Scaling Up Description Logic Reasoning by Distributed Resolution

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von
Anne Schlicht
aus München

Mannheim, Januar 2012

Dekan: Professor Dr. Heinz Jürgen Müller, Universität Mannheim
Referent: Professor Dr. Heiner Stuckenschmidt, Universität Mannheim
Korreferent: Professor Dr. Christoph Weidenbach, Max-Planck-Institut für Informatik, Saarbrücken

Tag der mündlichen Prüfung: 29. März 2012

Abstract

Benefits from structured knowledge representation have motivated the creation of large description logic ontologies. For accessing implicit information and avoiding errors in ontologies, reasoning services are necessary. However, the available reasoning methods suffer from scalability problems as the size of ontologies keeps growing.

This thesis investigates a distributed reasoning method that improves scalability by splitting a reasoning process into a set of largely independent subprocesses. In contrast to most description logic reasoners, the proposed approach is based on resolution calculi. We prove that the method is sound and complete for first order logic and different description logic subsets. Evaluation of the implementation shows a heavy decrease of runtime compared to reasoning on a single machine. Hence, the increased computation power pays off the overhead caused by distribution. Dependencies between subprocesses can be kept low enough to allow efficient distribution.

Furthermore, we investigate and compare different algorithms for computing the distribution of axioms and provide an optimization of the distributed reasoning method that improves workload balance in a dynamic setting.

Contents

List of Tables

List of Figures

Acknowledgement

Many thanks to Heiner Stuckenschmidt for his advice, his support and his patience and impatience. My special thanks to Christoph Weidenbach for his valuable feedback and for the SPASS he contributed. Also, I would like to thank my friends and family for sympathy and cheering up.

Part I

Introduction

Chapter 1

Motivation

The idea of describing the world in a formal way was first advertised by ancient Greek philosophers. Their motivation for creating ontologies was to get a better understanding of basic principles that induce the perceivable laws of nature [28]. Formal axiomatization has been especially popular in mathematics, where the prospect of designing a machine that would automatically find a proof or counterexample for every given conjecture thrilled mathematicians [44]. With Gödels proofs on the limits of computability [23] some expectations where disappointed but anyway, automatic deduction has gained increased attention with the invention of computers.

While first provers where limited to rather simple theories, available computation resources have reached a speed now that allows building systems that are relatively close to the vision of artificial intelligent machines. Expert knowledge is being formalized for making it easier accessible. For example, in areas like biology and medicine, ontologies are used to provide people with the knowledge covered by racks of textbooks and publications. Enterprises start exploiting the benefits from maintaining structured description of their expert knowledge. Like collections of tables in text files were replaced by database systems, knowledge management is shifting from text documents and even undocumented knowledge to ontological representations that are easily accessible.

Accessible representation of knowledge is already a big step towards artificial intelligence. But, without automatic deduction, benefits are limited. With increasing size and complexity of the representation, automatic deduction is essential to check correctness and to access implicit knowledge. Recently, the development of deduction methods is further boosted and challenged by the growth of the world wide web. The enhancement of information presented for human readers with machine readable semantic markup, initiated the creation of large ontologies. Ontologies for common knowledge and domain specific knowledge are the basis for translating and relating different semantic markup vocabulary used on the web.

1.1 Scalability Problem

Available deduction methods are not designed for very large ontologies, they face serious scalability problems considering runtime of query answering procedures.
In general there are three approaches to solving a scalability problem. The first solution approach is to *reduce the complexity of the problem*, i.e. to solve a less complex approximation of the problem. This is the first consideration when a large ontology is created: Computational complexity of the applied language should be as low as possible. The second approach is to *reduce the complexity of the solution algorithm*, i.e. compute not the exact solution but give an approximation of the solution. Large ontology projects [33, 43] usually use reasoners specifically designed for these ontologies that do not consider all possible implications. If the complexity of problem and solution cannot be reduced any more and the problem still remains intractable, the remaining last option is to simply *use more computation power*.
Unfortunately, there are hard limits to the amount of instructions per second that can be computed for a single process imposed by physical laws. Less hard but more relevant are the restrictions imposed by economical laws. A single supercomputer is considerable more expensive than a set of smaller machines that would provide the same total amount of computation speed. For some large ontologies currently in use we cannot provide a single process with enough computation resources for checking consistency. Hence, we aim at distributing the computation to a set of processes.
The decisive factor for a successful distribution to multiple processes are dependencies that require halting subprocesses and the amount of inter-process communication necessary for solving the task. Amdahl's law [2] states each program has a sequential component that cannot be parallelized and limits the speedup gained from adding processors. Avoiding dependencies corresponds to reducing the sequential part of a program.

1.2 Goal

Within this thesis, we aim at splitting up a reasoning task into a set of widely independent subproblems, such that the subproblems can be solved in parallel and combination of the sub-solutions to a solution of the original task is easy. We address applications that strictly require expressive ontologies and sound and complete reasoning, i.e. lossy translation to a more tractable language is not an option and reasoning results have to be reliable. Considering the expressivity supported by our distributed reasoning approach, we focus on the standard languages used for knowledge description on the Web. The most important standards are the Resource Description Framework (RDF) and the different dialects of the Web Ontology Language (OWL) [4]. The

most popular OWL dialect OWL-DL is based on description logic [29], a decidable subset of first order logic. Deciding consistency of an OWL-DL ontology is NP-complete. For answering queries to these kind of ontologies in realistic scenarios, special optimizations are necessary. Reasoning tasks for OWL-DL ontologies include queries like "Is the ontology consistent?", "What are the instances of concept C?", "Is C a subconcept of D?", "What is the concept hierarchy of the ontology?".
All of these query are reduced to satisfiability checks. For example, subsumption queries ("Is C a subconcept of D?") are executed by introducing a new instance x and adding to the ontology that x is an instance of C and an instance of the complement of D. If the obtained ontology is unsatisfiable, the subsumption is implied by the original ontology. Since other reasoning tasks are reduced to a satisfiability check, we focus on this basic reasoning task in this thesis.
We consider a simplified setting, where the available machines all have the same characteristics in terms of memory and computation speed. Optimizing the distribution of a reasoning task for machines with different characteristics is left to future work.
There are different strategies for distributed reasoning. Apart from the performance of a distributed reasoning task, also the maintenance requirements of a large ontology affect our choice of distribution strategy. Large ontologies are usually structured in modules for simplifying development. Hence, using a distribution strategy that is based on distributing the axioms of the ontology has advantages for maintenance. The distributed representation of the ontology can be used for both reasoning and maintenance.
To sum up, the goal is to create a distributed reasoning method that

- checks satisfiability of OWL-DL ontologies,
- is sound and complete and terminates,
- allows distributing the task to a number of processes that can be executed in parallel,
- uses a distribution strategy that is based on distributing the axioms of the ontology,
- achieves runtimes of about t/n for a number n of applied reasoners where t is the runtime on a single processor.

A reasoning method with these properties would improve scalability and help developing large expressive ontologies without inconsistencies.

1.3 Research Questions

In this work, we present a method that largely complies with the requirements stated above. We investigate preconditions, properties and capabili-

ties of distributed reasoning for description logic ontologies. In particular, we answer the following research questions.

Q1 Which reasoning method is a good basis for distributed reasoning on description logic ontologies?

Many reasoning methods have been developed for different purposes. We do not develop a completely new method from scratch, but base our distributed method on existing work.

Q2 Is it possible to preserve soundness, completeness and termination of this reasoning method when distributing computation to a set of parallel processes?

Of course, we have to start with a sound complete and terminating reasoning method. But, it is not clear if these properties can be preserved by distribution. The main contribution of this work is the proof of a positive answer to this question.

Q3 Is the distribution efficient, i.e. is the runtime decreased by distributing a reasoning task?

Assume we can find a reasoning method and distribution strategy with the desired theoretical properties. The next question is, whether the benefits from using multiple processors pays off for the overhead generated by distribution.

Q4 Does the distributed reasoning method scale?

Our distributed reasoning method is developed for large ontologies. Hence an important question is the behavior of the method when the size of the input is increased. It would be perfect if we could decrease runtime arbitrarily by applying more reasoners. But, we expect that the number of reasoners that can successfully contribute to a reasoning task is limited and depends on the size and complexity of the ontology.

Q5 What is the expressivity that can be supported by distributed reasoning?

The goal is to give full support for reasoning on OWL-DL ontologies. However, there are important subsets of this language, that allow specialized reasoning methods and hence require adaption of the distributed method.

Q6 What is the best method for computing a distribution of input axioms?

The performance of distributed reasoning depends on the method we choose for distributing the axioms. We describe different methods and compare the performance on distributed reasoning to a random distribution. The investigation shows if additional overhead for computing a good distribution is paid off by reduced runtime of the reasoning task.

Q7 Is it possible to change the distribution of axioms at runtime?

In a dynamic setting, the number of available compute nodes may change at runtime. For making full use of all available computation power, adapting the distribution to a increased or decreased number of compute nodes may be necessary.

Q8 What optimizations of the method are necessary and/or possible?

After a prototype for distributed reasoning on description logic ontologies is presented, the next step is to analyze which optimizations should be considered for a future industrial application of the method.

1.4 Overview

This work is structured into four parts. Part I is the introduction, where we motivate distributed reasoning and specify the goals and research questions that will be addressed. The preliminaries are explained in Chapter 2, including the basics of resolution reasoning and description logic. Chapter 3 starts with a typology of distributed reasoning methods and gives an overview of related work on distributed reasoning and resolution. Here, Q1 is addressed and we explain why we base our distributed reasoning approach on resolution reasoning.
The second part is the main part of this thesis. First, the idea of our approach is explained in Chapter 4 using examples from propositional logic. We prove soundness, completeness and termination of the distributed approach and thereby answer Q2 in Chapter 5. In the remaining chapters of Part II the approach is extended to a calculus for transitive properties and a calculus for equalities. The limits of applying distributed resolution on full first order logic are discussed for answering Q5. For every calculus, efficiency of distribution is investigated in experiments with real world ontologies. The results are an answer to Q3. They show runtime of a saturation can be reduced considerably. Furthermore, the scalability of the approach questioned in Q4 is investigated.
Part III addresses the allocation of input clauses and derived clauses to reasoners. Chapter 8 proposes different methods for computing the allocation. Q6 is answered by a comparison of saturation processes using different allocation algorithms. Chapter 9 addresses the distributed reasoning setting

where the number of compute nodes changes during saturation and answers Q7. The investigation shows that the allocation can be changed at runtime for adapting to a dynamic environment.
Finally, Part IV concludes with plans for future work and the summary. Chapter 10 discusses extensions and modifications of the distributed resolution approach and answers Q8. Chapter 11 summarizes the contributions and results of this thesis.

Chapter 2

Preliminaries

Before we explain the idea and details of our distributed reasoning approach, we summarize the required background on OWL, description logic, resolution and the representation of description logic ontologies in first order clauses. Additionally, an ordering of literals is defined that is relevant for efficient resolution reasoning.

2.1 RDFS and OWL

The most important standardardized languages for knowledge representation on the web are the Resource Description Framework Schema (RDFS) and the different dialects of the Web Ontology Language (OWL). OWL-DL is based on the description logic $\mathcal{SHOIN}$(D), a decidable subset of first order logic. The subset of RDFS that is relevant for the described content is ρdf, it is closely related to a restricted subset of the description logic $\mathcal{ALH}$. The main conceptual difference between RDFS and OWL languages is satisfiability. In OWL many queries are answered using an indirect proof based on refutation. In contrast, RDFS is designed for direct derivation of all implications and has no negation. Consequently, errors are not indicated by contradictions in a set of RDFS axioms. Instead, special queries can be designed for detecting unintended consequences. The different design of OWL that requires indirect proofs of implications calls for very different reasoning methods. Since OWL languages are much more expressive than RDFS, the focus of this work is on the description logics that are the basis for OWL-DL. We do not address the most expressive OWL language (OWL-full) because due to its computational complexity and difficult semantics it is rarely used. OWL-full is undecidable and the benefits from additional expressivity compared to OWL-DL are too small to play a relevant role in the OWL language area. Concequently, the new OWL standard OWL 2 is based on OWL-DL and adds contructs like cardinality restrictions and syntactic suger.

2.2 Description Logic

The most basic description logic addressed in this work is $\mathcal{ALC}$[7]. An $\mathcal{ALC}$ ontology is a set $\mathcal{O}$ of axioms α build from concepts C according to the syntax given by the following grammar:

$$\begin{aligned}
\alpha &::= C \sqsubseteq C | C \equiv C | C(a) | R(a,a) \\
C &::= \top | \bot | A | \neg C | C \sqcap C | C \sqcup C | \exists R.C | \forall R.C \\
A &::= \textit{concept_name} \\
R &::= \textit{property_name} \\
a &::= \textit{individual_name}
\end{aligned}$$

The *signature* $Sig(\mathcal{O})$ of an ontology $\mathcal{O}$ is the disjoint union of concept names N_C, property names N_R and individual names N_I.
The grammar is extended to $\mathcal{SHOIQ}$ by adding transitivity axioms ($\mathcal{S}$ denotes $\mathcal{ALC}$ plus $\mathcal{T}$), property hierarchy ($\mathcal{H}$), inverse properties ($\mathcal{I}$), nominals ($\mathcal{O}$) and qualified cardinality restrictions ($\mathcal{Q}$).

$$\begin{aligned}
\alpha &::= Trans(R) && (\mathcal{T}) \\
\alpha &::= R \sqsubseteq R && (\mathcal{H}) \\
\alpha &::= R \sqsubseteq R^- && (\mathcal{I}) \\
C &::= \{a, ..., a\} && (\mathcal{O}) \\
C &::= \exists_{\leq n} R.C | \exists_{\geq n} R.C && (\mathcal{Q})
\end{aligned}$$

The semantics of DL are defined model theoretically based on the notion of an *interpretation* $(\Delta, \mathcal{I})$ where Δ is the interpretation domain and $\mathcal{I}$ is a function that maps every individual name to an element of Δ, every concept name to a subset of Δ and every property name to a subset of $\Delta \times \Delta$. A set of DL axioms is consistent, if there is an interpretation that satisfies all axioms in the set. Such an interpretation is called a *model* of the axioms. An interpretation satisfies an axiom if it satisfies the following requirements:

- $\mathcal{I}(C) \subseteq \mathcal{I}(D)$ for an axiom $C \sqsubseteq D$
- $\mathcal{I}(C) = \mathcal{I}(D)$ for an axiom $C \equiv D$
- $\mathcal{I}(a) \in \mathcal{I}(C)$ for an axiom $C(a)$
- $(\mathcal{I}(a), \mathcal{I}(b)) \in \mathcal{I}(R)$ for an axiom $R(a,b)$
- $\mathcal{I}(R) \subseteq \mathcal{I}(S)$ for an axiom $R \sqsubseteq S$
- For an axiom $Trans(R)$ and all $a, b, c \in \Delta$:
  If $(a,b) \in \mathcal{I}(R)$ and $(b,c) \in \mathcal{I}(R)$ then $(a,c) \in \mathcal{I}(R)$

- $\forall a \in N_I : \mathcal{I}(a) \in \Delta$
- $\forall A \in N_C : \mathcal{I}(A) \subseteq \Delta$
- $\forall R \in N_R : \mathcal{I}(R) \subseteq \Delta \times \Delta$
- $\mathcal{I}(\top) = \Delta$
- $\mathcal{I}(\bot) = \{\}$
- $\mathcal{I}(\neg C) = \Delta \setminus \mathcal{I}(C)$
- $\mathcal{I}(C \sqcup D) = \mathcal{I}(C) \cup \mathcal{I}(D)$
- $\mathcal{I}(C \sqcap D) = \mathcal{I}(C) \cap \mathcal{I}(D)$
- $\mathcal{I}(\exists R.C) = \{x \in \Delta \mid \exists y \in \mathcal{I}(C) : (x, y) \in \mathcal{I}(R)\}$
- $\mathcal{I}(\forall R.C) = \mathcal{I}(\neg(\exists R.\neg C))$
- $\mathcal{I}(R^-) = \{(a, b) \in (\Delta \times \Delta) \mid (b, a) \in \mathcal{I}(R)\}$
- $\mathcal{I}(\{a_1, ..., a_n\}) = \bigcup_i \mathcal{I}\{a_i\}$
- $\mathcal{I}(\exists_{\geq n} R.C) = \{x \in \Delta \mid \#(\{y \in \Delta \mid (x, y) \in \mathcal{I}(R), y \in \mathcal{I}(C)\}) \geq n\}$
- $\mathcal{I}(\exists_{\leq n} R.C) = \mathcal{I}(\neg \exists_{\geq n+1} R.C)$

Alternatively, the semantics of $\mathcal{SHOIQ}$ can be defined by mapping the axioms to first order logic (FOL). For example, the $\mathcal{ALC}$ axiom $Car \sqsubseteq \exists part.Engine$ translates to $\forall x\colon Car(x) \rightarrow \exists y\colon part(x, y) \wedge Engine(y)$. The translation of a description logic ontology to FOL is the conjunction of the translated axioms. Note that concept names correspond to unary predicates and property names correspond to binary predicates. The normalization presented in the next section greatly simplifies the translation to first order clauses, therefore we give the simplified mapping after describing normalization.

2.3 Normalization

The resolution calculus we apply requires first order clauses as input, hence the first order formulas obtained from an ontology are translated to clauses. To guarantee termination of the applied resolution calculus, the ontology has to be normalized prior to clausification. This ensures that only certain types of axioms and corresponding clauses occur in the reasoning procedure. For simplicity, we assume the ontology contains only subsumption axioms $A \sqsubseteq C$ where A is not a complex concept and no equivalence axioms. I.e. equivalences are replaced by two subsumptions and complex subsumptions

$C \sqsubseteq D$ are replaced by $\top \sqsubseteq \neg C \sqcup D$.
The definitorial form normalization we use replaces complex concepts C in the right hand side of an axiom by a new concept name Q and adds the axiom $Q \sqsubseteq C$ to the ontology. Thus, it splits up nested axioms into simple ones by introducing new concepts. For describing the replacement formally we first define the term *position*.

Definition 1 (Position).
A position p is a sequence of integers used to specify the subterm $E|_p$ of a given expression E at position p. For the root position $p = \epsilon$, the subterm is the whole expression: $E|_\epsilon = E$. For other positions the subterm $E|_p$ of an expression $E = f(t_0, ..., t_n)$ is defined recursively: $E|_{i.p} = t_i|_p$.
The term obtained from E by replacing $E|_p$ with the term F is denoted by $E[F]_p$.

Applied to concept expressions, description logic operators correspond to functions. For example, the subterm $E|_{2.1}$ of the expression $E = A \sqcap ((\exists R.B) \sqcup C)$ is the first subterm of the second subterm of E, i.e. $E|_{2.1} = \exists R.B$. The example expression with replaced subterm is $E[F]_{2.1} = A \sqcap (F \sqcup C)$.

Definition 2 (Definitorial Form).
For simple subsumptions $A \sqsubseteq D$ with atomic concept A, and concept D in negation normal form, the Definitorial Form is defined by

$$Def(A \sqsubseteq D) := \begin{cases} \{A \sqsubseteq D\} & \text{if all subterms of } D \text{ are literal concepts} \\ \{Q \sqsubseteq D|_p\} \cup Def(A \sqsubseteq D[Q]_p) & \\ & \text{if } D|_p \text{ is not a literal concept} \end{cases}$$

where a literal concept is either a concept name or a negated concept name and Q is a new concept name.

For example, the axiom $Car \sqsubseteq Vehicle \sqcap \exists hasPart.Engine$ is replaced by $Car \sqsubseteq Vehicle \sqcap Q$ and $Q \sqsubseteq \exists hasPart.Engine$.

2.4 Clausification

After normalization, the ontology contains only simple axioms that are translated to first order clauses as defined in Table 2.1. E.g. the axiom $A \sqsubseteq B$ corresponds to the first order formula $\forall x\colon A(x) \rightarrow B(x)$ which is equivalent to the clause $\neg A(x) \vee B(x)$. As usual, all variables are implicitly $\forall$-quantified, existential quantifiers are translated using skolem functions. Literals $\bot$ (...) and $\neg\top$(...) are false (i.e. redundant literals), clauses containing a literal $\top$(...) or $\neg \bot$ (...) are tautologies (i.e. redundant clauses).

