort order list is generated. Hence, Partition Manager generates partitions based on the order of topological sort order list. In this scenario, the partition size is not fixed.

As it is shown in Figure 6.27, classification time for Scenario III, using Informed Partitioning is better than Scenario I, which is Fixed-Size Partitioning. This means that generating the topological order through preprocessing and applying Atomic Decomposition on top of it, gives a better classification time. Scenario II, which is Atomic Decomposition, is the worst case scenario for all the test cases. The comparison between Scenario I and Scenario II, emphasizes that using topological sort order gives a better performance for the tested ontologies. In Figure 6.28, total subsumptions for Scenario III shows better partitioning as

Table 6.5: Characteristics of the used test ontologies.

Ontology	DL language	No. of named concepts
Espr-gcis	$\mathcal{FL}-\mathcal{HN}$	143
Stereo-nums	$\mathcal{ALEHN}$	395
Embassi-2	$\mathcal{ALCHN}$	657

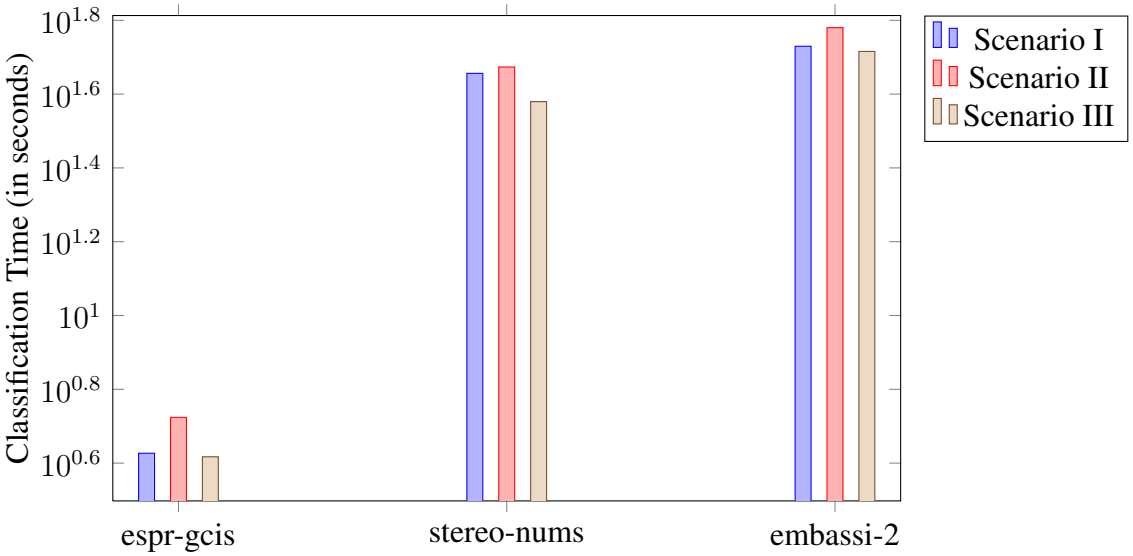


Figure 6.27: Classification Time for ontologies using 3 settings: Scenario I, Scenario II, Scenario III.

opposed to Scenario II, and it is similar to the classification time. And as it is presented in Figure 6.29, the same theme is existed for the Rerun ratio, which is $\frac{WastedRerun}{Rerun}$. The ratio measures the percentage of the total re-runs which are wasted. As it is shown in the evaluation of the different scenarios of partitioning, Informed Partitioning shows that it is more efficient comparing to the other partitioning scenarios.