DL-axiom	FOL clause	
$C(a)$	$C(a)$	
$R(a,b)$	$R(a,b)$	
$A \sqsubseteq B$	$\neg A(x) \vee B(x)$	
$A \sqsubseteq B \sqcap C$	$\neg A(x) \vee B(x)$ $\neg A(x) \vee C(x)$	
$A \sqsubseteq B \sqcup C$	$\neg A(x) \vee B(x) \vee C(x)$	
$A \sqsubseteq \exists R.B$	$\neg A(x) \vee R(x, f(x))$ $\neg A(x) \vee B(f(x))$	
$A \sqsubseteq \forall R.B$	$\neg A(x) \vee \neg R(x,y) \vee B(y)$	
$Trans(R)$	$\neg R(x,y) \vee \neg R(y,z) \vee R(x,z)$	$(\mathcal{T})$
$R \sqsubseteq S$	$\neg R(x,y) \vee S(x,y)$	$(\mathcal{H})$
$A \sqsubseteq \{a_1, ..., a_n\}$	$\neg A(x) \vee x = a_1 \vee ... \vee x = a_n$	$(\mathcal{O})$
$A \sqsubseteq \exists R.\{a\}$	$\neg A(x) \vee R(x,a)$	$(\mathcal{O})$
$R \sqsubseteq S^-$	$\neg R(x,y) \vee S(y,x)$	$(\mathcal{I})$
$A \sqsubseteq \exists_{\leq n} R.B$	$\neg A(x) \vee \bigvee_{i=1}^{n+1} (\neg R(x,y_i) \vee \neg B(y_i) \vee \bigvee_{j=1}^{i-1} y_i = y_j)$	$(\mathcal{Q})$
$A \sqsubseteq \exists_{\geq n} R.B$	$\neg A(x) \vee R(x, f_i(x)) \quad i = 1..n$ $\neg A(x) \vee f_i(x) \neq f_j(x) \quad j = 1..i-1$ $\neg A(x) \vee B(f_i(x))$	$(\mathcal{Q})$

Table 2.1: Translation of simple DL axioms to first order clauses.

2.5 Resolution

The simplest type of resolution is the variant for propositional logic. Our distributed reasoning approach is designed for more expressive logics, but for explaining the basic idea, we use propositional logic. Resolution is only refutation complete but not implication complete, i.e. the calculus does not derive every implied clause. But, if resolution is applied to a set of unsatisfiable clauses, it will eventually derive an empty clause and thereby prove the contradiction. In this work, with "completess" we refer to refutation completess.

Definition 3 (Ordered Propositional Resolution).
Based on any total precedence $>_p$ on propositional literals with

- *$\neg b > b$ for every propositional variable b and*
- *there is no literal a such that for some variable b: $\neg b > a > b$*

ordered propositional resolution is defined for propositional variable b and clauses C, D by

$$\frac{b \vee C \qquad \neg b \vee D}{C \vee D}$$

where b (respectively $\neg b$) is a maximal literal in the clause $b \vee C$ ($\neg b \vee D$) according to the precedence $>_p$.

The literals b and $\neg b$ are *resolved*. The clauses above the horizontal line are the *premises*. The premise $b \vee C$ with positive resolved literal is the *side premise*, the premise with negative literal is the *main premise*. Below the line is the *conclusion* that is inferred from the premises.
An additional rule is necessary to remove duplicated literals in a clause.

Definition 4 (Ordered Propositional Factoring).
For a propositional variable b and clause C, propositional factoring is defined by

$$\frac{b \vee b \vee C}{b \vee C}$$

In propositional logic, the premise $b \vee b \vee C$ is redundant after inferring $b \vee C$ and can be deleted. Instead of first performing the inference and than deleting the premise, the duplicate literal is deleted directly from the premise.
Without the restriction to maximal literals, resolution is not efficient because much more clauses than necessary are derived. For example, consider the propositional clauses $a \vee b \vee c$, $\neg a \vee b$, $\neg b \vee c$, $\neg c$. With alphabetical ordering, resolution derives only $b \vee c$ and c and then detects the contradiction with $\neg c$. $a \vee b \vee c$ is not resolved with $\neg b \vee c$ because b is not maximal in the first clause. But, without ordering, also $a \vee c$, $\neg a \vee c$, $a \vee b$, b and $\neg b$ can be derived. To avoid redundant derivations, the propositional variables are ordered according to some precedence and only the maximal variables are resolved.

2.6 Term Ordering

When resolution is lifted to first order logic, the clauses consist of literals instead of propositional variables. Hence, first order resolution requires an ordering of literals. Different ordering variants are possible for first order resolution, we use lexicographic path orderings that are based on a precedence of the function and predicate symbols.

Definition 5 (Lexicographic Path Ordering).
A lexicographic path ordering (LPO) is a term ordering $\succ$ induced by a well-founded strict precedence $>$ over function, predicate and logical symbols. For terms s and t, $s \succ t$ holds if and only if

1. *t is a proper subterm of s or*
2. *$s = f(s_1, \ldots, s_m), t = g(t_1, \ldots, t_n)$ and at least one of the following holds*
   - *(i) $f > g$ and $s \succ t_i$ for all i with $1 \leq i \leq n$*
   - *(ii) $f = g$ and for some j we have $(s_1, \ldots, s_{j-1}) = (t_1, \ldots, t_{j-1})$, $s_j \succ t_j$ and $s \succ t_k$ for all k with $j < k \leq n$*
   - *(iii) $s_j \succeq t$ for some j with $1 \leq j \leq m$*

LPOs have the subterm property, i.e. $t \succ t'$ for all terms t' that are subterms of term t. Furthermore, if $>$ is total, the LPO induced by $>$ is total on ground terms.

Definition 6 (Admissible Ordering).
An ordering $\succ$ is admissible if it is

- *well-founded, stable under substitutions, and total on ground literals,*
- *$\neg A \succ A$ for all ground atoms A*
- *$B \succ A$ implies $B \succ \neg A$ for all atoms A and B.*

The ordering of equality literals is defined by the lexicographic comparison of corresponding tuples $(arg_{max}, rel, arg_{min})$ where arg_{max} and arg_{min} are the larger and smaller argument of the equality literal and rel is either '$\approx$' or '$\not\approx$' (predicates are represented as equalities). The comparison of arguments is defined by the term ordering, '$\approx$' precedes '$\not\approx$'.

For simplifying the definition of calculi and ordering of literals, predicate literals are represented by equalities as follows: The literal $P(t_1, t_2)$ is represented by the equality $P'(t_1, t_2) \approx \top$ where P' is the function that maps (t_1, t_2) to $\top$ iff $P(t_1, t_2)$ is true. Similar translations apply for unary predicates. Hence we can assume that all literals are equalities without loss of generality. We will still use the term 'predicate literal' for literals of the type $P'(t_1, t_2) \approx \top$ and the term 'equality literal' or 'equality' for equality literals that do not correspond to predicate literals.
For example, consider the comparison of the literals $P(f(x))$ and $Q(x)$. Since both are predicate literals, the comparison of the corresponding tuples $(P(f(x)), \approx, \top)$ and $(Q(x), \approx, \top)$ boils down to comparing $P(f(x))$ and $Q(x)$. From (1.) in Definition 5 we get $f(x) \succ x$ because x is a proper

subterm of $f(x)$. Assuming alphabetical order of functions and predicates ($f > Q$) we apply (i) to $s = f(x), t = Q(x)$ and obtain $f(x) \succ Q(x)$. Now we use (iii) to conclude $P(f(x)) \succ Q(x)$.
The ordering of literals is extended to an ordering of clauses by representing clauses as sets of literals. A multiset is a set that may contain more than one occurrence of each element. Multisets are represented by a mapping from set elements to natural numbers denoting the number of occurrences of each element.

Definition 7 (Multiset Extension of Ordering).
For multisets of literals A and B, $A \succ B$ iff $A \neq B$ and for each literal l_1 with $B(l_1) > A(l_1)$ there is a literal l_2 such that $l_2 \succ l_1$ and $A(l_2) > B(l_2)$.

2.7 Redundancy

Based on the ordering of clauses, ordered resolution derives smaller clauses until the smallest derivable clause is reached. If the smallest clause is the empty clause, the input clauses are unsatisfiable. Consequently, the order is also relevant for the definition of redundancy:

Definition 8 (Redundant Clause).
In a set of clauses $\mathcal{S}$, a clause $c \in \mathcal{S}$ is redundant if there are clauses $c_1, ..., c_n \in \mathcal{S}$ such that $c_1, ..., c_n \vdash c$ and $c \succ c_i$.

For efficient resolution reasoning it is essential to remove redundant clauses by the application of reduction rules. When a reduction rule is applied, the premises are removed from the clauses set, in all other aspects it is similar to inference rules. The most important reduction rule is subsumption reduction.

Definition 9 (Subsumption Reduction).
For clauses C, and D where C contains all literals of D, subsumption reduction deletes C:

$$\frac{C \quad D}{D}$$

We assume resolution rules are not applied to a set of premises, if the conclusion is redundant. Exhaustive application of resolution rules to a set of clauses results in a saturated set of clauses.

Definition 10 (Saturation).
A set of clauses S is saturated by a calculus R, iff every clause c that can be derived from S using rules of R is redundant in S.

For simplifying the notation, we adapt the notion of completeness for calculi:

Definition 11 (Complete Calculus).
A calculus R is complete, if for every set S of clauses that is saturated by R: If S is unsatisfiable, then S contains the empty clause.

Note that the empty clause is never redundant because it is the smallest clause. All calculi used in this thesis are sound, i.e. if a set of clauses S is satisfiable then no empty clause is derived.

Chapter 3

Related Work

Before the summarizing work on reasoning that is related to our approach, we give a typology of distributed reasoning methods. Subsequently, reasoning and distributed reasoning approaches for different languages are described. We conclude the chapter with frameworks for implementing distributed computation.

3.1 Typology of Distributed Reasoning Methods

There are various options for distributing the process of logical reasoning. Many of these options have been investigated in the field of automated theorem proving for first-order logic [13, 12]. In the following we discuss these options and their advantages and disadvantages with respect to the requirements and goals defined in Chapter 1. In particular, we have to make two choices:

1. We have to choose a reasoning method that is sound and complete for description logics and permits distribution.

2. We have to choose a distribution principle that supports local reasoning and minimizes reasoning and communication costs.

3.1.1 Reasoning

Concerning the reasoning method [12] distinguishes between ordering-based, subgoal reduction, and instance-based strategies. Instance-based strategies are direct implementations of the Herbrand method that generate ground instances of the theory and use propositional methods for testing satisfiability. Subgoal reduction strategies build a single proof at a time by choosing and resolving subgoals and leave the logical model unchanged. Typical examples of subgoal reduction strategies are logic programming methods and

analytic tableaux. Ordering-based methods, finally are based on an informed modification of a clause representation by deriving new clauses and deleting redundant ones until the empty clause is derived or no new conclusions can be drawn. This way, ordering-based methods implicitly build many proof attempts in parallel as it is not clear a priori, which derivations finally contribute to the derivation of the empty clause. Typical examples of ordering-based methods are resolution and basic superposition.

Analytic tableaux are the dominant method for implementing sound and complete inference systems for description logics [20]. It has been shown, however, that sound and complete resolution methods for expressive description logics can be defined [58, 36]. We exclude other existing methods such as a reduction of DL reasoning to logic programming from our investigation because these approaches are not sound and complete for the languages we are interested in. Because tableaux-based as well as resolution-based methods meet our requirements with respect to language coverage and completeness, the decisive factor is their suitability for distributed reasoning.

3.1.2 Distribution Principles

The survey [12] discusses different strategies for parallelizing logical inference. In particular, the authors distinguish between parallelism at the term level, the clause level and the search level. The idea of parallelism at the term and the clause level is to speed up basic reasoning functions such as matching, unification or single resolution steps by executing them in parallel using a shared memory in order to improve performance. This approach is not suitable for our purposes as it does not envision a distribution of the ontology axioms but just aims at parallel execution of basic reasoning methods. Without shared memory this approach is not feasible because it requires to much interaction between processes. Parallelism at the search level means the parallel execution of the overall derivation process and can be further distinguished into multi-search and distributed search approaches.

Multi-search In the first case, multiple search processes, often with different heuristics or different starting points are run in parallel. This approach requires the complete ontology to be available at all reasoners. It exploits the fact that most reasoning tasks belong to tractable problem classes where an efficient solution strategy is available. The task is hard because we do not know in advance, to which simple class the problem belongs. Multi-search replaces the expensive analysis and classification of the reasoning task by applying different strategies in parallel, that are optimized for different problem classes. Using this approach for description logic is not advisable because description logics are a decidable subset of first-order logic for which

efficient reasoning procedures are known. I.e., we already know which of the strategies that we could run in parallel performs best.

Distributed search The second type of parallel search approaches assign parts of the search space to individual reasoners and coordinate the overall search process by exchanging intermediate results. The distributed search paradigm naturally fits the distributed storage of parts of the ontology and therefore represents a paradigm that complies to the goals of our research. It allows to assign the part of the search space relevant for a specific ontology module to a local reasoner instance that interacts with other local reasoners if necessary.

The choice of the distributed search paradigm has consequences for the choice of the reasoning method. In particular, it has been shown that distributed search can be used in combination with ordering-based methods [15, 11] to support parallel execution of logical reasoning.

3.2 Distributed RDF Reasoning

Approaches that implement distributed reasoning on description logic ontologies are rare. Distribution of RDFS ontologies is more popular because adapting the rule based inference mechanism of RDFS ontologies to a distributed setting is easier than distributing the tableau methods that are usually used for description logic.
Marvin [40] is a platform for parallel and distributed processing of RDF data on a network of loosely-coupled peers. Successful experiments are reported for computation of the deductive closure of large RDFS ontologies. Although there are no contradictions and proofs in RDFS, the process of computing the deductive closure is similar to saturation of a set of clauses. Hence, the method implemented by Marvin can be classified as distributed search strategy.

3.3 Modular DL Reasoning

Current approaches to distributed reasoning on description logic mostly rely on tableaux methods. Distribution by solving the different choices of nondeterministic tableaux rules in parallel is difficult as it hampers the application of optimization and blocking strategies. Instead, most distributed tableaux approaches try to identify all possible conflicts, i.e. all axioms that might follow from another module and would cause a contradiction and send these as queries to the other modules. So far, this is only done for links with rather restricted expressiveness between the modules.

The most prominent actually distributed terminology reasoning implementation for ontologies is *Distributed Description Logic* (DDL [14]), a distributed search strategy. It supports only a special type of links (*bridge rules*) between ontologies. The local domains have to be disjoint, i.e. there is no real subsumption between elements of different modules.
Like DDL, $\mathcal{E}$*-connections* [17] treat local domains as disjoint and do not support subsumption relations between modules. $\mathcal{E}$-connections can be used to link ontologies of different expressivity and the resulting network can be translated to common description logics. The tableau reasoner pellet has been extended to support $\mathcal{E}$-connections but reasoning is performed by a single pellet reasoner and not distributed.
[35] propose an approach to modular ontologies based on *conservative extensions*. It is not a distributed reasoning method but a strategy for creating ontology networks with self-contained ontologies such that reasoning on the ontologies separately is complete. Roughly speaking, an ontology A is an conservative extension of another ontology B, if there is no axiom implied by A and not implied by B that uses only symbols from the local signature of B. Hence, for reasoning in the combined ontology about symbols of B, the ontology A can be safely ignored, it has no effect on the semantics of B. The idea of the approach is, that reasoning is very simple, when the ontology network is a conservative extension of each of the ontologies. While the actual reasoning is relatively easy in this approach, the complex task is to create the ontologies in compliance to the strong conservative extension requirements. A method for extracting this type of self-contained ontology modules from a larger ontology is proposed in [16].
KAONp2p [24] is an infrastructure for query answering over distributed ontologies based on the KAON2 system. The approach is focused on ontology management and knowledge selection for reasoning about assertions (Aboxes). The relevant parts of the terminologies are copied to the peer that answers the query.

3.4 Resolution Methods

Resolution is a very popular reasoning method for first-order logic (FOL) provers. As description logics are a strict subset of first-order logic, resolution can be applied to description logic ontologies as well [60]. For this purpose the DL ontology is transformed into a set of first-order clauses as defined in Section 2.4. This translation from DL axioms to clauses can be done on a per axiom basis independently of other parts of the ontology.
[30] proposed a resolution variant that is sound and complete for clauses obtained from ontologies in the DL $\mathcal{ALCHIQ}$. This resolution method is discussed in detail in the next chapters because our distributed approach is based on it.

Approaches to distributed first-order reasoning are motivated by efficiency considerations, performance is improved by using multiple processors in parallel. *Roo*[34], for example, is a parallelization of the widely (re)used first-order reasoner Otter. While common resolution provers pick one so-called *given* clause and resolve it with all possible partner clauses, Roo picks multiple given clauses in parallel and solves arising conflicts of the parallel processes. In difference to the distributed methods mentioned so far, Roo uses a shared memory and can be classified as parallelization at the clause level according to [12].

Partition-Based Reasoning [3] is a distributed search strategy that requires local reasoning to be complete for consequence finding. The strong completeness requirement for local reasoning inevitably causes derivation of more clauses than necessary for refutation. Nevertheless the distribution method was shown to speed up some resolution strategies in a parallel setting with shared memory. Note that the resolution calculi applied for our distributed reasoning method are not complete for consequence finding, for efficiency considerations they are designed to generate a relatively small number of clauses.

An approach for distributed resolution reasoning on equational logic is proposed in [11], it uses the ancestor-graph criterion for allocating clauses to processes. Comparision to a random allocation of clauses shows the allocation based on heuristics performs better.

[1] propose a distributed reasoning method for propositional theories. The authors automatically create sets of clausal theories connected by shared variables and investigate the correlations between characteristics of the created theory networks and queries (e.g. number and length of link clauses) and characteristics of the query processing (e.g. depth of a query). The query runtimes are investigated for different complexity of theory networks and queries but not compared to centralized computation.

3.5 Parallel Computation

Apart from work on reasoning and distributed reasoning, methods for parallel computation are relevant to our work. Recently, the MapReduce framework developed by Google labs is popular for implementation of distributed reasoning methods. However, it imposes some restrictions on the way data is exchanged between compute nodes. The more flexible actor model was proposed decades ago. It is still in use, a current project provides robust implementation.

3.5.1 MapReduce

The MapReduce framework[19] provides a simple interface for cluster computation. An open source implementation is available[1]. Low effort for distributing a computation to multiple machines has motivated a number of distributed reasoning implementations that apply the MapReduce framework. Core of the framework and interface for its application are two functions that have to be implemented by the user to access the automatic distribution. Both functions are executed by a set of *workers* (i.e. machines) that share the computation load. First, the *map* function assigns a key to each input value (for reasoning applications the values are axioms) and outputs $(key, value)$ pairs. Then, the *reduce* function is called once for each key. It processes all corresponding values and outputs a list of results. For reasoning, parallel execution of the map function is not essential. Only the keys generated by the map function are required for distributing the work for the reduce workers. A partition function assigns the keys of the map output to reduce workers. Application of inference rules is implemented in the reduce function, parallel execution of this function is the motivation for applying MapReduce.
Originally, MapReduce was developed for generating indexes over web pages, a computation that has a very large input and performs mainly a counting function. Hence, the reduce function effectively reduced the number of values. In contrast, reasoning applications of MapReduce add new axioms to the input axioms, i.e. the reduce function actually expands the input. This can cause efficiency problems when a sequence of MapReduce jobs is executed for distributed reasoning.

RDF Schema Materialization

One of the first applications of MapReduce in ontology reasoning is the computation of the closure of a large RDFS graph described in [62]. RDF Schema rules are implemented by MapReduce jobs. For example, the RDFS subclass rule

$$\frac{s \text{ rdf:type } x \quad x \text{ rdf:subClassOf } y}{s \text{ rdf:type } y}$$

is implemented by a map function that maps potential premises to the shared element x. I.e., the key for triples with predicate "rdf:type" is the object, the key for triples with predicate "rdfs:subClassOf" is the subject of the triple. The whole triple is returned as value of the map output pair. The reduce function is called once for each key and derives new axioms according to the subclass rule from all triples that share this key. Note that a single call to this job performs all derivations of this rule. The work for deriving all implied triples of type $(s, \text{rfd:type}, o)$ is partitioned among the reduce

[1] e.g. Apache Hadoop `http://hadoop.apache.org`

workers based on the objects o that are the keys in the input to the reduce function.
The other RDFS rules are implemented by MapReduce jobs in a similar way. The complete materialization consists of a sequence of MapReduce jobs, where the output of one job is the input of the next job. As shown in [62], this method is quite efficient when the number of schema triples is small enough to be stored in memory of each reducer node. With clever ordering of the RDFS rules, the materialization is usually[2] complete after calling each job once. Hence, only a handful of MapReduce jobs is necessary for materialization of the deductive closure.

OWL Horst Materialization

The RDFS materialization was extended to OWL Horst in [61]. OWL Horst [59] is a fragment of the Web Ontology language OWL that can be materialized using a set of rules that is an extension of the set of RDF schema rules. The fragment is popular for triple stores that are focused on scalability because of the relatively high expressivity and feasible reasoning methods. The additional rules add semantics for the OWL constructs "owl:someValuesFrom", "owl:allValuesFrom" and "owl:TransitiveProperty". The higher expressivity of OWL Horst compared to RDFS requires a couple of optimizations to keep tractability. While for RDFS it is possible to have a single 'stream' of instance triples for each reduce worker, OWL Horst requires joins over more than one instance triple. The number of necessary expensive joins is reduced by storing the "owl:sameAs" triples only implicitly and other optimizations for transitive properties and property restrictions. With these optimizations, the authors were able to compute the closure of 100 billion triples. However, some inefficiencies were detected: For OWL Horst rules, there is no order that can avoid the need for iterating repeatedly over all rules. As the authors report, this is problematic because the same conclusions are derived again and again in every iteration.

$\mathcal{EL}+$ Classification

$\mathcal{EL}+$ [6] is a fragment of OWL that does not contain union operators or universal restrictions on properties. Concepts in $\mathcal{EL}+$ are built according to the grammar

$$C ::= A|\top|C \sqcap D|\exists r.C,$$

where A is a concept name, r is a role name and C, D are concept names or complex concepts. In addition to general concept inclusions $C \sqsubseteq D$ and assertions, an $\mathcal{EL}+$ ontology may contain role inclusions $r_1 \circ ... \circ r_n \sqsubseteq r$ where

[2]For certain cases (e.g. if subproperties of 'rdf:SubpropertyOf' are defined) that are very rare in real world ontologies, repeated application of the rule sequence is necessary for completeness.

$r, r_1, ..., r_n$ are role names. The nice property of $\mathcal{EL}+$ is the existence of a simple set of derivation rules that allows classification of $\mathcal{EL}+$ ontologies in polynomial time. For example, the rule

$$\frac{X \sqsubseteq A \quad A \sqsubseteq \exists r.B}{X \sqsubseteq \exists r.B}$$

propagates a restriction on a class A to the subclass X of A. Motivated by the materialization approaches mentioned before, [37] proposes a MapReduce variant of the $\mathcal{EL}+$ classification algorithm CEL. The derivation rules of CEL are translated to MapReduce jobs. Before the translation, the rules are slightly adapted, such that for every rule all premises share at least one class or property name. The shared terms are used as key in the input of the reduce function (output of the map function) similar to the RDFS materialization. For the above rule, axioms $A \sqsubseteq B$ are assigned the key B and restrictions $A \sqsubseteq \exists r.B$ are assigned the key A. The reduce workers derive new axioms from sets of axioms that share the same key. In contrast to the previous approaches, only the input to the reduce function is considered as premises and this set of potential premises is not changed while the reduce worker runs. Recall that in the RDFS materialization, *all* applications of a certain rule are executed in a single MapReduce job. In $\mathcal{EL}+$ classification, an axiom derived by a reduce worker can only be considered as premise in the next job. Hence, the number of required MapReduce jobs is at least the depth of the derivation graph. Another difference to previous approaches is the maintenance of the axiom set. The authors propose to store the axioms in a database instead of the files that are used by, e.g., the Hadoop implementation of MapReduce.
The approach suffers from an unsolved efficiency issue: Rules of the underlying CEL algorithm are only applied, if the conclusion is not already contained in the current axiom set. But, in the MapReduce variant of the algorithm, the authors do not report how this preconditions are checked and the preconditions are not mentioned in the adapted rules set. We assume, that the database that is used for storing intermediate results deletes duplicate axioms. But anyway, if already derived axioms are repeatedly derived in every iteration, the method is inefficient, especially because the number of iterations is very high as mentioned before.

The general problem with reasoning applications of the MapReduce framework is iterative execution of MapReduce jobs. MapReduce has no built in strategy for avoiding repetition in the computation. Originally, MapReduce was designed for the creation of indexes where repetition is not an issue, e.g., word counts are completed after a single run of a MapReduce job. In contrast, reasoning applications require iterative application of inference rules. But, when a MapReduce job is executed repeatedly, already performed inferences are also repeated. This problem has a severe effect on

performance as the 'reduce' of reasoning applications actually expands the input by materializing implied axioms. In [50] we showed that repeating inferences can be avoided by maintaining separate lists for axioms that are already exhaustively used in derivation rules and axioms that require application of rules. However, this implementation is not very efficient because the complete axiom lists have to be written and parsed in every run of a job. To sum up, the simplicity of the MapReduce interface implies restrictions that are not desirable for distributed reasoning.

3.5.2 Actor Model

Frameworks that use message passing are more flexible regarding the interaction between compute nodes. E.g., [27] proposed the actor model for parallel computation, where a computational unit is an actor that communicates with other actors via messages. Each actor has its own life cycle, local state and local message queue. Besides sending and reacting on messages actors have access to their own local memory. In difference to MapReduce, this application of the message passing paradigm allows fast integration of results from other reasoners into the local computation. It is not necessary to split the computation into a sequence of jobs, where results from other actors are only available in the next job. Consequently, the costly serialization and parsing of axioms between jobs is avoided.
Based on the distributed reasoning method developed for this thesis, [39] provides an implementation that uses the AKKA[3] implementation of the actor model. Experiments showed a considerable speedup achieved by distribution, but the basic implementation of the ordered resolution calculus is relatively slow compared to state of the art theorem provers.

3.6 Conclusion

The distribution principle that is most suitable for our setting is distributed search. The choice of the distributed search paradigm has consequences for the choice of the reasoning method. In particular, it has been shown that distributed search can be used in combination with ordering-based methods. We build on top of these results by proposing distributed reasoning methods based on the principles of resolution.
Our proposal extends beyond the state of the art in distributed theorem proving as it addresses specific decidable subsets of first-order logic that have not yet been investigated in the context of distributed theorem proving. Further, existing strategies for assigning inference steps to reasoners such as the ancestor-graph criterion [11] or partition-based reasoning [3] cannot avoid redundancy. Other approaches are limited to propositional

[3] `http://akka.io/`

logic [1] or apply parallelism on the term and clause level that is not compatible with the requirement of distributed axioms. We propose a sound and complete distributed reasoning method that uses an assignment of clauses to reasoners without redundancy. Our method takes advantage of the restrictions imposed by description logics and can be used in combination with different calculi.

Approaches for distributed RDFS reasoning are not applicable for our setting, because they do not support the required expressivity. Extension towards description logics is problematic because RDFS reasoning requires materialization. In contrast, indirect proofs are usually preferred for description logic because complete materialization of implied axioms is not efficient.
Existing approaches for modular DL reasoning are not designed for parallel computation but focused on the combination of different ontologies. There are strong limits on the axioms that contain concepts from different ontology modules. To the best of our knowledge, there is no distributed reasoning approach for expressive description logic ontologies that achieved a performance enhancement by distributing the computation. Distributed $\mathcal{EL}+$ classification [37] aimed in this direction, but the theory has serious flaws and the method was not implemented.
For implementing distributed reasoning, the MapReduce framework is not suitable because it imposes too strong restrictions on the communication between reasoners. A better choice is the actor model proposed by [27]. Most promising is implementing distributed reasoning using a state of the art theorem prover for local reasoning and the actor model for distributing computation.

The method for distributed reasoning on description logic ontologies proposed in this thesis is based on resolution methods for description logic proposed by [58, 36, 30].

Part II

Distributed Resolution

Chapter 4

Distributed Resolution

In this chapter we describe the basic idea of distributed resolution using propositional logic. By restricting the expressivity we focus on the explanation of the basic idea and avoid a couple of difficulties introduced by more expressive axioms.
The idea of distributed resolution is to partition the input clauses into separate sets and run a reasoner on every part of the input. In particular, the inferences are distributed across different reasoners, thereby increasing the available computation resources:

- Every reasoner separately saturates the clause set assigned to it.
- Newly derived clauses are propagated to other reasoners if necessary.

In contrast to the centralized case, a reasoner that has saturated the local clause set may have to continue reasoning once a new clause is received from another reasoner. The whole system of connected reasoners stops if the empty clause ($\square$) is derived by one of the reasoners or all reasoners are locally saturated.

4.1 Reasoning Method

We aim at distributing different resolution methods for checking satisfiability of a set of clauses. Resolution methods use rules for deriving new clauses that are added to the input clause set until either a contradiction is found or no new clause can be derived.

Definition 12 (Resolution Rule).
A resolution calculus for a class of clauses $\mathcal{C}$ consists of a set R of resolution rules. Each rule is a function r that maps each list $p_1, ..., p_n$ of premises with $p_i \in \mathcal{C}$ to a set $r(p_1, ..., p_n) \subseteq \mathcal{C}$ of conclusions. If the set of conclusions $r(p_1, ..., p_n)$ is non-empty, the rule r is applicable *to the premises P. A*

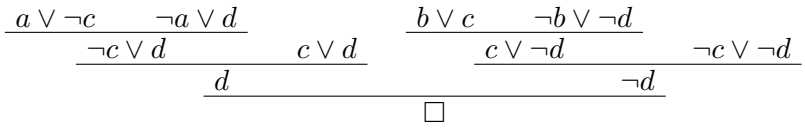

Figure 4.1: Example of propositional ordered refutation.

set of clauses C is saturated *iff every conclusion $c \in r(p_1, ..., p_n)$ with $r \in R, p_i \in C$ is redundant with respect to C.*

Mostly, resolution rules have only a single conclusion. Reduction rules are resolution rules where the premises are always redundant after the conclusions have been added. Hence, the premises are deleted on application of reduction rules.
We write $C \vdash_R c$ to denote that the conclusion c is derived from the set of clauses C by repeated application of resolution rules from the calculus R. Additionally, $C \vdash_R c$ holds for $c \in C$.
To keep the presentation simple, we first use propositional ordered resolution as defined in Section 2.5. An example refutation with six input clauses and five inferences is depicted in Figure 4.1. The clause $\neg c \vee d$ is inferred from the two input clauses $a \vee \neg c$ and $\neg a \vee d$ and then resolved with the third input clause $c \vee d$ to obtain the conclusion d. In a similar way, $\neg d$ is derived from three other input clauses. Finally, resolving d with $\neg d$ results in an empty clause, i.e. a contradiction is found because the input clauses are unsatisfiable.

4.2 Allocation

In theory, every inference (depicted by a horizontal line in Figure 4.1) can be performed by a different reasoner. But, the premises of each inference must be propagated to the reasoner that performs the inference. If reasoner 1 derives $\neg c \vee d$ and reasoner 2 is supposed to derive d, the clause $\neg c \vee d$ has to be propagated from reasoner 1 to reasoner 2. In general, a conclusion c has to be send to every reasoner that performs an inference were c is a premise.

Definition 13 (Inference Allocation).
Inferences are identified by tuples $(r, p_1, ..., p_n)$ where $r \in R$ is the inference rule applied to the premises $p_1, ..., p_n$. The inference allocation function $ia : R \times \mathcal{C}^n \to M$ maps each applicable inference to a reasoner $m \in M$.

E.g., allocation of the first inference of Figure 4.1 to reasoner 1 is stated by $ia(propositionalOrderedResolution, a \vee \neg c, \neg a \vee d) = 1$. For enabling a derivation, the premises have to be allocated to the reasoner that performs an inference.

Definition 14 (Clause Allocation).
A clause allocation is a relation $ca \in (\mathcal{C} \times M)$ *that maps clauses* $c \in \mathcal{C}$ *to reasoners* $m \in M$ *such that*

$$\forall c \in \mathcal{C} \colon \exists m \in M \colon ca(c, m)$$

The set of modules a clause c *is allocated to by the allocation* ca *is*

$$ca(c) := \{m \in M \mid ca(c, m)\}$$

If the allocation relation is functional we may omit the parenthesis and write $ca(c) = m$ instead of $ca(c) = \{m\}$. We say a reasoner m is responsible for an inference or clause x iff x is allocated to m.
Obviously, distributing inference steps randomly to reasoners is not a good idea, because we would not know to which reasoners a clause has to be propagated. Nevertheless, we have to make sure the clause allocation is complete, otherwise possible inferences may be skipped, the method would be incomplete.

Definition 15 (Complete Clause Allocation).
A clause allocation ca *is complete for a calculus* R*, iff for every applicable inference* $(r, p_1, ..., p_n)$ *with* $r \in R$ *every premise* p_i *is allocated to the same reasoner:* $\exists m \in M, \forall i : ca(p_i, m)$

For inferences (r, p) with a single premise, this condition is trivially true. In practice, ca is defined in a way that ensures completeness for rules with more than one premises. Then, inferences with a single premise p are allocated to the reasoner that derived the clause p or to the reasoner $ca(p)$.

The challenge is to find an inference allocation and corresponding clause allocation that is complete, efficiently computable and does not require too many propagations of clauses. Only a complete clause allocation can guarantee that the distributed reasoning method is complete. Furthermore, moving clauses between reasoners is costly, hence we prefer allocations where clauses are often allocated to the reasoner that derived them. Ideally, the clause allocation is functional, i.e. each clause is allocated to exactly one reasoner. Allocating a clause to multiple reasoners potentially causes duplicated inferences and additional computation that should be avoided.
Essential is the availability of a simple algorithm for computing the allocation of a clause. It is not possible to analyze the possible partner clauses in other reasoners because communication would consume the benefits of parallel computation.
We solve this problem by defining an allocation of inferences, that allows deciding the allocation of a clause independently of the other clauses. The idea is to find an allocation of inferences, that induces a partition of clauses

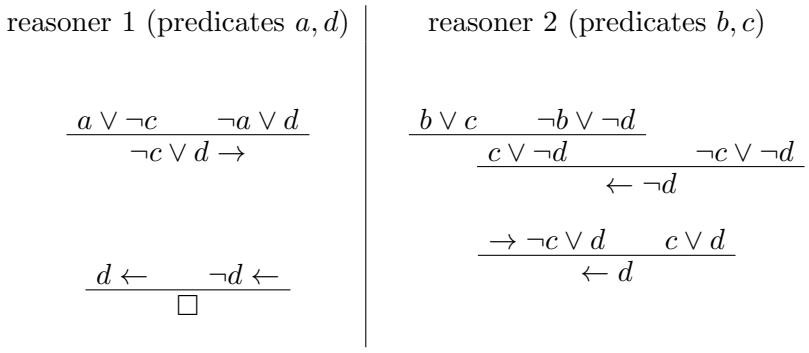

Figure 4.2: Distributed propositional refutation.

into disjoint sets and each clause is resolved only with clauses from the same part.
For example, for propositional resolution, we partition the inferences according to the resolved literals. In any propositional resolution inference, the two premises share a predicate p that occurs positive in one premise and negative in the other premise. Hence, we can allocate the inferences based on an allocation of symbols.

Definition 16 (Symbol Allocation).
A symbol allocation for a set of clauses $\mathcal{C}$ is a function $sa : Sig(\mathcal{C}) \rightarrow M$ that maps each symbol of the signature of $\mathcal{C}$ to a reasoner $m \in M$. For a set of symbols S, $sa(S)$ denotes the set $\{sa(s) \mid s \in S\}$.

Each reasoner m is responsible for the inferences that involve resolving literals p or $\neg p$ where $sa(p) = m$. In our example, one reasoner is responsible for the predicates a, d and another reasoner is responsible for b, c. Of the input clauses $a \vee \neg c$, $\neg a \vee d$, $b \vee c$, $\neg b \vee \neg d$, $\neg c \vee \neg d$, $\neg c \vee \neg d$ the first two are allocated to reasoner 1 and the others to reasoner 2. The corresponding distributed refutation is depicted in Figure 4.2. Arrows depict propagation between reasoners. Reasoner 2 derives $c \vee \neg d$ and then $\neg d$ which is send to reasoner 1. At the same time reasoner 1 derives the clause $\neg c \vee d$ and sends it to reasoner 2. From $\neg c \vee d$ and the local input clause $c \vee d$, reasoner 2 derives and sends d. Finally, reasoner 1 detects the contradiction by deriving an empty clause.
For unrestricted propositional resolution, we have to allocate a clause c to every reasoner that is responsible for one of the predicates contained in c. But, without restrictions, propositional resolution is not efficient anyway. If we add the restrictions for ordered propositional resolution, the calculus is not only much more efficient, it also enables an efficient allocation of clauses: Ordered resolution uses a total precedence $>_p$ of predicate symbols. The resolution rule is restricted to pairs of premises, where the matched literals

Algorithm 1 Distributed Resolution

isSatisfiable(KB)

1: $[Us_0, Us_1, ..., Us_{n-1}] \leftarrow partition(KB, n, ca)$ // partition knowledge base according to allocation relation into array of n sets of clauses
2: **for all** i **in** $\{0, ..., n-1\}$ **do**
3: // loops are executed in parallel by n reasoners
4: **if** isSatisfiable($Us[i], \emptyset, i$) == FALSE **then**
5: **return** *FALSE*
6: **end if**
7: **end for**
8: **return** *TRUE*

p and $\neg p$ are maximal, i.e. no premise contains a predicate q with $q >_p p$. Hence, we allocate each clause c to the reasoner that is responsible for the largest predicate of c. Without loss of generality we assume each predicate occurs at most once in each clause. Duplicated literals are removed by factoring, clauses that contain literals p and $\neg p$ are tautologies, and can be deleted.
Like for propositional resolution, the distribution of other resolution calculi is based on an allocation of symbols. In general, a function that selects relevant symbols from a clause for allocation extends the allocation of symbols to an allocation of clauses.

Definition 17 (Clause Allocation Based on Allocation Symbols).
The allocation $ca(c)$ of a clause c is based on an allocation as of the signature symbols and a function a-symbol *that maps each clause c to a subset of $Sig(c)$.*

$$ca(c) = \{m \in M \mid \exists S \in \text{a-symbol}(c) : sa(S) = m\}$$

In the case of propositional resolution, a-symbol(c) is the largest predicate of c. More expressive logics require more complex definitions of a-symbol.
Next, we explain how a resolution prover is turned into a distributed resolution prover using the appropriate clause allocation function.

4.3 Distributed Algorithm

The distributed saturation process is controlled by Algorithm 1. It distributes the input clauses to the reasoners according to the given allocation ca of clauses. Each reasoner is started on the local clause set, and derives new clauses by applying resolution rules (i.e. it runs Algorithm 2 locally). If an empty clause is derived by one of the reasoners, it returns FALSE, otherwise all reasoners continue until they are saturated and return TRUE.

Algorithm 2 Distributed Resolution Prover

isSatisfiable(Us, Wo, $localID$)

```
1: while TRUE do
2:   Given ← chooseGiven(Us)
3:   Us ← Us \ {Given}
4:   New ← resolve(Given,Wo)
5:   Wo ← Wo ∪ {Given}
6:   if □ ∈ New then
7:     return FALSE
8:   end if
9:   reduce(Given, Wo, Us, New)
10:  Us ← Us ∪ {clause ∈ New | localID ∈ ca(clause)}
11:  for clause in New do
12:    for reasonerID in ca(clause) do
13:      if reasonerID ≠ localID then
14:        sendTo(clause,reasonerID)
15:      end if
16:    end for
17:  end for
18:  if Us == ∅ then
19:    Us ← receive()
20:  end if
21:  if receivedShutdownSignal then
22:    return TRUE
23:  end if
24: end while
```

Algorithm 2 controls the local application of resolution rules on a set of input clauses. The algorithm is an adapted version of the resolution prover specified in [63]. For recording which clauses have already been resolved with each other, the clause set is split into two sets, the *usable (Us)* set of clauses that have to be resolved and the *worked off (Wo)* set. Clauses in the *Wo* set are saturated, i.e. every clause that could be derived from *Wo* is either already contained in *Wo* or contained in *Us* or redundant. The algorithm is started with isSatisfiable($KB, \emptyset$, *localID*) to test satisfiability of the knowledge base KB. If a part $Wo \subset KB$ of the knowledge base is already known to be consistent, the algorithm could be started with isSatisfiable($KB \setminus Wo$, Wo, *localID*) to speed up the procedure. Until the whole set of clause is saturated or an empty clause is found, the algorithm repeatedly picks a clause (the *Given* clause) from the *Us* set and moves it to the *Wo* set (lines 2 to 5). Resolution rules are applied to the Given clause and each clause from the *Wo* set that can be resolved with Given (e.g. contains a literal that can be unified with a negated literal of the given clause) to obtain new clauses. Reduction is performed in line 9, the "reduce"-function includes deletion of redundant Given clauses (forward reduction) and redundant *Wo* clauses (backward reduction). Without distribution, lines 11-17 would be replaced by $Us \leftarrow Us \cup New$. But, instead of directly adding the new clauses to the local *Us* set, some of the new clauses may be added to the *Us* sets of other reasoners and new clauses received from other reasoners may be added to the local *Us* set. Note that the algorithm contains an endless loop (line 1). If no contradiction is found and the local *Us* set is empty, the algorithm gets stuck in line 19 until the reasoner is shut down or new clauses are received.
The extension to description logic consists in replacing the 'resolve' and 'reduce' function and the clause allocation *ca*.