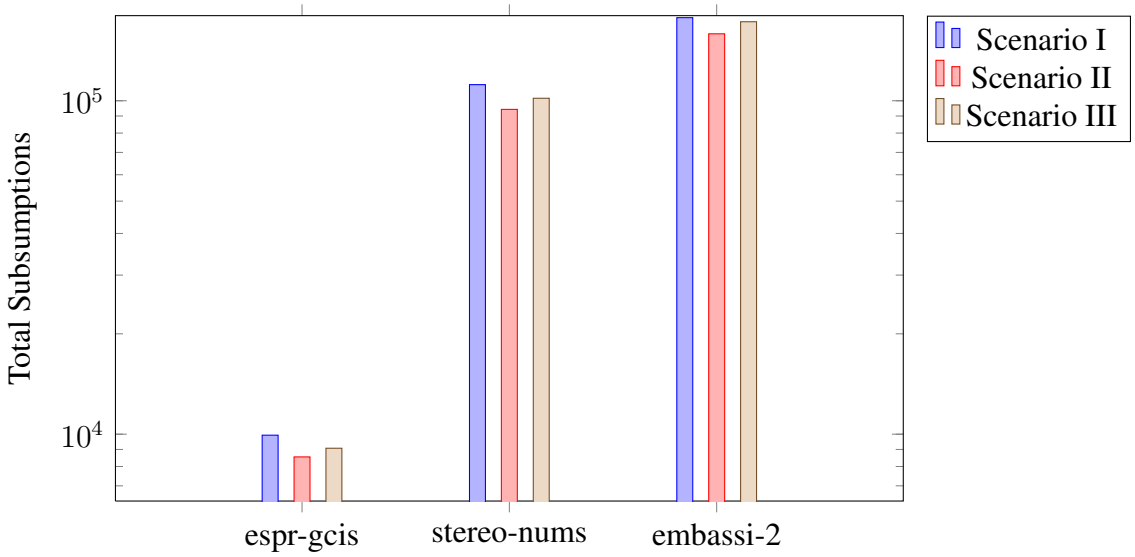


Figure 6.28: Total Subsumptions for ontologies using 3 settings: Scenario I, Scenario II, Scenario III.

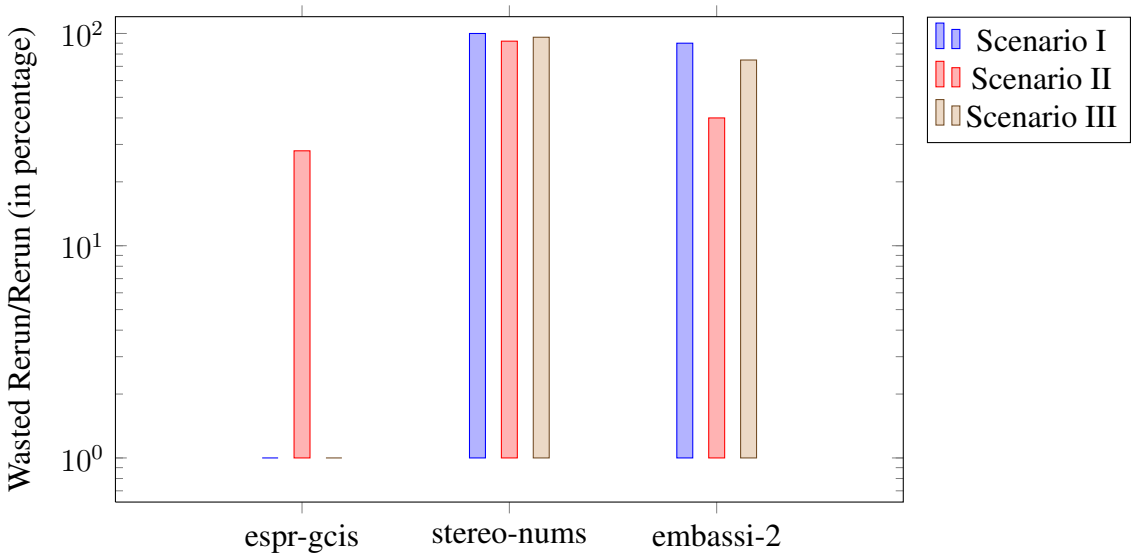


Figure 6.29: Wasted Rerun/Rerun for ontologies using 3 settings: Scenario I, Scenario II, Scenario III.