4.4 Distributed Calculus

On the logical level, distributing resolution is a modification of the applied calculus. The physically separated clause sets are reflected in the modified calculus by restricting resolution to premises that are allocated to the same reasoner.

Definition 18 (Distributed Resolution Calculus).
A distributed resolution calculus $R(ca)$ is a resolution calculus R augmented with an allocation relation $ca \in (\mathcal{C} \times M)$, where M is the set of available reasoners. Each rule $r \in R(ca)$ is restricted to premises $P \subset \mathcal{C}$ with

$$\exists m \in M : \forall c \in P : ca(c, m)$$

This additional restriction is called allocation restriction.

A distributed calculus can be obtained from any resolution calculus by defining an allocation relation and adding the allocation restriction to each rule of the calculus. Logically, the difference between a resolution calculus and a distributed resolution calculus is similar to the difference between standard resolution and ordered resolution. A relation is added, and the rules are restricted to premises that comply to additional requirements based on that relation.
For guaranteeing completeness of a distributed calculus, we have to define an appropriate complete clause allocation *ca*.
It is easy to define an allocation that is complete for any allocation of inferences: If each clause is allocated to every reasoner, no inference is skipped compared to the original calculus. But obviously, this method is not efficient. The challenge in distributed resolution is to allocate a clause to as few reasoners as possible and still guarantee complete reasoning.

4.5 Soundness, Completeness, Termination

Distributing a calculus only restricts the applicability of the rules by the allocation restriction. Consequently, no inference is added and the possible inferences are equivalent to inferences of the original calculus. Hence, soundness is preserved by distribution.

Corollary 1 (distributed resolution soundness)**.** *If a resolution calculus R is sound, the distributed resolution calculus $R(ca)$ is sound for every clause allocation ca.*

Assuming duplicate literals are deleted in each clause, propositional resolution terminates for any finite input because the set of clauses that can be constructed from a finite set of predicates is finite. Since any clause is only derived once, the number of possible inferences is finite. Like soundness, termination is preserved by the distributed calculus, because no new clauses are added. In distributed resolution, the set of clauses that may be created is still finite. Inferences that derive redundant clauses may be added e.g. because a duplicate clause exists at another reasoner. But, the number of duplicates is limited by the number of reasoners. Hence, termination is still preserved.

Corollary 2 (distributed resolution termination)**.** *If exhaustive application of a resolution calculus R terminates, exhaustive application of the distributed resolution calculus $R(ca)$ terminates for every clause allocation ca.*

In contrast to soundness and termination, the completeness of a distributed resolution calculus depends on the applied allocation of clauses. The allocation of inferences is defined by the allocation of clauses, we only have to make sure the clause allocation is complete.

Theorem 1 (distributed resolution completeness)**.** *If a resolution calculus R is complete and the clause allocation ca is complete for R, then the distributed resolution calculus $R(ca)$ is complete.*

Proof. Assume in contrary, there is an unsatisfiable set of clauses C and saturating C by $R(ca)$ does not derive an empty clause. Since R is complete and terminates, saturating C by R results in a set that contains an empty clause. Hence, there is at least one inference that is applicable in R but not applicable in $R(ca)$. We consider the first inference of R that is not applicable in $R(ca)$, i.e. an inference with premises p_1, p_2 with

- $C \vdash_R p_i$ and $C \vdash_{R(ca)} p_i$ for $i = 1, 2$
- $r(p_1, p_2) = c$ for some rule $r \in R$ but
- $C \nvdash_{R(ca)} c$

The maximal literals of the premises p_1 and p_2 contain the same predicate p. Hence, both premises are allocated to the reasoner responsible for p. But, the rule r with allocation restriction $\exists m \in M : ac(p_1, m), ac(p_2, m)$ is then also applicable to premises p_1, p_2 and derives the conclusion c. Consequently $C \vdash_{R(ca)} c$ holds. □

In other words, the allocation restriction is always true in $R(ca)$, when the other restrictions are true. Hence, distribution does not restrict the applicability of ordered resolution rules.

From the algorithmic point of view, Theorem 1 implies that if the resolve function implements a complete calculus and reduction deletes only redundant clauses, then completeness is preserved by distribution. Every reasoner creates a locally saturated set of clauses. According to the proof of Theorem 1, the union of the locally saturated clause sets is also a saturated set of clauses, i.e. the resolve function would not derive any new clause from the union.

Note that the requirement on reduction is not trivial. There are reduction rules that are not applicable in a distributed setting. For example, if a certain propositional variable p occurs only positive and never in a negative literal, all clauses that contain the variable are redundant. But, this reduction rule makes a closed world assumption, it assumes all input clauses are analysed. In distributed resolution, only the local clause set is taken into account for reduction. E.g., $\neg p$ could be contained in a clause that is allocated to another reasoner. Consequently, we can only apply reduction rules that comply with the open world assumption and do not require complete information about all clauses.

The distributed propositional ordered resolution calculus is not only sound, complete and terminates, also the distribution is compact. Since the allocation of clauses is functional, each clause is allocated to exactly one reasoner.

Duplicate clauses may occur when they are derived from different premises. But, duplicates are always allocated to the same reasoner that takes care of eliminating all duplicates.

Chapter 5

Distributed Resolution for First Order Logic

As we have seen above, the ability to define a sound and complete distributed reasoning method relies on two requirements: (1) the existence of a sound and complete resolution calculus and (2) the ability to find a corresponding allocation that satisfies the allocation restriction. In this section, we show that there is a calculus and allocation for full first order logic that satisfy both of these requirements leading to a sound and complete distributed resolution method. For the case of ontologies defined in the description logic $\mathcal{ALCHI}$ – a decidable subset of first order logic – the allocation relation is functional, it allocates each clause to only one reasoner and duplication of clauses and inferences is avoided.

The basic theory of this chapter was first published in [45] and [47] with results from a simulated distributed resolution. A more comprehensive description of the approach and experiments on query performance are published in [49].

We do not address reduction rules in this section because reduction is not necessary to guarantee the theoretical properties of the proposed calculus. However, for efficient reasoning, reduction is essential and hence the practical effects distribution has on reduction are discussed in the experimental section.

While in propositional logic, we have only nullary predicates (i.e. propositional variables) and no functions, clauses obtained from description logic ontologies contain unary and binary predicates corresponding to concepts and properties of the ontology. Furthermore, there are nullary functions (constants) corresponding to instances of the ontology and unary functions introduced by skolemization of existential quantifiers. Without properties and instances, an ontology is a concept hierarchy that can be represented in propositional logic. The main difference between description logic and propositional logic are the properties. $\mathcal{ALC}$ allows using property restric-

tions in concept expressions, i.e. expressions $\exists R.C$ (all instances are related via R to an instance of C) and $\forall R.C$ (via property R, all instances are only related to instances of C). This chapter is focused on the description logic $\mathcal{ALCHI}$ that additionally allows expressing property hierarchy ($\mathcal{H}$) and inverse properties ($\mathcal{I}$).

5.1 Calculus

It has been shown that standard resolution methods can be adapted to provide sound and complete reasoning for $\mathcal{ALC}$ [58, 36]. These methods that provide the basis for our work rely on ordered resolution, which depends on two parameters: An *order* of literals and a *selection function* that maps every clause to a subset of its negative literals. These parameters are used to restrict the applicability of inference rules, thus reducing the number of derived clauses and avoiding redundant inferences. The selection function is an ordering refinement. While the ordering is defined globally for all literals, a literal may be selected in one clause and not selected in another clause. Usually, selection is used to extend the global order of literals for clauses that have more than one maximal literal. The ordered resolution calculus consists of the two inference rules ordered resolution and factoring described in Definition 19. The letters C and D refer to clauses, A and B are literals.

Definition 19 (Ordered Resolution).

$$\textit{Ordered resolution} \quad \frac{C \vee A \quad D \vee \neg B}{C\sigma \vee D\sigma}$$

where

1. *σ is the most general unifier of A and B*
2. *either $\neg B$ is selected in $D \vee \neg B$ or else nothing is selected in $D \vee \neg B$ and $B\sigma$ is maximal w.r.t. $D\sigma$*
3. *$A\sigma$ is strictly maximal with respect to $C\sigma$*
4. *nothing is selected in $C\sigma \vee A\sigma$*

$$\textit{Positive factoring} \quad \frac{C \vee A \vee B}{C\sigma \vee A\sigma}$$

where

1. *σ is the most general unifier of A and B*
2. *$A\sigma$ is maximal with respect to $C\sigma \vee B\sigma$*
3. *nothing is selected in $C\sigma \vee A\sigma \vee B\sigma$*

This calculus is well known to be sound and complete for first order clauses. However, for guaranteeing termination on $\mathcal{ALCHI}$ a special parameterization is necessary. In particular, [58] showed that using ordered resolution, decidability on $\mathcal{ALC}$ can be preserved by transforming the ontology into definitorial form (see Definition 2) before clausification and applying ordered resolution with appropriate selection and ordering. Extending the proof to $\mathcal{ALCHI}$ is straight forward.

Definition 20 ($\mathcal{ALCHI}$ Resolution).
$R_{\mathcal{A}}$ is the calculus

- *consisting of the inference rules ordered resolution and factoring,*
- *with selection of exactly the negative binary literals, and*
- *literal ordering $\succ$ with $R(x, f(x)) \succ \neg C(x)$ and $D(f(x)) \succ \neg C(x)$, for all function symbols f, and predicates R, C, and D.*

The ordering requirement is satisfied by every lexicographic path ordering (LPO, Definition 5) based on a total precedence $>$ on function, predicate and logical symbols with $f > P > \neg$ for every function symbol f and predicate symbol P.
For the simple types of literals occurring in clauses translated from $\mathcal{ALC}$ axioms (Table 5.1 described below), a LPO ordering with precedence as required by Definition 20 implies a simple set of ordering rules. Literals that contain different variables are incomparable, for literals that share the same variables,

1. Literals containing a function symbol precede literals that do not contain a function symbol.
2. Literals containing a function symbol are ordered according to the precedence of the function symbols.
3. Literals that do not contain a function symbol are ordered according to the precedence of the predicate symbols.

In theory, any precedence that complies with the requirements of Definition 20 can be used. The standard method for defining a precedence is based on the number of occurrence of each symbol in the input because a maximal literal that contains rare symbols is probably resolved with less partner clauses than a maximal literal that contains symbols with high frequency. Therefore, symbols are usually ordered according to the number of occurrences, rare symbols precede symbols that occur more frequently.
$R_{\mathcal{A}}$ is sound and complete, because ordered resolution is sound and complete for first order logic for any admissible ordering and any selection of negative literals. For termination, the proof for termination of $R_{\mathcal{A}}$ on clauses from $\mathcal{ALC}$ [58] is extended straight forwardly to $\mathcal{ALCHI}$.

type #	$\mathcal{ALCHI}$ clause type	resolvable literal type
1	$R(x, f(x)) \vee \mathbf{P}(x)$	$R(x, f(x))$
2a	$\mathbf{P}(x)$	$(\neg)P(x)$
2b	$\mathbf{P_1}(\mathbf{f}(x)) \vee \mathbf{P_2}(x)$	$(\neg)P(f(x))$
3	$\neg R(x, y) \vee \mathbf{P_1}(x) \vee \mathbf{P_2}(y)$	$\neg R(x, y)$
4	$\mathbf{P}(\mathbf{a})$	$(\neg)P(a)$
5	$(\neg)R(a, b)$	$(\neg)R(a, b)$
6	$\neg R(x, y) \vee S(x, y)$	$\neg R(x, y)$
7	$\neg R(x, y) \vee S(y, x)$	$\neg R(x, y)$
8	$R(f(x), x) \vee \mathbf{P}(x)$	$R(f(x), x)$

Table 5.1: Clause types resulting from the translation of an $\mathcal{ALCHI}$ ontology to first order clauses. $\mathbf{P}(t)$, where t is a term, denotes a possibly empty disjunction of the form $(\neg)P_1(t) \vee \cdots \vee (\neg)P_n(t)$. $\mathbf{P}(\mathbf{f}(x))$ denotes a disjunction of the form $\mathbf{P_1}(f_1(x)) \vee \cdots \vee \mathbf{P_m}(f_m(x))$. Each $\mathbf{P_i}(f_i(x))$ may contain positive and negative literals.

Theorem 2 ($\mathbf{R}_\mathcal{A}$ decides $\mathcal{ALCHI}$)**.** *For an $\mathcal{ALCHI}$ knowledge base KB, saturating the clauses obtained from the definitorial form of KB by $R_\mathcal{A}$ decides satisfiability of KB.*

Proof. The set of $\mathcal{ALC}$ clauses (i.e. clauses obtained from translation of an $\mathcal{ALC}$ ontology) is closed under $R_\mathcal{A}$ [58]. Also, the set of $\mathcal{ALCHI}$ clauses depicted in Table 5.1 is closed under $R_\mathcal{A}$.
When an $\mathcal{ALCHI}$ ontology in definitorial form is translated to first order clauses as described in Section 2.4, every clause is of one of the types listed in Table 5.1. Further, applying ordered resolution to clauses of the listed types results in a listed clause type. [58] proved that the clause types 1-5 are closed under $R_\mathcal{A}$. The additional inferences with clauses of type 6-8 are easy to check. The resolvable literals of all additional clause types are binary predicate literals, Table 5.2 shows the possible types of premises and conclusions. For example, the first line of the table states resolving a clause of type 1 with a clause of type 3 results in a conclusion of type 2b. The type of the conclusion does not matter, the point is that the conclusion is of one of the listed types. Essential is the limited nesting depth of functions. For $\mathcal{ALCHI}$, functions are not nested at all, the argument is always a variable or constant. Consequently, the number of literals that can be built using a finite number of symbols is finite. Hence, if duplicated literals are deleted, the saturation terminates because otherwise it would create a infinite number of clauses from a finite number of literals. □

Compared to $\mathcal{ALC}$, the clause types 6 and 7 are added for property hierarchy and inverse properties. Type 8 results from resolving a type 7 clause with a clause of type 1. We adapted the notation of the $\mathcal{ALC}$ clauses from [36] and extended it to $\mathcal{ALCHI}$. Additionally, the resolvable literal (discussed

premise 1	premise 2	conclusion
1	3	2b
1	6	1
1	7	8
2a	2a	2a
2a	2b	2a/2b
2a	4	4
2b	2b	2a/2b
3	8	2b
5	6	5
5	7	5
6	8	8
7	8	1

Table 5.2: Types of premises and conclusion of $R_{\mathcal{A}}$ inferences on $\mathcal{ALCHI}$ clauses.

below) is depicted in Table 5.1.

In this section, we described a sound and complete resolution calculus for first order logic that terminates for clauses obtained from an $\mathcal{ALCHI}$ ontology. For turning it into a complete distributed resolution method, we have to find a corresponding allocation function that preserves completeness of the calculus.

5.2 Distribution

The idea for defining an allocation function is to restrict the inference options such that there is a unique literal for every clause that may be resolved in a subsequent inference step. Then, we can use this unique literal as a basis for deciding the allocation of the clause and guarantee that clauses that can be the premises of a resolution inference are always sent to the same reasoner. Hence, every inference that occurs in the centralized resolution reasoner is also possible in distributed resolution. The idea is stated precisely in the following definitions and results.

Definition 21 (Resolvable Literal).
A literal L of a clause c is a resolvable literal in c iff

1. *L is a selected literal or*
2. *no literal is selected in c and L is strictly maximal in c.*

The resolvable literals of a clause c are denoted by $resolvableLiteral(c)$.

Again, we may omit parentheses, if $resolvableLiteral(c)$ is a unit set.

Corollary 3 (Resolvable Literal)**.** *For every ordered resolution inference in $R_{\mathcal{A}}$ the literals A and B in Definition 19 are resolvable literals in $(C \vee A)$ and $(D \vee \neg B)$ respectively.*

The essential precondition for our distribution strategy is that for a given clause, the literal that will be resolved in the next resolution inference does not depend on the available other premise candidates, it can be determined without considering other clauses.

Corollary 4 (Unique resolvable literal)**.** *Every $\mathcal{ALCHI}$ clause contains exactly one resolvable literal.*

Corollary 4 holds because $\mathcal{ALCHI}$ clauses contain at most one selected literal and the ordering is total on ground literals and literals containing the same variables. Thus, only clauses that contain more than one variable may have multiple maximal literals, but then they are of type 3, 6 or 7 and contain a selected literal.
For full first order logic, this property does not hold, here an arbitrary number of resolvable literals is possible theoretically. However, an increased number of resolvable literals usually decreases the efficiency of a calculus also without distribution. As explained in Chapter 4, restrictions are added to a calculus to reduce the number of inference options. Hence, for efficient resolution calculi and decidable subsets of first order logic, the number of resolvable literals is usually limited.
Relying on the resolvable literals, we can now define a suitable allocation for FOL that is functional for $\mathcal{ALCHI}$ clauses. Like in propositional resolution, the allocation is based on an allocation of the signature. In particular, an allocation of the predicates is required. Then, ordered resolution inferences are allocated to the reasoner responsible for the top predicate of the unified literals.

Definition 22 (a-symbol for FOL).
The set of allocation symbols of a first order clause c is the set of all predicates contained in a positive or negative resolvable literal of c:

$$\text{a-symbol}_{\mathcal{A}}(c) = \{P \mid (\neg)P(t_1, ..., t_n) \in resolvableLiteral(c)\}$$

where P is a predicate symbol and $t_1, ..., t_n$ are terms.

For $\mathcal{ALCHI}$ clauses c, $\text{a-symbol}_{\mathcal{A}}(c) = \{P\}$ contains only a single symbol because c has only one resolvable literal (Corollary 4). For unit sets we omit brackets and write $\text{a-symbol}_{\mathcal{A}}(c) = P$. The clause allocation is defined based on the allocation of predicates.

Definition 23 (Allocation for FOL).
The allocation of a clause c for the distributed calculus $R_{\mathcal{A}}(ca_{\mathcal{A}})$ is

$$ca_{\mathcal{A}}(c) := sa(\text{a-symbol}_{\mathcal{A}}(c))$$

Obviously, the clause allocation is functional for $\mathcal{ALCHI}$ clauses because they have only one a-symbol.
Note that for interlinked ontologies, $\mathcal{O}$ is the union of all ontologies and the symbol allocation $sa(X)$ can be defined by the namespace of the concept or property name X. For a single ontology, every partitioning of the ontology terms induces an allocation of symbols via randomly allocating parts to reasoners. Methods for computing an optimized allocation of symbols are discussed in Chapter 8.
For determining where (and if) a derived clause is sent, we first pick the resolvable literals of the clause, then the predicates of these literals and finally the reasoners these predicates are allocated to. For $\mathcal{ALCHI}$ clauses, the destination is always a single reasoner. The allocation is complete for both $\mathcal{ALCHI}$ and full FOL.

Theorem 3. *$ca_{\mathcal{A}}$ is a complete allocation for $R_{\mathcal{A}}$.*

Proof. We have to show that the premises of each inference are allocated to the same reasoner. Again, it is sufficient to consider inferences with more than one premise, i.e. the ordered resolution rule. For each inference $(orderedResolution, p_1, p_2)$, the literal A in Definition 19 is strictly maximal, i.e. resolvable literal of the clause $C \vee A$. Similarly, the literal $\neg B$ is resolvable literal of $D \vee \neg B$. Since there is a unifier σ of A and B, the two literals necessarily have the same top symbol. I.e., both A and B are of the form $P(..)$ with the same unary or binary predicate symbol P. Hence, P is the allocation symbol of both clauses: a-symbol$(C \vee A)$ = a-symbol$(D \vee \neg B)$. Consequently both clauses are allocated to the same reasoner $sa(P)$ that is responsible for P. □

Since $ca_{\mathcal{A}}$ is a complete allocation, we can decide satisfiability of $\mathcal{ALCHI}$ ontologies using the distributed resolution in Algorithm 2 with calculus $R_{\mathcal{A}}$ and allocation function $ca_{\mathcal{A}}$.

Theorem 4. *The distributed resolution calculus $R_{\mathcal{A}}(ca_{\mathcal{A}})$ decides $\mathcal{ALCHI}$ satisfiability.*

Proof. We have to show that $R_{\mathcal{A}}(ca_{\mathcal{A}})$ is sound, complete and terminates. Since $R_{\mathcal{A}}$ is sound and terminates for $\mathcal{ALCHI}$, Corollary 1 and 2 apply, hence $R_{\mathcal{A}}(ca_{\mathcal{A}})$ is sound and terminates for $\mathcal{ALCHI}$. $R_{\mathcal{A}}$ is complete and $ca_{\mathcal{A}}$ is a complete allocation for $R_{\mathcal{A}}$. Consequently, according to Theorem 3 and Theorem 1, $R_{\mathcal{A}}(ca_{\mathcal{A}})$ is complete for deciding $\mathcal{ALCHI}$ satisfiability. □

Ontology A	Ontology B
$A{:}Set \sqsubseteq \exists A{:}part.A{:}Set$	$B{:}Tuple \sqsubseteq B{:}Set$
$A{:}Tuple \sqsubseteq \forall A{:}part.\neg A{:}Set$	$B{:}Pair \sqsubseteq B{:}Tuple$
	$B{:}Pair(a)$

Mapping

$$A{:}Tuple \doteq B{:}Tuple$$
$$A{:}Set \doteq B{:}Set$$

Figure 5.1: Description Logic example of two ontologies and a mapping. The ontology axioms describe the concepts Set, Tuple and Pair. Mapping axioms describe equivalences between elements of different ontologies.

Figure 5.2 illustrates distributed resolution on the example ontologies from Figure 5.1. Here the setting is an ontology network consisting of two ontologies and a set of link axioms that connect the ontogies. The axioms are translated to clauses specified by lists of literals, all variables are implicitly universally quantified. The existential quantification is translated to a skolem function. Inferred clauses are depicted below the dashed line, the superscript numbers refer to the origin of a clause, i.e. the premises it is derived from or the ontology that sent it. Arrows indicate propagation of a clause to the other ontology. The precedence of symbols is $B{:}Set > B{:}Tuple > B{:}Pair > A{:}Set > A{:}Tuple > A{:}part$. All literals with top symbol $A{:}part$ are selected. To help tracking the inferences, resolvable literals are set in black, other literals in gray. Hence, a derived clause consists of the gray literals of its premises. The first inference of ontology A is clause 4, it is derived from clauses 1 and 3. The mapping axioms are added to ontology B. Here the first inference is clause 8. Clause 10 is derived from 2 and 5 and then propagated to ontology A. From another propagated clause and the local clauses, ontology A finally derives the empty clause and hence detects the inconsistency in the ontology network.

After proving theoretical properties of distributed resolution and explaining the distributed process on an example, we now turn to the implementation of the algorithm and experiments.

5.3 Implementation

The distributed resolution implementation used in the experiments is based on the first order prover SPASS[4] [64] developed at the Max-Planck-Institut

[4] `http://www.spass-prover.org`

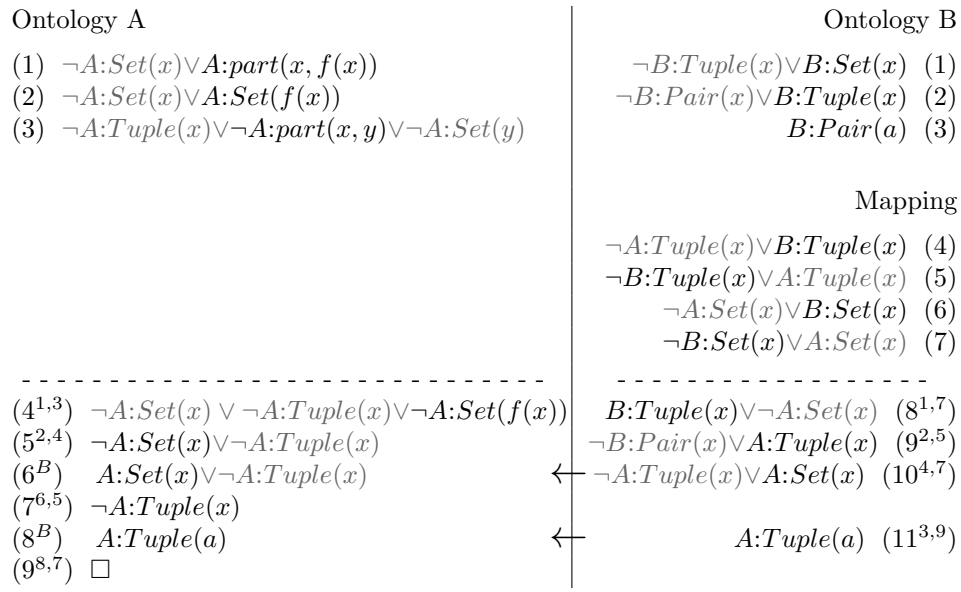

Figure 5.2: Distributed ordered resolution example on the ontologies from Figure 5.1.

Informatik. In particular, we use the YAGO version of the SPASS reasoner that is optimized for large input. In contrast to the actor model implementation used in [39], SPASS is written in C and benefits from the more flexible memory management. Furthermore, SPASS is designed with a focus on extendability and readability of the code and constitutes a good basis for implementing distributed resolution following the message passing paradigm. SPASS supports a number of different resolution strategies including ordered resolution and basic superposition. The configuration allows specifying the precedence, selection and ordering we need for $\mathcal{ALCHI}$ clauses. We implemented definitorial form normalization and clausification in a separate tool. Clauses are stored in separate files for each ontology and include precedence and selection in every input file. The precedence complies with the requirements of $R_{\mathcal{A}}$ (Definition 20). Additionally, for some experiments the function and predicate symbols are ordered according to the number of occurrences in the input clauses, with rare symbols first. The applied reduction rules include forward and backward subsumption reduction[5]. Note that we can use any reduction rule that does not impose a closed world assumption. For example, a clause c that is subsumed by another clause s is redundant independently of all other clauses.

[5]The complete configuration for Spass is: Distributed=1 Auto=0 Splits=0 Ordering=1 Sorts=0 Select=3 FullRed=1 IORe=1 IOFc=1 IEmS=0 ISoR=0 IOHy=0 RFSub=1 RBSub=1 RInput=0 RSSi=0 RObv=1 RCon=1 RTaut=1 RUnC=1 RSST=0 RBMRR=1 RFMRR=1

For turning SPASS into a distributed reasoner (i.e. adding the "Distributed" option) we added support for sending and receiving clauses and for controlling the distributed process.

Sending clauses is the easy part, the provided printing methods were modified to print to a string instead of a file and the string is then send to another reasoner via TCP. For receiving new clauses the input parser was modified to read clause lists from a string that are subsequently added to the local clause store. We changed the format of clauses to print symbol numbers instead of symbol names to speed up parsing.

Sending is implemented as an additional reduction method. If a derived clause is still considered non-redundant after all other reductions are applied, allocation of the clause is computed. If the clause is not allocated to the reasoner that derived it, it is marked as redundant and send to the reasoner responsible for the clause. A set of received clauses is treated like a set derived from a given clause, i.e. it is forward and backward reduced with respect to the local worked off clause list before adding the non redundant received clauses to the usable list. Hence, forward and backward reduction are performed prior to sending and after receiving clauses for deleting as many redundant clauses as possible. Clauses that have to be propagated are send in every loop, but new clauses are received from other reasoners only when the local clause set is completely saturated.

The allocation consists of two components, an allocation table that maps symbols to reasoner IDs and a routing table that stores the corresponding addresses for each reasoner ID. At startup, every reasoner reads the routing table. For the allocation table we implemented different options. One option is to specify the whole allocation table (symbol name - reasonerID) in a file that is then read along with the routing table. For this variant, the allocation table is created in advance by other tools as explained in Chapter 8. Other options include allocation based on namespaces that are specified as prefixes in the symbol names. The simplest type of allocation is computed from the symbol index (i.e. the position of the symbol in the precedence): Divide by the number of reasoners and allocate to the reasoner specified by the remainder of the division.

Startup and shutdown of the system is initialized by a central control process. In a fully decentralized P2P system this job is performed by the peer that receives a query. The control process starts the separate machines on their respective input clauses files. Apart from passing clauses between each other, the reasoners send status messages whenever they are locally saturated, when they continue reasoning on newly received clauses and when they derive an empty clause. When one reasoner finds a proof or all reasoners are saturated for an interval longer than the maximal time necessary for clause propagation, the query is answered and the reasoners are shut down.

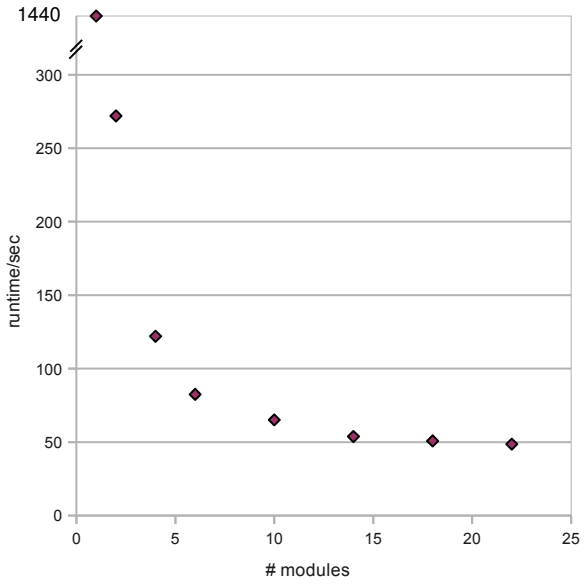

Figure 5.3: Runtimes for saturation of the FMA ontology, using different numbers of reasoners.

5.4 Experiments

Experiments were executed on the Esslingen cluster of the bwGrid[6] using one core for each reasoner. Each compute node has 8 cores (two 4-core Intel Nehalem CPUs with 2.27GHz/2.8GHz) and 24GB memory. We did not take advantage of shared memory, communication between reasoners on the same node was the same as between reasoners on different nodes.

5.4.1 FMA

For the first set of tests, we used the Foundational Model of Anatomy (FMA light) ontology, one of the largest ontologies from the Open Biological and Biomedical Ontologies (OBO) that is available in OWL. We removed annotations as they are not relevant for reasoning. The expressivity of the tested ontology is $\mathcal{ALEH}$, translation to first order logic resulted in 164445 clauses. Apart from the requirements of Definition 20 the precedence was random. The allocation of symbols is computed using the METIS graph partitioning tool. Details of the allocation method are explained in Chapter 8.
Saturation of the ontology was performed using different numbers of reason-

[6]bwGRiD (`http://www.bw-grid.de`), member of the German D-Grid initiative, funded by the Ministry for Education and Research (Bundesministerium für Bildung und Forschung) and the Ministry for Science, Research and Arts Baden-Württemberg (Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg).

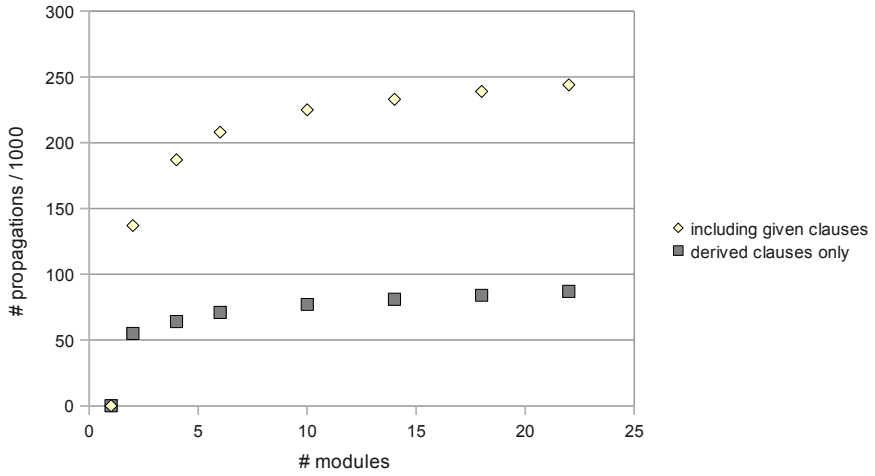

Figure 5.4: Number of propagated clauses for saturation of the FMA ontology, using different numbers of reasoners. Denoted are the total number of propagations and propagations of derived clauses

ers, the resulting runtimes are depicted in Figure 5.3. The used compute nodes are equipped with 2.27GHz CPUs. The saturation without distribution on a single reasoner took more than 1400 seconds. Using two reasoners for the same task reduced the runtime to about 300 seconds. From the duplication of computation power we would expect a decrease by at most 50% to 700 seconds. The decrease in runtime that exceeds this number can have different reasons. Probably, preprocessing like input reduction is not linear in the input size and contributes to the observation. Another reason could be the reduced size of the *Wo* clause lists. For the single reasoner, every *Wo* clause is considered as partner clause for a given clause. In the distributed setting only the local *Wo* clauses are considered resulting in fewer unification attempts. Additionally, the amount of available CPU cache increases with the number of reasoners. Hence, a single reasoner has to access the slower memory more often.

For up to 10 reasoners, adding more reasoners to the problem reduced runtime considerably. But, then the runtime converges to about 50 seconds. If the total runtime sum of all reasoners would be constant, the system runtime would decrease by 23% from 14 reasoners to 18 reasoners. But, the decrease is only 6%. Possible reason for this deviation from the reference value is overhead caused by distribution or a sequencial component of the computation that cannot be computed in parallel.

The number of derived clauses did not increase considerably with adding more reasoners. Hence, the restricted application of reduction rules was no problem for this ontology. Figure 5.4 shows the increase of the number of

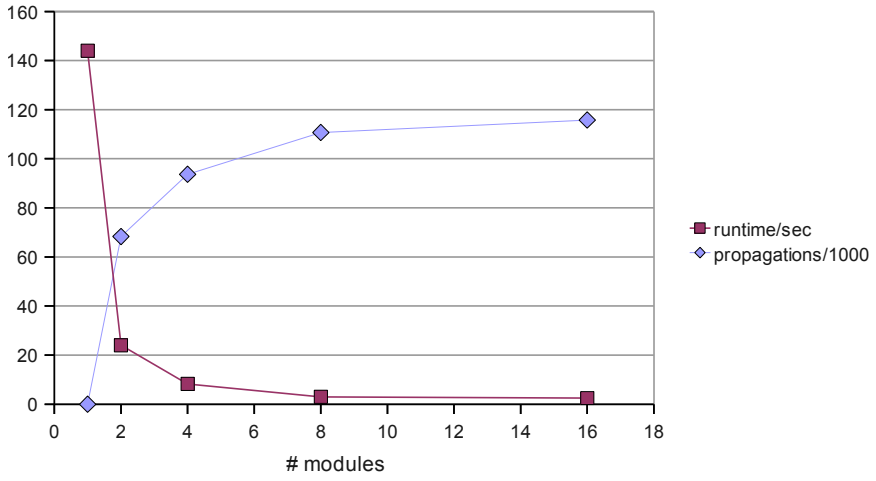

Figure 5.5: Runtime and propagation for saturation of the NCI ontology depending on the number of reasoners.

propagated clauses, when more reasoners are applied. The upper values are the total number of propagations, lower values show only propagations of derived clauses. The input clauses are distributed randomly, hence there are some input clauses that are propagated when they are picked as given clauses. The number of propagations increases, but the average number of propagations per reasoner decreases. The moderate increase in derivations and propagations indicates that the reason for the runtime convergence is not caused by propagation or restricted reduction.

5.4.2 NCI

Another set of tests with a similar setting was executed for the NCI ontology, an ontology developed by the National Cancer Institute[7] of the U.S. Department of Health and Human Services. Compared to the FMA ontology, the NCI is smaller, it contains 27625 concepts and 70 object properties, additional 100 concepts where introduced by normalization. The expressivity of the ontology is $\mathcal{ALC}$, the first order representation consists of 60971 clauses. For the precedence, predicate symbols are ordered according to the frequency. The allocation is close to random, it is computed from the predicate index numbers. The applied nodes have CPUs with 2.8GHz.
Runtimes for the saturation are shown in Figure 5.5 depending on the number of applied reasoners. Increasing the number of reasoners from one to two and four again decreases the runtime much more than expected. One reasoner takes 144 seconds for the saturation while two reasoners need only

[7] `www.cancer.gov`

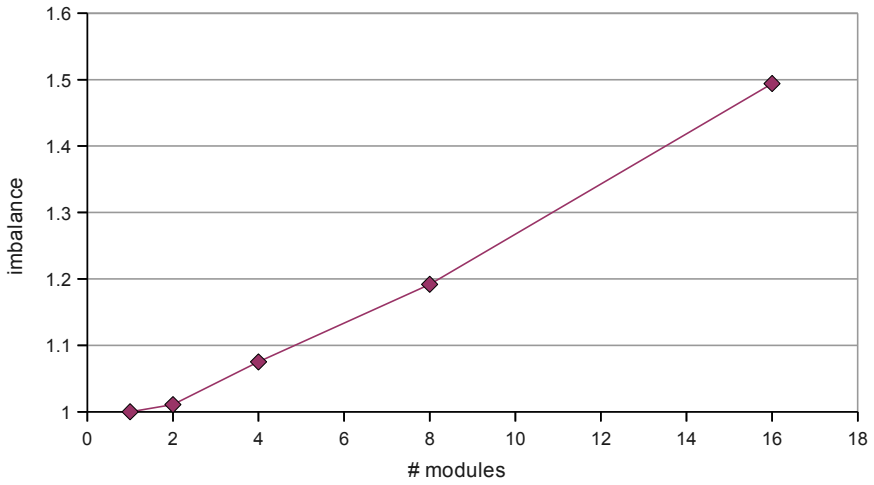

Figure 5.6: Balance of NCI saturation.

24 seconds and with four reasoners it takes less than 9 seconds. The number of derivations was 123439 for all tests. The graphs for runtime and propagation look very similar to those of the FMA, only the values are smaller and the stagnation of runtime is observed for a smaller number of reasoners. With 8 reasoners, the runtime is decreased to about 3 seconds which is close to the best runtime we achieved. For the FMA, where the saturation requires about 10 times more runtime, we observe decreasing runtimes when using up to 14 reasoners.
For FMA, the number of propagations per second for two reasoners was only 500 although this number includes propagated input clauses. For NCI an average of 2800 propagations per second was observed. Despite the differences in propagation, the runtime decrease for two and four reasoners is similar for the two ontologies. This indicates that not distribution overhead but sequencial parts of the computation limit the distributability.
Figure 5.6 shows the saturation of NCI was well balanced for up to 8 reasoners. For 8 reasoners, one reasoner performed 20% more inferences than the average. Imbalance increased linearly with the number of reasoners.

To sum up, the decrease in runtime achieved by distributed resolution exceeded our expectations. For some settings, a duplication of the number of reasoners resulted in a runtime decrease by more than 50%.

Chapter 6

Distributed Resolution with Transitive Properties

An important type of axiom that is not covered by $\mathcal{ALCHI}$ are axioms that turn a property into a transitive property. The most prominent example is the part-of property used in many ontologies. Obviously, part-of should be declared as transitive property. Otherwise, it would not be possible to derive for example "part-of(finger,arm)" from "part-of(finger,hand)" and "part-of(hand,arm)".
Unfortunately, with the standard translation of transitivity axioms to clauses (Section 2.4), the calculus $R_{\mathcal{A}}$ is not guaranteed to terminate. One option is to replace transitivity axioms by simple subsumptions using a well known transformation, thereby reducing the expressivity of a $\mathcal{SHI}$ ontology to $\mathcal{ALCHI}$. The transformation is polynomial in the size of the input, but the adaption to the distributed setting is problematic. In contrast to the transformation of $\mathcal{ALCHI}$ axioms to first order clauses, the advanced translation of transitivity depends on the whole ontology and not only on the transitivity axioms. Hence, a new axiom that is added to a $\mathcal{SHI}$ ontology can not be translated to clauses independently.
However, there is an approach proposed by [8], that replaces transitivity axioms not by simpler axioms but by additional resolution rules called *ordered chaining*. For the saturation of the very large YAGO ontology reported in [57], application of the chaining rules caused the essential speedup that enabled the saturation. In this chapter, we describe how the chaining calculus used by [57] can be distributed for reducing the runtime of the saturation. The addressed expressivity is the *Bernays-Schönfinkel Horn class with equality* where all clauses are range restricted.

Definition 24 (BSHE).
The Bernays-Schönfinkel Horn class with equality (BSHE) is a subset of first order clauses where clauses are of the form $C \vee A$ or C with:

i) C contains only negative literals and A is a positive literal.

ii) Every variable contained in A is also contained in a non-equality atom of C.

iii) C and A contain no function symbols, only constants.

iv) Equality is present among the predicate symbols.

It is assumed, that all constants are different (unique name assumption), i.e. literals $a \not\approx b$ are always true and $a \approx b$ always false for all different constants a and b. Consequently clauses can be simplified to contain no ground equality literals by deleting false literals and deleting clauses that contain true literals. Furthermore, clauses $C \vee x \not\approx t$ with variable x and constant or variable t can be simplified to $C[x \leftarrow t]$ (replacing all occurrences of x by t). Hence, all remaining equality literals are positive and non-ground. Unit clauses cannot be equalities because (ii) implies all positive unit clauses are ground.
Compared to the expressivity of $\mathcal{ALC}$, BSHE does not support existential restrictions on properties ($\exists R.C$) because translation to first order logic would introduce a skolem function (see Table 2.1). Furthermore, axioms $A \sqsubseteq C \sqcup D$ are not expressible. But, it is possible to state that a property is functional or transitive.