Chapter 7

Conclusion and Future Work

The main objective of this thesis as outlined in Section 1.1 was to design sound and complete algorithms to handle TBox classification in parallel, as no existing general system classifies TBoxes in parallel. Hence, the research methodology utilized consists of devising new sound and complete classification algorithms, designing a practical implementation of such algorithms, as well as conducting an evaluation of the corresponding prototypes with (large) real world ontologies. Therefore, the need for a new classification algorithm led to the design, implementation and evaluation of three generations of Parallel Classifier.

7.1 Theoretical Contributions

- A set of algorithms to construct a TBox in parallel, which is independent of a particular DL logic, although it sacrifices completeness was proposed and published in [1]. This algorithm was the first one focusing on TBox classification in parallel. The algorithm was fully explained in Section 5.1.1.

- A sound and complete algorithm for TBox classification in parallel was proposed and published in [2] and [3]. The algorithm was the first sound and complete algorithms for TBox classification in parallel, and described thoroughly in section 5.1.2 along with the proofs for completeness.

- An optimized extension of the sound and complete algorithm which is used to handle TBox classification concurrently, was proposed and published in [4]. This algorithm was detailed in section 5.1.3.

- Optimization technique suitable for partitioning the list of concepts to be inserted into the TBox was designed and described in Section 5.2.5. The goal of the proposed Informed Partitioning is to achieve performance improvement for classification time.

7.2 Practical Contributions

A running prototype, Parallel TBox Classifier, was developed for every generation of the classifier and evaluated against existing SOTA benchmarks, as described in Sections 6.2, 6.3 and 6.4, respectively. To the best of our knowledge, Parallel TBox Classifier is the first general TBox classifier which runs concurrently with linear scalability to the utilized number of processors/cores.

Parallel TBox Classifier consists of seven components. The Configuration Manager, checks, verifies and utilizes the user preferences specified in the config file. The Ontology Loader, gets the input ontologies file, selected from the computer directory. The Parser parses the loaded ontologies and fills the corresponding data structures with the per-concept information available in the file such as

its name, parents (in the complete taxonomy), told disjoints, told subsumers, and pseudo model information. The Preprocessor, sorts the list of the concepts to be inserted in the TBox using Topological Sort Order. The Partitioning Manager partitions the concept list using one of the four partitioning methods, e.g. Fixed-size Partitioning, Dynamic-size Partitioning, Atomic Decomposition Partitioning and Informed Partitioning. The Serializer, captures the statistics while classification is running. Finally and most importantly, TBox classifier, constructs the TBox in parallel.

The empirical evaluation in Chapter 6 illustrated that the prototypes for Parallel TBox Classifier in the first and second generations were restricted to classify TBoxes of up to 10000 node. However, in the third generation, the prototype can classify TBoxes with more than 370000 nodes. Therefore, the scalability is more than one order of magnitude. The assessments also show that Parallel TBox Classifier outperforms the Sequential TBox Classifier in all the test cases (all of them are real world ontologies), particularly with very large ontologies (e.g. Snomed with over 370,000 named concepts) with a linear or superlinear speedup factor of over 30. Hence, the usability and scalability of the approach for the SOTA ontologies, are met. Also, utilizing the Informed Partitioning, which is a partitioning optimization technique, depicted that it results up to 20% of performance improvement for classification time.

7.3 Future Research

Our proposed approach can be extended for improvement in the following highlighted aspects. Some of these open works were not explored due to time con-

straints, and some were noticed while conducting the experimental evaluation.

- Utilizing per-concept's told disjoint and told-subsumer information available in the input file provided by Racer. Once enabled as an optimization for reducing the number of subsumption tests, one should see improvement in qualitative and quantitative metrics.

- Using heuristics to reduce wasted re-running of Top Search and Bottom Search while the concepts are inserted in TBox concurrently. The optimization technique should not sacrifice the completeness of the algorithm explained in Section 6.4.

- Exploiting well-known optimizations to reduce the number of subsumption tests. These optimizations can avoid subsumption tests or eliminate the bottom search for some DL languages or decrease the number of bottom searches in general.