6.1 Calculus

Since BSHE does not contain function symbols, we do not need superposition rules in the calculus. Hence, no translation of predicates to general functions is necessary. Motivated by the difficulties in deciding satisfiability of the YAGO ontology, [57] proposed a calculus for checking satisfiability of BSHE theories. For guaranteeing completeness also for transitive predicates, ordering requirements are necessary that are stronger than those used for ordered resolution and basic superposition.

Definition 25 (Admissible Ordering for $R_{\mathcal{B}}$).
An ordering $\succ$ on ground terms and literals is admissible for $R_{\mathcal{B}}$ if

- *it is well founded and total on ground terms and literals,*
- *$L \succ L'$ whenever literals L and L' contain the same transitive predicate Q, and the maximal subterm of L' is strictly smaller than the maximal subterm of L,*
- *$\neg A \succ A$ for all ground atoms A,*
- *$\neg Q(s,t) \succ Q(s',t')$ whenever Q is a transitive predicate and $max(s,t) \succeq max(s',t')$,*

- $\neg Q(s,s) \succ \neg B$ *whenever* Q *is a transitive predicate and* B *is* $Q(s,t)$ *or* $Q(t,s)$ *and* $s \succ t$.

These properties of the ordering ensure the calculus $R_{\mathcal{B}}$ is efficient also for transitive properties. We assume transitivity is not represented by clauses but by marking some properties as transitive. The chaining rules make sure the calculus is complete for the implications of transitivity, hyperresolution is for the other implications. The last rule is object equality cut, it implements the unique name assumption.

Definition 26.
The calculus $R_{\mathcal{B}}$ *consist of the rules Ordered Chaining, Negative Chaining and Hyperresolution where the ordering* $\succ$ *is admissible for* $R_{\mathcal{B}}$. *In the following rules,* Q *is a transitive predicate.*

$$\textit{Ordered Chaining} \quad \frac{Q(l,s) \qquad Q(t,r)}{Q(l,r)\sigma}$$

where

1. σ *is the most general unifier of* s *and* t
2. $l\sigma \not\succeq s\sigma$ *and* $r\sigma \not\succeq t\sigma$

$$\textit{Negative Chaining Right} \quad \frac{Q(l,s) \qquad D \vee \neg Q(t,r)}{D\sigma \vee \neg Q(s,r)\sigma}$$

where

1. σ *is the most general unifier of* l *and* t
2. $s\sigma \not\succeq l\sigma$ *and* $r\sigma \not\succeq t\sigma$

$$\textit{Negative Chaining Left} \quad \frac{Q(l,s) \qquad D \vee \neg Q(t,r)}{D\sigma \vee \neg Q(t,l)\sigma}$$

where

1. σ *is the most general unifier of* s *and* r
2. $l\sigma \not\succeq s\sigma$ *and* $t\sigma \not\succeq r\sigma$

$$\textit{Hyperresolution} \quad \frac{A_1 \quad ... \quad A_n \qquad \neg B_1 \vee ... \vee \neg B_n \vee P}{P\sigma}$$

where

1. $n \geq 1$, $A_1, ..., A_n$ *are unit clauses,*
2. P *is a positive literal or false*
3. σ *is the simultaneous most general unifier of* A_i *and* B_i *respectively, for all* $i \in \{1, ..., n\}$

$$OECut \quad \frac{a \approx b}{\square}$$

where a and b are two different constants.

Before we distribute this calculus, we address the theoretical properties that have to be preserved by distribution.

6.1.1 Soundness, Completeness, Termination

It is easily checked that all inferences in $R_{\mathcal{B}}$ are sound. It is well known, that hyperresolution alone is complete for first order theories, hence $R_{\mathcal{B}}$ is also complete for BSHE. But, with clauses for transitivity, $R_{\mathcal{B}}$ would not be efficient because all implied binary unit clauses are materialized. Therefore, transitivity clauses are deleted and the corresponding predicates are marked as transitive. The calculus terminates because no conclusion is longer than the main premise of an inference. Up to variable renaming, the number of different clauses of a given length is finite.
More difficult to prove is the completeness of the calculus. For theories without transitive predicates, hyperresolution is complete. Hence, it has to be shown that the chaining rules are a proper replacement for inferences with transitivity clauses. [57] proves completeness of the chaining calculus by adapting the ideas from [8]. It shows the herbrand interpretation constructed from all ground instances of a saturated set of clauses is a model also for the transitivity clauses.

Theorem 5. *A set of clauses N saturated by $R_{\mathcal{B}}$ is satisfiable if and only if N does not contain the empty clause.*

Proof sketch. In a saturated set N of clauses that contains no empty clause, all ground instances of clauses are also saturated. A sequence of herbrand interpretations I_k of the k smallest clauses in the set can be defined such that I_{k+1} can be constructed from I_k. This construction consists basically in extending the interpretation of predicate P to instance a if the next clause is a positive unit clause $P(a)$. If the next clause is not a unit clause, the interpretation does not change. The finally obtained interpretation I_n (where n is the number of clauses in N) is a model for N and the transitivity clauses and satisfies the unique name assumption. □

For a detailed proof see [57].

6.2 Distribution

For distributing the calculus, we have to find an allocation of clauses that preserves completeness. For efficiency, a clause should not be allocated to too many reasoners, preferable is a functional allocation. For ordered resolution,

each clause has a unique resolvable literal that determines clause allocation (Corollary 4). However, the BSHE calculus does not have this property. In negative chaining inferences any negative literal with transitive predicate is a resolvable literal. Fortunately, clauses do not usually contain many literals of this type. Also, hyperresolution has multiple resolvable literals. We assume the allocation of clauses is based on a given allocation of symbols sa and analyze the rules of $R_{\mathcal{B}}$ to find requirements of the allocation. For each rule, we note which allocations are complete for this rule and what are the allocation symbols. Thereby we find a complete allocation that creates as few duplicates of clauses as possible.

Ordered Chaining For ordered chaining inferences, all premises are positive unit clauses. An allocation is complete for ordered chaining if all unit clauses that consist of a positive transitive predicate literal $Q(l, s)$ are allocated to one reasoner responsible for Q. The corresponding allocation symbol for these clauses is Q, the same as for ordered resolution.

Negative Chaining Since there is no maximality requirement for negative chaining, a clause has to be allocated to all reasoners responsible for a transitive predicate contained in a negative literal of the clause. Hence, we add to the allocation symbols all transitive predicates of negative literals.

Hyperresolution The unit clauses $A_1, ..., A_n$ have to be allocated to the same reasoner as the main premise. Since this is not feasible, we propose a modification of the hyperresolution rule below.

Object Equality Cut Any allocation is complete for OECut because the rule has only one premise.

Note that the transitivity clauses have to be allocated to all reasoners because the information is necessary for deciding about allocation of clauses with multiple binary literals. But, the chaining inferences for a transitive predicate Q are only performed by the reasoner responsible for predicate Q.

The problematic rule for distribution is hyperresolution. Allocating all premises of a possible hyperresolution inference to the same reasoner is not efficient in general because it restricts the symbol allocation sa. For obtaining a complete clause allocation, we would have to allocate any pair of predicates that occurs in two negative literals of the same clause to one reasoner. If there are many clauses that contain multiple negative literals, this could prevent allocating predicates to different reasoners. We might end up with allocating all predicates to the same reasoner. Furthermore, the restrictions on the symbol allocation have to be analyzed before starting

the reasoning process. Fortunately, it is possible to define a clause allocation that is complete for any allocation of symbols if we modify the hyperresolution rule.
Hyperresolution is equivalent to a sequence of resolution inferences, where the intermediate conclusions are deleted. I.e., instead of resolving the main premise with all unit clauses $A_1, ..., A_n$ at the same time, we can resolve with A_1 and then resolve the conclusion with A_2. By repeatedly resolving the conclusion with the next unit clause, we end up with the final conclusion that is identical to the conclusion of the corresponding hyperresolution inferrence. The drawback is the higher number of derivations and the additionally necessary reductions.
As described in Chapter 5, ordered resolution can be distributed without problems. Hence, we could replace hyperresolution by the corresponding ordered resolution rule where the side premises are unit clauses. However, we prefer another solution that also avoids most of the redundant conclusions that are skipped by hyperresolution. For applying the necessary restriction to hyperresolution, we define a selection function that is designed for distribution. Note that hyperresolution is complete for first order logic for any selection function, hence we are free to adapt the selection to our needs.

Definition 27 (BSHE Selection).
Based on a precedence $>$ of predicate symbols, the selection function selects from each clause c

- *the literal $\neg P_{max}(...)$ that contains the largest predicate P_{max} of all predicates contained in a negative literal of c and*
- *all literals $\neg P(...)$ with $sa(P) = sa(P_{max})$.*
- *If c is a positive unit clause, nothing is selected.*

With this selection function hyperresolution is restricted for distribution:

Definition 28 (Restricted Hyperresolution).

$$\textit{Hyperresolution} \quad \frac{A_1 \quad ... \quad A_n \qquad \neg B_1 \vee ... \vee \neg B_n \vee D}{D\sigma}$$

where

1. *$n \geq 1$, $A_1, ..., A_n$ are unit clauses,*
2. *D is a (possibly empty) clause*
3. *$\neg B_1, ..., \neg B_n$ are selected in $\neg B_1 \vee ... \vee \neg B_n \vee D$ and nothing else is selected.*
4. *σ is the simultaneous most general unifier of A_i and B_i respectively, for all $i \in \{1, ..., n\}$*

The calculus $R_{\mathcal{B}}$ with hyperresolution replaced by restricted hyperresolution is denoted by $R^*_{\mathcal{B}}$. The restricted hyperresolution rule enables efficient distribution. It splits up one hyperresolution inference into a sequence of hyperresolution inferences, where in each inference all premises are allocated to the same reasoner. We can now define an allocation that allocates a clause to at most k reasoners, where $k = min(q, m)$ is the maximum number q of different transitive predicates that may occur in one clause or the number of reasoners m.
Like for the other calculi, the clause allocation for BSHE is based on the allocation symbols of a clause.

Definition 29 (BSHE Allocation Symbols).
For each clause c and predicate $P \in Sig(c)$, we have $P \in$ a-symbol$_{\mathcal{B}}(c)$ iff at least one of the following holds:

- *c is a unit clause.*
- *P is a transitive predicate and c contains a literal $\neg P(t_1, t_2)$.*
- *there is no literal $\neg P'(t_1, t_2)$ in c with $P' > P$*

As before, the allocation of a clause c depends on the symbol allocation, i.e. $ca_{\mathcal{B}}(c) = sa(\text{a-symbol}_{\mathcal{B}}(c))$.

6.2.1 Soundness, Completeness and Termination

Soundness of $R^*_{\mathcal{B}}$ is easily verified by checking the restricted hyperresolution rule. Also, restricting hyperresolution does not prevent termination because conclusions are always smaller than premises. Since the empty clause is the smallest clause, only a finite number of inferences is possible. Furthermore, a hyperresolution inference is equivalent to a sequence of restricted hyperresolution inferences with elimination of redundant clauses. Hence, restricting hyperresolution preserves completeness.

Corollary 5. *$R^*_{\mathcal{B}}$ is a sound, complete and terminating calculus for deciding BSHE satisfiability.*

Soundness, completeness and termination of the distributed chaining calculus $R^*_{\mathcal{B}}(a_{\mathcal{B}})$ are implied by the properties of $R^*_{\mathcal{B}}$.

Theorem 6. *$R^*_{\mathcal{B}}(a_{\mathcal{B}})$ is a sound, complete and terminating calculus for deciding BSHE satisfiability.*

Proof. According to Corollary 1 and 2, soundness and termination are preserved by distribution. It remains to be shown that the allocation $a_{\mathcal{B}}$ is complete. Then, according to Theorem 1, distributed chaining is complete. We check completeness of the allocation $a_{\mathcal{B}}$ for each rule in Definition 26.

setting	runtime
single reasoner	58 min
6 reasoners, 1 monadic	45 min
6 reasoners, 3 monadic	12 min

Table 6.1: Runtimes of Yago saturation for a single reasoner and different distributed settings.

For ordered chaining inferences, both premises are unit clauses, hence they are allocated to the same reasoner $m = sa(Q)$ that is responsible for predicate Q. Also, the premises of negative chaining are allocated to $sa(Q)$. The side premise because it is a unit clause and the main premise because Q is a transitive predicate in a negative literal. Finally, the allocation is complete for restricted hyperresolution because all literals $\neg B_i$ are selected and hence the contained predicates are allocated to the same reasoner $sa(P_{max})$ according to Definition 27. All literals A_i are unified with a B_i and hence also allocated to $sa(P_{max})$. Since all premises of an admissible inference are allocated to the same reasoner, the allocation is complete. □

6.3 Experiments

We tested distributed chaining on the ontology the calculus was developed for, YAGO (Yet Another Great Ontology). Yago was automatically generated from Wikipedia and WordNet by the database/information retrieval group at the Max Planck Institute for Informatics [56]. For efficient saturation, the ontology was translated to the Bernays-Schönfinkel Horn class with equality [57]. The obtained ontology consists of 9'918'686 clauses, that contain 248301 unary predicates (classes), 79 binary predicates (properties) and 4433159 constants (instances) and no functions. In description logic notation, the expressivity of the ontology is in $\mathcal{SF}$, it contains functional properties and transitive properties but no existential restrictions on properties. The SPASS-YAGO variant of the SPASS reasoner was extended with the chaining rules described in Definition 26. With improved transitivity reasoning, the ontology was saturated in about 1 hour [57].

We compared the runtime for different distributed settings to the runtime of a single reasoner, the results are depicted in Table 6.1. All experiments are executed on the Esslingen cluster of the bwGrid[6].
We first repeated the saturation using a single SPASS-YAGO reasoner with the configuration proposed by [57]. Our results confirmed the reported runtime of [57], the saturation was completed after 58 minutes. Additional tests showed that hyperresolution is necessary. With hyperresolution replaced by

ordered resolution, the saturation was not finished after 5 hours. This is a problem because distributed resolution does not guarantee completeness when unrestricted hyperresolution inferences are applied and restricted hyperresolution is not implemented in SPASS-YAGO.
In the first distributed setting one reasoner was responsible for all monadic predicates (i.e. the classes of the ontology) and hyperresolution was switched on. Binary predicates where distributed randomly among the other reasoners. For the monadic clauses completeness is guaranteed, because all literals are allocated to the same reasoner. We found that the saturation is also complete for clauses containing binary literals: For every hyperresolution inference, all top symbols of selected literals happen to be allocated to the same reasoner. The runtime was reduced by a quarter with this distribution. Furthermore, the reasoners for binary predicates where locally saturated after 10 minutes, only the monadic reasoner needed 45 minutes for the saturation.
In a second test we used three reasoners for the monadic predicates and three for the binary predicates. Here, an addition benefit of the distribution is exploited. For optimizing the performance, we can use different configurations for the different reasoners. Hyperresolution was switched on only for the reasoners responsible for binary predicates, for the monadic reasoners hyperresolution was switched off to guarantee completeness. Now, the saturation was finished after 12 minutes. Maybe, implementing restricted hyperresolution would further decrease the runtime.

Chapter 7

Distributed Resolution with Equalities

In this chapter, we extend the distributed resolution method to description logics with cardinality restrictions. Cardinality restrictions are used to declare certain properties as functional or restrict the property instances of a certain type, e.g., $Chair \sqsubseteq Furniture \sqcap \exists_{\geq 3} hasPart.Leg$ states that a chair has at least three legs. Unqualified cardinalities ($\mathcal{N}$) restrict only the number of property instances, e.g., $Triple \sqsubseteq \exists_{=3} hasElement$ states a triple has three elements.
If an ontology contains cardinality restrictions or functional properties ($\mathcal{F}$), the corresponding set of first order clauses contains equality literals (see Section 2.4). To deal with these equalities, a complex calculus is necessary that is incompatible with the previous communication strategy used for $\mathcal{ALCHI}$ and FOL. In particular, it is not possible to define a functional allocation for the complex calculus, some clauses have to be copied to multiple reasoners. Note that, although BSHE clauses may contain equalities as well, the calculus $R_{\mathcal{B}}$ needs no special rule for equalities. This is due to the combination of unique name assumption, absence of function symbols and horn property. Most of the equalities in input clauses can be removed, and the remaining equalities are very simple and all positive. In contrast, the distributed resolution method that covers cardinalities requires a more involved calculus for complete reasoning with complex equalities.
The extension of distributed resolution described in this chapter was first proposed in [48].

7.1 Calculus

A complete calculus that terminates on clauses obtained from ontologies that contain number restrictions is basic superposition [9, 30], an extension of ordered resolution. Like ordered resolution, basic superposition uses an

ordering of literals and selection function for restricting applicability of the resolution rules.
As usual for theories containing equalities, we assume a translation of predicates to general function symbols such that all literals are equalities (e.g. the literal $P(x)$ translates to $P(x) \approx \top$, see Section 2.6), we may still write $P(x)$ for readability purpose and call these literals *predicate literals*. Clauses are split into skeleton clause C and substitution σ representing all substitutions introduced by previous unifications. The clause $C\sigma$ is denoted as closure $C \cdot \sigma$ or alternatively a closure is denoted by enclosing non-variable subterms of $C\sigma$ that correspond to variables in C in brackets (e.g. $P([f(y)])$ for $P(x) \cdot \{x \mapsto f(y)\}$).
For distributing basic superposition, the rules we have to take care of are positive and negative superposition and hyperresolution. The other rules contain only one premise and hence distribution of the input clauses into separate sets does not restrict application of these rules. Decomposition is the only reduction rule, the other rules are inference rules. We assume rules with only one premise are applied eagerly (decomposition first) and duplicated literals in a clause are deleted.

Definition 30 (Basic Superposition).

$$\textit{Positive superposition} \quad \frac{(C \vee s \approx t) \cdot \rho \quad (D \vee w \approx v) \cdot \rho}{(C \vee D \vee w[t]_p \approx v) \cdot \theta}$$

where

1. σ *is the most general unifier of* $s\rho$ *and* $w\rho|_p$ *and* $\theta = \rho\sigma$
2. $t\theta \not\succeq s\theta$ *and* $v\theta \not\succeq w\theta$
3. *in* $(C \vee s \approx t) \cdot \theta$ *nothing is selected and* $(s \approx t) \cdot \theta$ *is strictly maximal*
4. *in* $D \vee (w \approx v) \cdot \theta$ *nothing is selected and* $(w \approx v) \cdot \theta$ *is strictly maximal*
5. $w|_p$ *is not a variable*
6. $s\theta \approx t\theta \not\succeq w\theta \approx v\theta$

$$\textit{Negative superposition} \quad \frac{(C \vee s \approx t) \cdot \rho \quad (D \vee w \not\approx v) \cdot \rho}{(C \vee D \vee w[t]_p \not\approx v) \cdot \theta}$$

where

1. σ *is the most general unifier of* $s\rho$ *and* $w\rho|_p$ *and* $\theta = \rho\sigma$
2. $t\theta \not\succeq s\theta$ *and* $v\theta \not\succeq w\theta$
3. *in* $(C \vee s \approx t) \cdot \theta$ *nothing is selected and* $(s \approx t) \cdot \theta$ *is strictly maximal*

4. $(w \not\approx v) \cdot \theta$ *is selected or maximal and no other literal is selected in* $D \vee (w \not\approx v) \cdot \theta$
5. $w|_p$ *is not a variable*

Ordered Hyperresolution

$$\frac{(C_1 \vee A_1) \cdot \rho \; ... \; (C_n \vee A_n) \cdot \rho \qquad (D \vee \neg B_1 \vee ... \vee \neg B_n) \cdot \rho}{(C_1 \vee ... \vee C_n \vee D) \cdot \theta}$$

where

1. σ *is the most general substitution such that* $A_i\theta = B_i\theta$ *for* $1 \leq i \leq n$ *and* $\theta = \rho\sigma$
2. *each* $A_i \cdot \theta$ *is strictly maximal in* $C_i \vee A_i$
3. *each* A_i *and* $\neg B_i$ *is a predicate literal*
4. *each* $\neg B_i \cdot \theta$ *is selected or nothing is selected,* $n = 1$ *and* $\neg B_1 \cdot \theta$ *is maximal w.r.t.* $(D \vee \neg B_1) \cdot \theta$.

$$\textit{Reflexivity resolution} \quad \frac{(C \vee s \not\approx t) \cdot \rho}{C \cdot \theta}$$

where

1. σ *is the most general unifier of* $s\rho$ *and* $t\rho$ *and* $\theta = \rho\sigma$
2. $(s \not\approx t) \cdot \theta$ *is selected or maximal and no other literal is selected*

$$\textit{Equality factoring} \quad \frac{(C \vee s' \approx t' \vee s \approx t) \cdot \rho}{(C \vee s' \approx t' \vee t \not\approx t') \cdot \theta}$$

where

1. σ *is the most general unifier of* $s\rho$ *and* $s'\rho$ *and* $\theta = \rho\sigma$
2. $t\theta \not\succeq s\theta$ *and* $t'\theta \not\succeq s'\theta$
3. $(s \approx t) \cdot \theta$ *is selected or maximal and no other literal is selected in* $(C \vee t \approx t' \vee s' \approx t') \cdot \theta$

$$\textit{Decomposition} \quad \frac{C \cdot \rho}{C_1 \vee Q([t]) \qquad \neg Q(\mathbf{x}) \vee C_2 \cdot \theta}$$

where

1. $\mathbf{x}$ *is the vector of* m *free variables of* $C_2\theta$ *and* $\mathbf{t}$ *is a vector of* m *terms*
2. $C \cdot \rho = C_1 \cdot \rho \vee C_2 \cdot \theta\{\mathbf{x} \mapsto \mathbf{t}\}$
3. Q *is a fresh predicate if no clause was decomposed into* $C_2\theta$ *before, otherwise the previously introduced predicate is reused*
4. $\neg Q(\mathbf{t})\tau \succ L\sigma\tau$ *for each ground substitution* τ *where* L *is the main premise of the inference that derived* $C \cdot \rho$ *and* σ *is the corresponding unifier.*

Without the decomposition rule (Definition 5.4.1 in [36]), basic superposition terminates only if there are no number restrictions on roles that have subroles [36]. The decomposition rule is applied to derived clauses before any other rule is applied. With number restrictions on roles that have subroles, basic superposition may temporarily generate clauses that are no $\mathcal{ALCHIQ}$ closures, but these are immediately reduced to $\mathcal{ALCHIQ}$ closures using the decomposition rule.
An ordered hyperresolution combines a sequence of ordered resolution inferences into one inference. This is equivalent to deleting intermediate conclusions of the corresponding ordered resolution sequence. If at most one literal is selected (i.e. $n = 1$), ordered hyperresolution is equivalent to ordered resolution. Ordered resolution is a special case of positive superposition, where $w|_p = w$, i.e. p is the root position. Hence, basic superposition is complete without the hyperresolution rule, but it is more efficient with hyperresolution.

Definition 31 (Resolution Calculus $R_\mathcal{Q}$ [30]).

$R_\mathcal{Q}$ *is the calculus with*

1. *inference rules positive and negative superposition, reflexivity resolution and equality factoring and decomposition reduction rule,*
2. *selection of all negative binary literals,*
3. *the term ordering* $\succ_Q$ *is a lexicographic path ordering (LPO, [38]) based on a total precedence* $>$ *of function, constant and predicate symbols with* $f > c > P > \top$ *for every function* f *constant* c *and predicate* P*.*

Literals that contain a function symbol are ordered first to avoid substituting the arguments of functions with function terms. Limited nesting depth of literal terms is necessary to guarantee termination of the calculus, it makes

sure only the types of clauses depicted in Table 7.1 occur when basic superposition is applied to clauses obtained from an $\mathcal{ALCHIQ}$ ontology (i.e. the set of $\mathcal{ALCHIQ}$ closures is closed under basic superposition). Strictly speaking, clauses of other types may be derived, but these are always redundant ([36], Lemma 5.3.6) and deleted immediately.

type #	closure type
1	$\neg R(x,y) \vee R^{-}(y,x)$
2	$\neg R(x,y) \vee S(x,y)$
3	$\mathbf{P^f}(x) \vee R(x, \langle f(x) \rangle)$
4	$\mathbf{P^f}(x) \vee R([f(x)], x)$
5	$\mathbf{P_1}(x) \vee \mathbf{P_2}(\langle \mathbf{f}(x) \rangle) \vee \bigvee \langle f_i(x) \rangle \approx / \not\approx \langle f_j(x) \rangle$
6	$\mathbf{P_1}(x) \vee \mathbf{P_2}([g(x)]) \vee \mathbf{P_3}(\langle \mathbf{f}[g(x)] \rangle) \vee \bigvee \langle t_i \rangle \approx / \not\approx \langle t_j \rangle$
7	$\mathbf{P_1}(x) \vee \bigvee_{i=1}^{n} \neg R(x, y_i) \bigvee_{i=1}^{n} \mathbf{P_2}(y_i) \vee \bigvee_{i=1}^{n} \overset{n}{}_{j=i+1} y_i \approx y_j$
8	$\mathbf{R}(\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle) \vee \mathbf{P}(\langle \mathbf{t} \rangle) \vee \bigvee \langle t_i \rangle \approx / \not\approx \langle t_j \rangle$

Table 7.1: The 8 types of $\mathcal{ALCHIQ}$ closures [30]. $\langle t \rangle$ denotes that term t may but need not be marked (i.e. has been introduced by a previous unification), $\approx / \not\approx$ denotes a positive or negative equality predicate. For clauses of type 6, t_i and t_j are either of the form $f([g(x)])$ or of the form x and the clause contains at least one term $f(g(x))$.

Because the set of clause types is finite and the set of symbols is finite for every given ontology, the number of clauses that can be derived is finite, too and hence basic superposition terminates for $\mathcal{ALCHIQ}$ input[30].

7.2 Allocation Method

For defining the allocation for $\mathcal{ALCHIQ}$, we have to take care of the rules positive superposition, negative superposition and hyperresolution. The other rules have only one premise, hence distribution does not restrict the application of these rules. The first consideration for distributing basic superposition are the a-symbols of the $\mathcal{ALCHIQ}$ closures. To simplify the observation, we first take advantage of the ordering restrictions imposed by basic superposition. Note that the resolvable literal definition for $\mathcal{ALC}$ applies to basic superposition, too. From Definition 30 we can see that all literals that are resolved are resolvable literals according to Definition 21. In particular, the literals $s \approx t \cdot \rho$ and $w \approx / \not\approx v \cdot \rho$ are resolvable literals. Furthermore, a close look at the ordering of Definition 31 reveals that the $\mathcal{ALCHIQ}$ closures of types 3-6 and 8 are totally ordered and types 1 and 2 contain exactly one selected literal. Only closures of type 7 contain multiple resolvable literals, but these are all predicate literals with the same predicate symbol. Hence, for finding the a-symbols of a clause, we pick the resolvable

literals and check which rules can be applied and which symbols necessarily occur in both premises.

Hyperresolution. If the resolvable literal is a positive (negative) predicate literal, the clause can be side premise (main premise) in an ordered hyperresolution inference. For positive and negative literals, the a-symbol for hyperresolution is the predicate of the resolvable literal. Only a clause of type 7 may contain multiple resolvable literals, but these are all predicate literals with the same predicate. Hence, the a-symbol is well defined.

Resolvable predicate literal, superposition at root position. For a predicate literal, if the position p is the root position, the resolvable literal of the other premise is a predicate literal, too. Positive superposition produces only redundant clauses, because $w[t]_p \approx v$ is the tautology $\top \approx \top$. Negative superposition is equivalent to ordered resolution, because $w[t]_p \approx v$ evaluates to $\top \not\approx \top$ (=false) and hence the conclusion is $(C \vee D) \cdot \theta$. Like for hyperresolution, the corrsponding a-symbol is also the predicate of the resolvable literal.

Resolvable equality, superposition at root position. For an equality literal, if the position is the root position, the resolvable literal of the other premise is an equality literal, too. The a-symbol is the top symbol of the larger argument of the equality. Note that the arguments of resolvable equalities are always comparable for $\mathcal{ALCHIQ}$ closures.

Superposition at non-root position. If the position is not the root position, $w|_p$ is a function term because according to 5.) it is not a variable. Hence, s is not a predicate, i.e. $s \approx t$ is an equality literal. s is not a variable, because variable equations are never resolvable literals of $\mathcal{ALCHIQ}$ closures. $w \approx v$ is a predicate literal, because strict subterms of function terms are always marked in $\mathcal{ALCHIQ}$ closures. I.e. if w is a function term, then $w|_p, p \neq \epsilon$ is a variable. $w|_p$ is necessarily the first function term of the literal. For literals with nested functions like $P(f[g(x)])$ the term $g(x)$ is not contained in w but part of the substitution ρ, i.e. the subterm $w|_{1.1}$ is a variable. Hence, function terms at position $p = 1$ or $p = 2$ of a resolvable literal are a-symbols of a clause.

Definition 32 summarizes the computation of a-symbol for $\mathcal{ALCHIQ}$ clauses.

Definition 32 (a-symbol for $\mathcal{ALCHIQ}$).
For an $\mathcal{ALCHIQ}$ clause c with resolvable literal L:

$$\text{a-symbol}_{\mathcal{Q}}(c) = \{A_P\text{-}symbol(c)\} \cup \{A_f\text{-}symbol(c)\}$$

where
A_P*-symbol*$(c) = P$ *if* L *is a (possibly negative) predicate literal* $(\neg)P(...)$
A_f*-symbol*$(c) = f$ *if* L *is an equality literal* $f(...) \approx / \not\approx t$ *or* L *is a (possibly negated) predicate literal* $(\neg)P(f(...))$ *or* $(\neg)P(..., f(...))$ *with unmarked term* $f(...)$.

Note that for closures of type 7 the A_P*-symbol* is identical for all possible selections, and A_f*-symbol* is not applicable. From the analysis above, we can see that the a-symbols of a $\mathcal{ALCHIQ}$ clause c are the union of A_P*-symbol*(c) and A_f*-symbol*(c). Hence, the maximum number of a-symbols for a clause is two.
The allocation of clauses to reasoners is based on the a-symbols and an allocation sa of the signature symbols:

Definition 33 (Allocation for $\mathcal{ALCHIQ}$).
The clause allocation $ca(c)$ *for the distributed calculus* $R_\mathcal{Q}(ca)$ *is defined by*

$$ca(c) = \{sa(A_P\text{-}symbol(c)), sa(A_f\text{-}symbol(c))\}$$

where $sa\colon Sig(\mathcal{O}) \to M$ *is an allocation of the signature symbols of the input ontology* $\mathcal{O}$*, including concepts introduced by the definitorial form transformation.*

An example refutation is depicted in Figure 7.2. Before translating the axioms from Figure 7.1 to clauses as defined in 2.4, the normalization defined in 2.3 is applied to obtain simple axioms. All predicates in Figure 7.2 are abbreviated by their first letter, Q is a new predicate introduced by normalization. The symbol precedence is $s > t > a > A > G > C > F > Q > R$, literals of each clause are ordered with the resolvable literals first. For brevity, we skip the subsumptions that are not relevant for the refutation, i.e. $\exists_{\leq 1}R.\neg G \sqsubseteq A$ and $C \sqsubseteq \exists_{\leq 1}R.F$ are not translated to clauses. The remaining subsumption of axiom $DL1$ is translated to clause $R1$, axiom $D3$ is translated to $G1$. The query is tested by introducing a constant a that is instance of A and $\neg C$ (clauses $A1$ and $C1$). All other input clauses are from the remaining subsumption of $DL2$. If the distributed resolution process

$$\begin{aligned} DL1: &\quad Awardee \equiv \exists_{\leq 1}Rated.\neg GradeA \\ DL2: &\quad CandidateScholar \equiv \exists_{\leq 1}Rated.Failed \\ DL3: &\quad GradeA \sqsubseteq \neg Failed \\ query: &\quad Awardee \sqsubseteq CandidateScholar \end{aligned}$$

Figure 7.1: Description logic example.

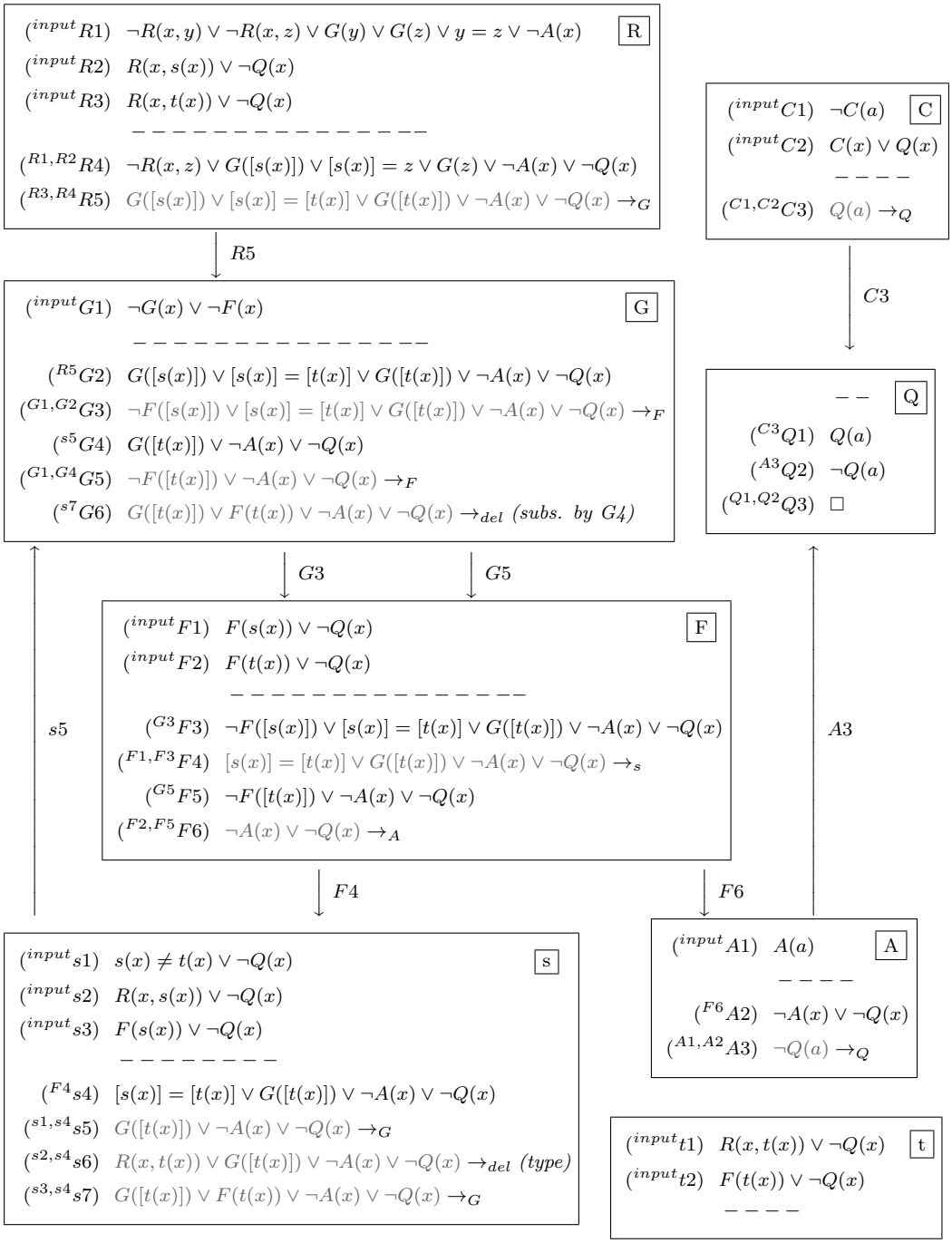

Figure 7.2: Distributed resolution example with equalities.

finds a proof (i.e. derives an empty clause $\square$) we know that $A \sqcap \neg C$ cannot have an instance and hence $A \sqsubseteq C$ holds.
For simplicity, every predicate and function is allocated to a different reasoner depicted by a box containing the processed clauses. Input clauses translated from the ontology and query are depicted above the horizontal dash lines. Superscripts in the clause labels refer to the origin of the clause, i.e. either two clauses from the same reasoner or a single clause that was propagated from another reasoner. Clauses that are deleted locally because they are propagated or redundant are set in gray.
The first box depicts the reasoner responsible for predicate R. Here the clause $R4$ is derived from clauses $R1$ and $R2$. Then, $R5$ is derived from $R3$ and $R4$ and propagated to the reasoner responsible for G because G is the predicate of the resolvable literal $G([s(x)])$. $R5$ is not propagated to the reasoner responsible for s because the term $[s(x)]$ is marked i.e. introduced by previous unification.
Note that the duplicated literal $\neg Q(x)$ is deleted immediately in clauses $F4$, $F6$ and $s5$. Clause $G6$ is deleted because it is subsumed by clause $G4$. Clause $s6$ is deleted because it is not of a type listed in Table 7.1, hence we know it is redundant and we do not need to search for the subsumer $G4$. The proof for the query is found by the reasoner responsible for Q, by deriving an empty clause from two received clauses. The boxes can be merged arbitrarily to reduce communication between reasoners. E.g. if a single reasoner is responsible for predicates A, C and Q, the corresponding boxes are merged and the clauses $C3$ and $A3$ are not propagated.

Before we address completeness of this distributed resolution method, we turn to redundancy problems that require additional restriction of inferences.

7.3 Restricted Inferences

With the above allocation, no necessary inference is skipped, but some inferences may be duplicated. For example, two clauses with resolvable literals $P(f(x))$ and $\neg P(f(x))$ would be allocated to the reasoner responsible for P and the reasoner responsible for f. Hence, both reasoners would perform the inference with these two premises. Fortunately, all duplications of inferences can be avoided by adding another allocation restriction to the basic superposition calculus. E.g. we restrict the inference with $w\rho|_p = w\rho$ to the reasoner responsible for P.

Definition 34 (Allocation Restrictions for Distributed $R_\mathcal{Q}$).
$R^*_\mathcal{Q}$ is the calculus $R_\mathcal{Q}$ with the restriction

$$sa(topSymbol(w\rho|_p)) = localID$$

added to both superposition rules. localID $\in M$ is a constant that identifies the reasoner that computes the inference.

Definition 34 is more restrictive than the standard allocation restriction $\exists m \in M : a(((C \vee s \approx t) \cdot \rho), m) \wedge a((D \vee (w \approx / \not\approx v) \cdot \rho), m)$. In the above example there are two $m \in M$ that comply with the standard restriction but for only one of them we have $sa(topSymbol(w\rho|_p)) = m$. The global restriction corresponds to checking locally, if the top symbol of the unified term is a local symbol. In the above example, the reasoner responsible for f would not resolve the two clauses, because $topSymbol(w\rho|_p) = P$ for this inference. But, the reasoner responsible for P would perform this inference and the calculus is still complete.
The idea is generalized to first order logic based on the term that is unified with the corresponding term of another clause in an inference rule of the calculus:

Definition 35 (General Allocation Restriction).
For a distributed resolution calculus $R(ca)$ with non-functional allocation ca based on symbol allocation sa, the restriction

$$sa(topSymbol(t)) = localID$$

where t is the unified term, is added to every rule $r \in R$ with more than one premise.

In contrast to $\mathcal{ALCHIQ}$, the literals can have arbitrary nesting depth in full first order logic. Consequently, the number of a-symbols of a clause is not limited, i.e. we can expect a considerably higher communication overhead than for $\mathcal{ALCHIQ}$. However, clauses that reach a certain nesting depth are often ignored to obtain an efficient approximation and decidable subsets of first order logic usually impose a limit on the nesting depth. Hence we expect that efficient variants of distributed resolution can be defined for many relevant subsets of first order logic.

7.4 Completeness and Termination

Obviously, distributing resolution does not add any inferences, hence soundness is preserved. We first proof the completeness and termination of $R^*_{\mathcal{Q}}(ca_{\mathcal{Q}})$ for $\mathcal{ALCHIQ}$ and then address the completeness for full first order logic.