- Extending Informed Partitioning to run with larger ontologies. The algorithm was explained in Section 5.2.5.

Bibliography

[1] M. Aslani and V. Haarslev. Towards parallel classifcation of TBoxes. In *Proceedings of the 2008 International Workshop on Description Logics (DL-2008), Dresden, Germany, May 13-16*, 2008.

[2] M. Aslani and V. Haarslev. Parallel tbox classification in description logics - first experimental results. In *Proceedings of the 19th European Conference on Artificial Intelligence - ECAI 2010, Lisbon, Portugal, Aug. 16-20, 2010, pp. 485-490*, pages 485–490, 2010.

[3] M. Aslani and V. Haarslev. Tbox classification in parallel: Design and first evaluation. In *Proceedings of the 2010 International Workshop on Description Logics (DL-2010), Waterloo, Canada, May 4-7*, pages 336–347, 2010.

[4] M. Aslani and V. Haarslev. Concurrent classification of owl ontologies - an empirical evaluation. In *Proceedings of the 2012 International Workshop on Description Logics (DL-2012), Rome, Italy, June 7-10*, pages 400–410, 2012.

[5] F. Baader and B. Hollunder. Kris : Knowledge representation and infererence system. In *Deutaches Forschungszentrum fur kunstliche Intelligenz (DFKI)*, 1991.

[6] F. Baader, B. Hollunder, B. Nebel, and H.-J. Profitlich. An empirical analysis of optimization techniques for terminological representation systems. In *German Research Center for AI*.

[7] F. Baader and W. Nutt. Basic description logics. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 43–95. Cambridge University Press, 2003.

[8] F. Baader and U. Sattler. Tableau algorithms for description logics. In *Proceedings of the International Conference on Automated Reasoning with Tableaux and Related Methods*, 2000.

[9] F. Bergmann and J. Quantz. Parallelizing description logics. In *Proc. of 19th Ann. German Conf. on Artificial Intelligence*, LNCS, pages 137–148. Springer-Verlag, 1995.

[10] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick. Classic: A structural data model for objects. In *In Proc. of the ACM SIGMOD Int. Conf. on Management of Data.*, pages 59–67, 1989.

[11] A. Borgida and P.F. Patel-Schneider. A semantics and complete algorithm for subsumption in the classic description logic. In *J. of Artificial Intelligence Research.*, page 277308, 1994.

[12] R. J. Brachman, D. L. McGuiness, P. F. Patel-Schneider, and Lori A. Resnick. Living with classic: when and how to use a kl-one-like language. In *In John Sowa, editor, Principles of semantic networks: Explorations in the representation of knowledge.*, pages 401–546, 1991.

[13] Ch. Cumbo, W. Faber, G. Greco, and N. Leone. Enhancing the magic-set method for disjunctive datalog programs. In *In Logic Programming, Lecture Notes in Computer Science*, pages 371–385. Springer, 2004.

[14] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Google, Inc*, 2008.

[15] F. M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. http://www.diegm.uniud.it/satt/papers/DLNS96b.pdf[Accessed July, 2012].

[16] B. Glimm, I. Horrocks, and B. Motik. Optimized description logic reasoning via core blocking. In *Proc. of the 5th Int. Joint Conf. on Automated Reasoning (IJCAR 2010)*, pages 457–471, 2010.

[17] B. Glimm, I. Horrocks, B. Motik, and G. Stoilos. Optimising ontology classification. In *Proc. of the 9th Int. Semantic Web Conf. (ISWC 2010)*, pages 225–240, 2010.

[18] Racer Systems GmbH and Co. KG. User guide for racerpro 2.0. Latest version available at http://www.racer-systems.com/products/racerpro/manual.phtml.

[19] W3C OWL Working Group. Owl 2 web ontology language,document overview. Latest version available at http://www.w3.org/TR/owl2-semantics/.

[20] V. Haarslev and R. Möller. Racer system description. In *Lecture Notes in Computer Science.*, pages 701–705, 2001.

[21] V. Haarslev, R. Möller, K. Hidde, and M. Wessel. The racerpro knowledge representation and reasoning system. In *Semantic Web, Volume 3, No. 3* ., pages 267–277, 2012.

[22] V. Haarslev, R. Möller, and A.-Y. Turhan. Exploiting pseudo models for TBox and ABox reasoning in expressive description logics. In *Proc. of the Int. Joint Conf. on Automated Reasoning, IJCAR'2001, June 18-23, 2001, Siena, Italy*, LNCS, pages 61–75, June 2001.

[23] V. Haarslev, R. Möller, and A.-Y. Turhan. Exploiting pseudo models for TBox and ABox reasoning in expressive description logics. In *Proc. of the Int. Joint Conf. on Automated Reasoning, IJCAR'2001, June 18-23, Siena, Italy*, pages 61–75, 2001.

[24] A. Hogan, J.Z. Pan, A. Polleres, and S. Decker. SAOR: template rule optimisations for distributed reasoning over 1 billion linked data triples. In *Proc. 9th Int. Semantic Web Conf.*, pages 337–353, 2010.

[25] I. Horrocks. Using an expressive description logic: Fact or fiction?, medical informatics group, department of computer science, university of manchester.

[26] I. Horrocks. Implementation and optimization techniques. In *Chapter 9, The Description Logics Handbook, Theory, Implementation, and Applications*, pages 306–346. Cambridge University Press, 2002.

[27] I. Horrocks. Description logic reasoning. In *Invited tutorial at LPAR, Montevideo, Uruguay*, March 2005.

[28] Y. Kazakov, M. Krötzsch, and F. Simancik. Concurrent classification of EL ontologies. In *Proc. of the 10th Int. Semantic Web Conf.*, pages 305–320, 2011.

[29] S. Kotoulas, E. Oren, and F. van Harmelen. Mind the data skew: distributed inferencing by speeddating in elastic regions. In *Proc. 19th Int. Conf. on World Wide Web*, pages 531–540, 2010.

[30] T. Liebig and F. Müller. Parallelizing tableaux-based description logic reasoning. In *Proc. of 3rd Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS '07), Vilamoura, Portugal, Nov 27*, volume 4806 of *LNCS*, pages 1135–1144. Springer-Verlag, 2007.

[31] R. MacGregor. The evolving technology of classification-based knowledge representation systems. In *In Principles of Semantic Networks: Explorations in the Representation of Knowledge, Morgan Kaufmann.*, pages 385–400, 1991.

[32] H. Majid. Knowledge representation: Historical perspective, state of the art and future prospects. In *Department of Computer Science and Information Systems, University of Limerick, Limerick, Ireland*.

[33] N. Monty. Automated theorem proving: Therory and practice. In *McGill University, Computer Science Department*.

[34] B. Motik, R. Shearer, and I. Horrocks. Hypertableau reasoning for description logics. In *Journal of Artificial Intelligence Research*, pages 457–471, 2009.

[35] R. Mutharaju, F. Maier, and P. Hitzler. A MapReduce algorithm for EL+. In *Proc. 23rd Int. Workshop on Description Logics*, pages 464–474, 2010.

[36] D. Nardi and R. J. Brachman. An introduction to description logics. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 1–40. Cambridge University Press, 2003.

[37] M. Newborn and Z. Wang. Octopus: Combining learning and parallel search. In *Journal of Automated Reasoning, Volume 33, Number 2*.

[38] G. A. Ringland and D. A. Duce. Approaches to knowledge representation : an introduction. In *Research Studies Press Ltd*, 1988.

[39] A. Schlicht and H. Stuckenschmidt. Distributed resolution for expressive ontology networks. In *Web Reasoning and Rule Systems, 3rd Int. Conf. (RR 2009), Chantilly, VA, USA, Oct. 25-26, 2009*, pages 87–101, 2009.

[40] A. Schlicht and H. Stuckenschmidt. Distributed resolution for expressive ontology networks. In *Proc. 3rd Int. Conf. on Web Reasoning and Rule Systems*, pages 87–101, 2009.

[41] A. Shaban-Nejad. A framework for analyzing changes in health care lexicons and nomenclatures. A PhD Thesis In The Department of Computer Science and Software Engineering, Concordia University.

[42] H. Shan and J. P. Singh. Parallel tree building on a range of shared address space multiprocessors: Algorithms and application performance. In *12th*

Int. Parallel Processing Symposium (IPPS '98), March 30 - April 3, 1998, Orlando, Florida, USA, pages 475–484, 1998.

[43] E. Sirin, B. Cuenca Grau, and B. Parsia. Pellet: a practical owl-dl reasoner. In *Proceedings of the International Conference on the Principles of Knowledge Representation and Reasoning (KR).*, pages 90–99, 2006.

[44] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz. Pellet: a practical owl-dl reasoner. In *Journal of Web Semantics*, pages 51–53, 2007.

[45] D. Tsarkov. Improved algorithms for module extraction and atomic decomposition. In *Description Logics*, 2012.

[46] D. Tsarkov and I. Horrocks. Fact++ description logic reasoner:system description. In *In Proc. of the Int. Joint Conf. on Automated Reasoning(IJCAR 2006), volume 4130 of Lecture Notes in Artificial Intelligence.*, pages 292–297. Springer, 2006.

[47] D. Tsarkov and I. Palmisano. Chainsaw: a metareasoner for large ontologies. In *OWL Reasoner Evaluation Workshop (ORE 2012) collocated with IJCAR 2012 Conference, Manchester, UK.*, pages 19–27, 2012.

[48] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. Bal. WebPIE: a webscale parallel inference engine using mapreduce. In *J. of Web Semantics*, 2011.

[49] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen. Scalable distributed reasoning using MapReduce. In *International Semantic Web Conference*, pages 634–649, 2009.

[50] Ch. Del Vescovo, B. Parsia, U. Sattler, and Th. Schneider. The modular structure of an ontology: Atomic decomposition and module count. University of Manchester, UK and Universität Bremen, Germany.

[51] J. Weaver and J.A. Hendler. Parallel materialization of the finite RDFS closure for hundreds of millions of triples. In *Proc. 8th Int. Semantic Web Conf.*, pages 87–101, 2009.

Appendix A

Complete Pseudo code for Parallel TBox Classification

In this chapter, in order to have the convenience of reading all the pseudo codes, the corresponding pseudo codes for each of the prototypes explained in chapter 5 and evaluated in chapter 6, will be provided.

A.1 First Generation

Algorithm 21 parallel_tbox_classification(*concept_list,shuffle_flag*)

topological_order_list ← topological_order(*concept_list*)
if *shuffle_flag* **then**
  topological_order_list ← random_shuffle(*topological_order_list*)
repeat
  assign each idle thread t_i a partition p_i from *topological_order_list*
  run idle thread t_i with insert_partition(p_i)
until all concepts in *topological_order_list* are inserted
compute missing subsumptions and ratio
print statistics

Algorithm 22 insert_ partition(*partition*)

```
for all new ∈ partition do
  parents ← top_search(new,⊤)
  lock(new)
  set predecessors of new to parents
  for all pred predecessor of new do
    lock(pred)
    add new to successors of pred
    unlock(pred)
  unlock(new)
  children ← bottom_search(new,⊥)
  lock(new)
  set successors of new to children
  for all succ successor of new do
    lock(succ)
    add new to predecessors of succ
    unlock(succ)
  unlock(new)
```

Algorithm 23 top_search(*new*,*current*)

```
mark(current,'visited')
pos-succ ← ∅
for all y ∈ successors(current) do
  if enhanced_top_subs(y,new) then
    pos-succ ← pos-succ ∪ {y}
if pos-succ = ∅ then
  return {current}
else
  result ← ∅
  for all y ∈ pos-succ do
    if y not marked as 'visited' then
      result ← result ∪ top_search(new,y)
  return result
```

Algorithm 24 enhanced_top_subs(*current*,*new*)

if *current* marked as 'positive' **then**
  return *true*
else if *current* marked as 'negative' **then**
  return *false*
else if for all $z \in$ predecessors(*current*)
      always enhanced_top_subs(*z*,*new*)
    and subsumes(*current*,*new*) **then**
  mark(*current*,'positive')
  return *true*
else
  mark(*current*,'negative')
  return *false*

Algorithm 25 subsumes(*current*,*new*)

Checks whether subsumer subsumes subsumee using computed told subsumer information provided by Racer.

A.2 Second Generation

Algorithm 26 parallel_tbox_classification(*concept_list*)

topological-order-list ← topological_order(*concept_list*)
repeat
  wait until an idle thread t_i becomes available
  select a partition p_i from *topological-order-list*
  run thread t_i with insert_partition(p_i, t_i)
until all concepts in *topological-order-list* are inserted
compute ratio and overhead
print statistics

Algorithm 27 insert_partition(*partition,id*)

lock($inserted\_concepts(id)$)
$inserted\_concepts(id) \leftarrow \emptyset$
unlock($inserted\_concepts(id)$)
for all *new* $\in$ *partition* **do**
  parents $\leftarrow$ top_search(*new*,$\top$)
  while $\neg$ consistent_in_top_search(*parents*,*new*) **do**
    parents $\leftarrow$ top_search(*new*,$\top$)
  lock(*new*)
  predecessors(*new*) $\leftarrow$ *parents*
  unlock(*new*)
  for all *pred* $\in$ *parents* **do**
    lock(*pred*)
    successors(*pred*) $\leftarrow$ successors(*pred*) $\cup$ {*new*}
    unlock(*pred*)
  children $\leftarrow$ bottom_search(*new*,$\bot$)
  while $\neg$ consistent_in_bottom_search(*children*,*new*) **do**
    children $\leftarrow$ bottom_search(*new*,$\bot$)
  lock(*new*)
  successors(*new*) $\leftarrow$ *children*
  unlock(*new*)
  for all *succ* $\in$ *children* **do**
    lock(*succ*)
    predecessors(*succ*) $\leftarrow$ predecessors(*succ*) $\cup$ {*new*}
    unlock(*succ*)
  $check \leftarrow check\_if\_concept\_inserted(new, inserted\_concepts(id))$
  if check $\neq$ 0 **then**
    if check = 1 $\vee$ check = 3 **then**
      new_predecessors $\leftarrow$ top_search(*new*,$\top$)
      lock(*new*)
      predecessors(*new*) $\leftarrow$ *new_predecessors*
      unlock(*new*)
    if check = 2 $\vee$ check = 3 **then**
      new_successors $\leftarrow$ bottom_search(*new*,$\bot$)
      lock(*new*)
      successors(*new*) $\leftarrow$ *new_successors*
      unlock(*new*)
  for all busy threads $t_i \neq id$ **do**
    lock($inserted\_concepts(t_i)$)
    $inserted\_concepts(t_i) \leftarrow inserted\_concepts(t_i) \cup \{new\}$
    unlock($inserted\_concepts(t_i)$)

Algorithm 28 top_search(*new*,*current*)

```
mark(current,'visited')
pos-succ ← ∅
captured_successors(new)(current) ← successors(current)
for all y ∈ successors(current) do
  if enhanced_top_subs(y,new) then
    pos-succ ← pos-succ ∪ {y}
if pos-succ = ∅ then
  return {current}
else
  result ← ∅
  for all y ∈ pos-succ do
    if y not marked as 'visited' then
      result ← result ∪ top_search(new,y)
  return result
```

Algorithm 29 consistent_in_top_search(*parents*,*new*)

```
for all pred ∈ parents do
  if successors(pred) ≠ captured_successors(new)(pred) then
    diff ← successors(pred) \ captured_successors(new)(pred)
    for all child ∈ diff do
      if subsumption_possible(child,new) then
        return false
return true
```

Algorithm 30 check_if_concept_inserted(*new*,*inserted_concepts*)

```
The return value indicates whether and what type of re-run needs to be done:
0 : No re-run in needed
1 : Re-run TopSearch because a possible parent could have been overlooked
2 : Re-run BottomSearch because a possible child could have been overlooked
3 : Re-run TopSearch and BottomSearch because a possible parent and child
could have been overlooked
if inserted_concepts = ∅ then
   return 0
else
   for all concept ∈ inserted_concepts do
      if subsumption_possible(concept,new) then
         if subsumption_possible(new,concept) then
            return 3
         else
            return 1
      else if subsumption_possible(new,concept) then
         if subsumption_possible(concept,new) then
            return 3
         else
            return 2
   return 0
```

A.3 Third Generation

Algorithm 31 informed_partitioning(*topological_sort_list*)

```
atoms ← get-atomic-decomposition-atoms(owlfile)
partition-counter ← 0
ignore-list ← 0
for all list ∈ topological_sort_list do
  for all c ∈ list do
    if ignore-list not contain c then
      ad-partition ← get-ad-partition-no(c, atoms)
      set informed-partitions(partition-counter) to ad-partition
      add ad-partition to ignore-list
      increment partition-counter
```

Algorithm 32 insert_partition(*partition,id*)

```
for all new ∈ partition do
  rerun ← 0
  finish_rerun ← false
  parents ← top_search(new,⊤)
  while ¬ consistent_in_top_search(parents,new) do
    parents ← top_search(new,⊤)
  predecessors(new) ← parents
  children ← bottom_search(new,⊥)
  while ¬ consistent_in_bottom_search(children,new) do
    children ← bottom_search(new,⊥)
  successors(new) ← children
  for all busy threads t_i ≠ id do
    located_concepts(t_i) ← located_concepts(t_i) ∪ {new}
  check ← check_if_concept_has_interaction(new, located_concepts(id))
  while (check ≠ 0) and ¬finish_rerun do
    if rerun < 3 then
      if check = 1 then
        new_predecessors ← top_search(new,⊤)
        rerun ← rerun + 1
        predecessors(new) ← new_predecessors
      else
        if check = 2 then
          new_successors ← bottom_search(new,⊥)
          rerun ← rerun + 1
          successors(new) ← new_successors
        check ←
        check_if_concept_has_interaction(new, located_concepts(id))
      else
        finish_rerun ← true
        for all busy threads t_i ≠ id do
          located_concepts(t_i) ← located_concepts(t_i) \ {new}
    if ¬finish_rerun then
      insert_concept_in_tbox(new, predecessors(new), successors(new))
```

Algorithm 33 consistent_in_top_search(*parents*,*new*)

```
for all pred ∈ parents do
  if successors(pred) ≠ captured_successors(new)(pred) then
    diff ← successors(pred) \ captured_successors(new)(pred)
    for all child ∈ diff do
      if found_in_ancestors(child,new) then
        return false
return true
```

Algorithm 34 check_if_concept_has_interaction(*new,located_concepts*)

The return value indicates whether and what type of re-run needs to be done:
0 : No re-run in needed
1 : Re-run TopSearch because a possible parent could have been overlooked
2 : Re-run BottomSearch because a possible child could have been overlooked

```
if located_concepts = ∅ then
  return 0
else
  for all concept ∈ located_concepts do
    if interaction_possible(new,concept) then
      if found_in_ancestors(new,concept) then
        return 2
      else
        return 1
    else if interaction_possible(concept,new) then
      if found_in_ancestors(new,concept) then
        return 2
      else
        return 1
  return 0
```

Algorithm 35 insert_concept_in_tbox(*new,predecessors,successors*)

```
for all pred ∈ predecessors do
  successors(pred) ← successors(pred) ∪ {new}
for all succ ∈ successors do
  predecessors(succ) ← predecessors(succ) ∪ {new}
```

Algorithm 36 interaction_possible(*new*,*concept*)

Uses pseudo model merging information [23], pre-computed by Racer, to decide whether a subsumption is possible between *new* and *concept*.

Algorithm 37 found_in_ancestors(*new*,*concept*)

Checks if *concept* is an ancestor of *new*.