Theorem 7 (Completeness of Distributed Resolution for $\mathcal{ALCHIQ}$)**.** *Distributed resolution with calculus $R^*_{\mathcal{Q}}(ca_{\mathcal{Q}})$ decides $\mathcal{ALCHIQ}$ satisfiability.*

Proof. Assume that all reasoners are saturated. We have to show that if the set of all input clauses is unsatisfiable, an empty clause is derived by one of the reasoners. Since $R_{\mathcal{Q}}$ is refutation complete, it remains to be

shown that the union $Wo_\cup$ of all Wo sets is a saturated set of clauses, i.e. every clause that can be derived from $Wo_\cup$ by $R_\mathcal{Q}$ is already contained in $Wo_\cup$ or subsumed by a clause in $Wo_\cup$. Then, if the input clauses are inconsistent, $Wo_\cup$ contains an empty clause and hence one of the separate Wo sets contains an empty clause.
Assume in contrary, that $Wo_\cup$ is not saturated, i.e. there is a non-redundant clause $n \notin Wo_\cup$ that can be derived from clauses $c \in Wo_\cup$ and $c' \in Wo_\cup$. The derivation of n required that a term from premise c is unified with a term of premise c'. The unified term is not a variable because in Definition 30 $w|_p$ is required to be not a variable and $s \cdot \rho$ is the larger argument of an equality which is no variable if $(s \approx / \not\approx t) \cdot \rho$ is resolvable literal of an $\mathcal{ALCHIQ}$ closure. Hence, every possible superposition inference with a premise c requires unification of a term with top symbol in a-symbol(c) and every possible partner clause c' must contain a resolvable literal that contains a term with top symbol in a-symbol(c). There is a symbol $S \in$ a-symbol(c) $\cap$ a-symbol(c') which implies $a(c, sa(S)) \wedge a(c', sa(S))$ i.e. c and c' are both allocated to reasoner $sa(S)$. But, since the Wo set of this reasoner is saturated, and the clause n is not redundant, n is contained in the reasoners Wo set and hence $n \in Wo_\cup$ □

Hence, if an empty clause can be derived from a set of input clauses by $R_\mathcal{Q}$, the empty clause is also derived by $R^*_\mathcal{Q}(ca_\mathcal{Q})$. The allocation $ca_\mathcal{Q}$ ensures all premises of a possible inference meet in one module. A clause is allocated to at most two modules, every inference is unique, i.e. the same clause is not derived again in another module from the same premises. Local saturation of the local clause sets is enough to guarantee completeness of the method. Termination of $R_\mathcal{Q}$ obviously implies termination of the distributed variant $R^*_\mathcal{Q}(ca_\mathcal{Q})$ since no inferences are added by distribution and the only necessary reduction rule has only one premise.

Corollary 6. *Distributed basic superposition terminates on $\mathcal{ALCHIQ}$.*

If we apply $R^*_\mathcal{Q}(ca_\mathcal{Q})$ to full first order logic with equalities, termination is obviously not guaranteed. We can preserve completeness when distributing basic superposition for FOL. But, many propagations may be necessary and the communication overhead may outweigh the computation power added by distribution. For completeness for FOL, we have to consider any predicate or function symbol contained in a resolvable literal as a-symbol.
Furthermore, for the proof of Theorem 7, we used the restriction that unified terms are never variables. In contrast, for first order logic, the input could contain a clause $x \approx y$ consisting of only one literal. It has two terms that could be unified with terms of other clauses, both are variables (namely x and y) and hence can be unified with any term. Consequently, we have to allocate this clause to all reasoners.

However, real world tasks do not contain this type of clause, as it states that any two constants are identical. Also, the nesting depth of functions is often limited in real world theories. For subsets of first order logic that provide an efficient calculus, we expect that a complete and efficient clause allocation can be defined.

7.5 Implementation

Our distributed resolution implementation is based on the first order prover SPASS[8] [64]. For extending the implementation described in Section 5.4 to equalities, the calculus and the allocation are changed.

Basic superposition requires recording previous unifications which is not implemented in SPASS. But, SPASS implements superposition. In this calculus, the restriction "$w|_p$ is not a variable" of positive and negative basic superposition is replaced by "$w|_p \cdot \sigma$ is not a variable". Hence, superposition adds inferences that are skipped by basic superposition and is also complete for $\mathcal{ALCHIQ}^-$. Basic superposition is more efficient and hence the runtimes and number of derived and propagated clauses would be smaller with basic superposition. However, we can use superposition alternatively for extending the supported expressivity of distributed resolution to equalities. This greatly simplifies the implementation because we can use the distributed resolution implementation described in Section 5.4. For changing the calculus, we only adapt the configuration of SPASS to enable superposition[9].

The allocation of a clause is extended in two ways. First, there are not only predicate literals but also equalities. If a resolvable literal is an equality, the allocation is defined by the top symbol of the larger argument (which is always a function symbol). Second, predicate literals that contain a function term are additionally allocated to the reasoner responsible for the top symbol of this function term. In the implementation, the first allocation symbol is either a function or predicate symbol and a derived clause is always sent to the responsible reasoner (if it is not the local reasoner). The second allocation symbol is either a function symbol inside a predicate literal or null. A clause is send to the reasoner responsible for the second allocation symbol if it is not null and the responsible reasoner is different from the local reasoner and different from the reasoner responsible for the first allocation symbol.

[8] `http://www.spass-prover.org`

[9] The complete configuration for Spass is: Distributed=1 Auto=0 Splits=0 Ordering=1 Sorts=0 Select=3 FullRed=1 IORe=1 IOFc=1 IEmS=0 ISoR=0 IOHy=0 -IERR=1 -ISpR=1 -ISpL=1 -IEqF=1 -RFSub=1 -RBSub=1 -RInput=0 -RSSi=0 -RObv=1 -RCon=1 -RTaut=1 -RUnC=1 -RSST=0 -RBMRR=1 -RFMRR=1

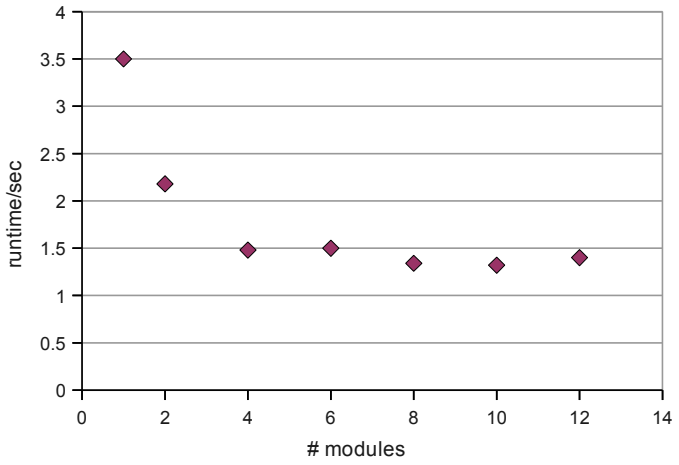

Figure 7.3: Runtimes for saturation of the SWEET ontology, using different numbers of reasoners.

7.6 Experiments

The experiments were executed on the Esslingen cluster of the bwGrid[6] using 2.27GHz CPUs with 24G memory. Our implementation was tested on the Semantic Web for Earth and Environmental Terminology (SWEET [42]), a set of linked ontologies published by the NASA Jet Propulsion Laboratory. The ontology network describes 5050 classes and 106 individuals, translation to first order logic yields 9112 clauses. We replaced datatype properties by object properties and nominals by common concepts because the current version of our system does not support them. Transformation of transitivity axioms did not add new axioms. The expressivity of the obtained test ontology network is $\mathcal{ALCHIN}$.

The time needed for saturating the merged ontology is short compared to the FMA ontology we tested in the previous chapter. Despite the small size of the ontology, Figure 7.3 shows runtime is still reduced by using more machines for the saturation. In our setting, the optimal number of reasoners for the task is 4, adding more reasoners does not reduce runtime considerably.

Ideally, the runtime in the distributed setting would equal the runtime of a single process divided by the number of reasoners used for the task. This would imply a decrease in runtime by 50% when using 2 reasoners instead of 1. But, here the decrease is 38% and from 2 reasoners to 4 reasoners the decrease is 32%. Note that without equalities, the runtime decrease was considerably higher in Chapter 5.

To find the specific reason for convergence, we investigate the distributed saturation in more detail. First, one reasoner may have a larger portion of the work assigned than the others. Since the runtime of the system is the

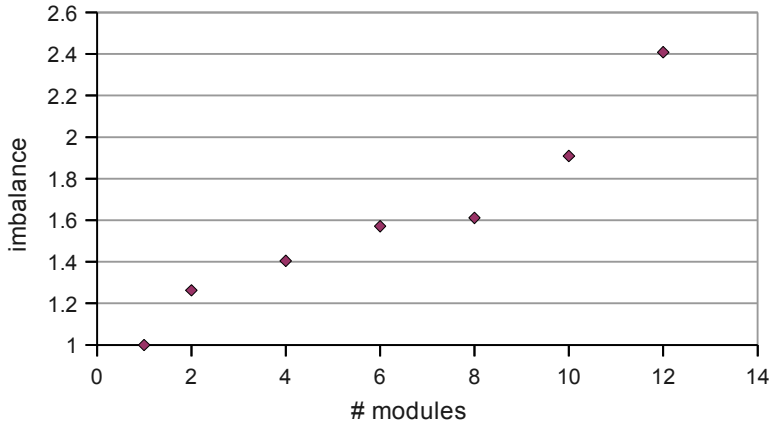

Figure 7.4: Balance of SWEET saturation.

maximum runtime of the participating reasoners and not the average, this increases runtime of the system. The imbalance depicted in Figure 7.4 is measured by comparing the maximum number of derived clauses of a single reasoner to the overall number of derivations. Obviously, a single reasoner is optimally balanced. With 4 reasoners, one of the reasoners derives 40% more clauses than the average. For six reasoners, this number increases to more than 50%.

Another possible reason for runtime convergence is additional work caused by distribution. Redundant clauses are only deleted if they are redundant with respect to the module that derives or receives them. Hence, some redundant clauses may be kept in the distributed setting, that are deleted in the conventional saturation process. Additionally, propagated clauses are printed and then parsed again when they are received, leading to updates of local index structures. Additional work is indicated by the overall number of derivations and propagations in Figure 7.5. The total number of derivations does not increase considerably with increased number of reasoners. I.e. the limited redundancy check is not an issue. As expected, the number of propagations increases. The question is, whether this increase explains the curve of runtime in Figure 7.3. Compared to the experiments in Chapter 5 the number of propagations and the increase is higher. We observe an average of 3700 propagations per second for two reasoners and an increase by 57% for 4 reasoners. This is considerably more than the 37% we observed for NCI and FMA. The difference is even larger when we take into account the imbalance depicted in Figure 7.4. Like for the NCI, imbalance increases linearly with the number of reasoners. But, imbalance values are considerably higher for SWEET. For 8 reasoners the imbalance is 1.61 while for NCI it was only 1.19. I.e., for SWEET there is one reasoner that performed 61% more derivations than the average.

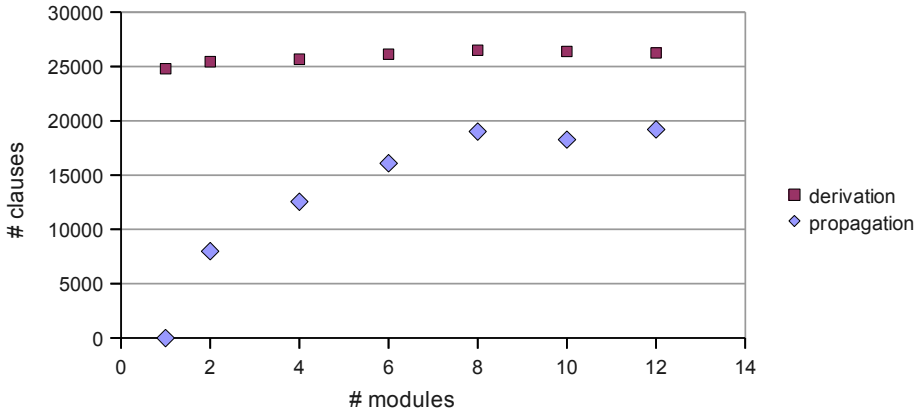

Figure 7.5: Number of derivations and propagation for saturation of SWEET ontology.

Note that although we used superposition in the experiments, the results also allow drawing conclusions about the performance of distributed basic superposition. The more efficient basic superposition calculus would require less derivations and reduce the runtime for both the single reasoner setting and the distributed setting. In particular, the total amount of redundant inferences avoided by basic superposition is similar for the distributed setting and the single reasoner setting. Hence, we can expect that distributing basic superposition reduces runtime similar as distributing superposition.

To sum up, in comparison to computation on a single reasoner, runtimes are decreased by distributed superposition. However, the decrease in runtime is smaller than for ordered resolution due to a higher number of propagations and higher imbalance of workload.

Part III

Allocation

Chapter 8

Partitioning

Distributed resolution requires an allocation of clauses to reasoners, that is based on an allocation of the signature symbols of the ontology. This allocation of symbols is obtained from a partitioning of the ontology terms by allocating each part to one reasoner. In the previous chapters we assumed a partitioning of the ontology terms is given, for example by the namespaces of a set of linked ontologies or by a random separation into disjoint sets. The symbol allocation has a considerable influence on the performance of the system. First, the number of clauses processed by each reasoner and hence the workload distribution depends on the symbol allocation. Second, the allocation of symbols determines if a derived clause is propagated, i.e. the necessary amount of communication between reasoners. Hence, a random allocation is probably not the best solution, we can expect a better performance for allocations that are designed for the requirements of distributed resolution.

8.1 Related Work

There are a couple of approaches to ontology modularization, corresponding to various use cases. The approaches can be classified into ontology partitioning and module extraction. The former addresses the task of turning one ontology into a set of linked ontologies, while the latter extracts a module from an ontology, e.g. for reusing it in another ontology. Despite the different focus of these approaches, one task can be reduced to the other. In particular, we can partition an ontology by repeatedly extracting parts from it and we can extract a module from an ontology by first partitioning the ontology and then choosing one of the created ontology modules. Module extraction approaches usually start with a set of ontology terms and traverses the graph representation of the ontology, selecting concepts, properties and axioms based on heuristics for their relevance to the desired module [18]. Ontology partitioning methods are either based on graph par-

titioning or use logical criteria for determining the partitioning. The most prominent logical approach was proposed for partitioning in [35], it choses modules such that the original ontology is a *conservative extension* of the module. This means that all axioms in a specified local part of the module signature that are implied by the original ontology are also implied by the module alone. Obviously, reasoning in an ontology partitioned into modules with this property is very efficient, since many axioms are known to be irrelevant for answering a given query. But, the reduced reasoning effort is traded by very complex methods for creating and preserving the conservative extension property. Furthermore, this partitioning method may create partitionings where one of the modules contains more than 80 % of the axioms of the whole ontology. More balanced partitioning is achieved by graph based methods. These create a graph from the ontology and apply graph partitioning methods [52] or traverse the graph and collect concepts and axioms [51]. Since our distributed reasoning approach requires balanced sizes of the modules, we rely on graph partitioning.

8.2 Graph-based Ontology Partitioning

Our partitioning method extends the graph based approach described in [52]. It first encodes dependencies between elements in the ontology in a graph structure. In a second step the obtained graph is partitioned using graph partitioning methods. Finally, a distributed ontology is created from the original ontology, based on the graph partitioning. We first give an overview over the three steps before explaining each step in detail. The partitioning method is not restricted to a specific application, it can be adapted to different requirements by choosing different algorithms for the subtasks and adjusting the parameters of the applied algorithms. We describe the method in general, address adaption for distributed resolution and compare the performance of different methods subsequently.

8.2.1 Step 1: Create Dependency Graph

In the first step, a graph structure is created that represents the dependencies between elements in the ontologies. The nodes of the graph correspond to classes and properties of the ontology. If the ontology to be partitioned is represented by clauses, unary predicate symbols correspond to class names and binary predicates to properties. In addition to predicates, function symbols that are introduced by skolemization may be considered for the dependency graph. An edge is created between two symbols, if there is a reason to prefer allocating both symbols to the same ontology part. The edges and weights of edges are determined by the application, it corresponds to the overhead generated by separation.

8.2.2 Step 2: Graph Partitioning

In the second step, a graph partitioning algorithm is used to determine sets of ontology elements that should be in one module. In principle any graph partitioning algorithm can be used for this task. The suitability of an particular algorithm depends on the requirements of the application that uses partitioned ontologies. Prerequisite for applying structural partitioning is that the requirements can be formulated as, or approximated by a graph partitioning objective function. For distributed reasoning, this objective function consists of minimizing the edge cut (that approximates communication overhead) and maximizing the balance of partition sizes.

8.2.3 Step 3: Partition Realization

Finally, a distributed ontology is created based on the graph partitioning. The straight forward realization allocates each axiom of the ontology to one of the modules. To allow for a more fine grained distribution, large axioms can be converted to a set of smaller axioms. If redundancy in the distributed representation of the ontology is allowed, each axiom may be copied to different modules to reduce dependencies between the modules. For distributed resolution, partition realization is not essential as we can use the allocation function for creating a partitioned clausal representation of the ontology. However, partition realization is still useful for obtaining a distributed DL representation of the ontology.

8.3 Dependency Graph

There are numerous ways of creating the dependency graph for a given ontology. We distinguish three types of dependency graphs.

8.3.1 Based on DL Axioms

Since an ontology is a graph by definition, this graph could be used directly as dependency graph. The nodes are the classes, properties and individuals, edges are subclass relations, instance relations and property relations. But, there are some problems with using the ontology graph directly: First, edges and nodes are not disjoint in the ontology graph. E.g. the same property corresponds to a node in the domain definition of the property and it corresponds to an edge in any property assertion. Most of the available graph partitioning methods do not allow overlap between the set of edges and the set of nodes of the graph. Furthermore, usually a single type of edges is assumed and weights for nodes and/or edges are required. Hence, the edges of the ontology graph are not used directly as edges of the dependency graph. Options for generating edges include

- Link a class to its (direct) subclasses
- Link a class to its instances
- Link a property to direct subclasses of its domain and range
- Link all direct subclasses of the domain of a property (classes that share a property)
- Link classes and properties that appear in the same axiom
- Link classes or properties that have similar names

All these options can be combined and weights can be assigned to each type of edge. This method for dependency graph creation is used by the PATO partitioning tool.

8.3.2 Based on Clauses

For distributed resolution, the axioms are represented as clauses, hence this representation can be used as basis for the dependency graph. There are different options for defining the edges and the weight for edges and nodes.

Nodes The nodes are the predicates and function symbols appearing in the clauses. The node weight is supposed to approximate the workload caused by a symbol. The most obvious approximation is the number of occurrences of a symbol. Alternatively, we could count the number of clauses or the number of literals, that contain the corresponding symbol. The difference between these numbers is probably small.

Edges Edges are introduced between nodes if there is a clause that contains both corresponding symbols. The weight of the edge is the number of clauses that contain both symbols. It can be restricted to count only pairs of symbols that occur in consecutive literals according to the ordering.
The edge weight should estimate the second performance factor, i.e. the communication costs caused by the separation of two symbols. The communication costs depend on the number of propagated clauses, i.e. clauses c with a parent clause p such that the top symbols of c and p are allocated to different reasoners. In ordered resolution, the top symbol of a derived clause is the top symbol of the second literal (assuming the clauses are ordered) of one of the parent clauses. Hence, if the order is given, the top symbols of consecutive literals are probable to be top symbols of a derived clause and its parent. For indicating that these symbols should be resolved by the same reasoner, we add an edge. If the order is not given (e.g. because it is computed after partitioning) any pair of symbols occurring in a clause may cause a propagation.

8.3.3 Based on Derivation

Another option for the dependency graph is to measure the actual size of the modules and the costs caused by separation. I.e. for distributed resolution we record the used computation resources and communication overhead. The computation effort for a symbol S is the computation time spend on derivations and reductions on clauses with top symbol S. It can be approximated by the number of derived clauses with top symbol S. The communication overhead corresponds to the number of propagated clauses. Obviously, this method is not applicable in general, as performing a reasoning task is required to create the dependency graph. However, the method is useful for evaluating other dependency graph methods.
The runtime of a distributed saturation indicates the quality of the whole partitioning method. Using the derivation graph, we can distinguish between the quality of the dependency graph and the quality of the graph partitioning method. We start reasoning with an arbitrary partitioning and record node weights and edges for every derived clause. The obtained dependency graph is what we are trying to approximate with the clause-based method. Furthermore, runtime logs of propagation and derivation can be used for the dynamically optimized partitioning presented in Chapter 9.

8.4 Graph Partitioning

After creating a dependency graph from the ontology, standard graph partitioning methods can be used for partitioning the ontology symbols. We used two available graph partitioning tools and additionally we implemented a simple algorithm for distributed resolution.

8.4.1 Greedy Balance

One of the simplest partitioning algorithms ignores edges for creating a balanced partitioning. The greedy balance algorithm repeatedly assigns the node with the highest weight to the smallest part until all nodes are assigned to one part. This method can serve as a baseline for evaluating other partitioning algorithms. We expect lower communication costs for algorithms that also take into account the edges of the graph.

8.4.2 Balanced Edge Cut

Graph partitioning methods are well known for resource optimization e.g. in grid clusters. According to the two important properties of the resulting partitioning, there are algorithms available that minimize both the *edge cut* and the *imbalance*. The edge cut is the sum of the weights of edges connecting different parts, it corresponds to the overhead introduced by separating the

whole system into parts. The *imbalance* of the size distribution corresponds to the balance of workload distribution. There are different definitions for imbalance, we use the size of the largest part divided by the average size of a part.

Definition 36 (Imbalance).
The imbalance of a partitioned graph is

$$imbalance = \frac{n \cdot size_{max}}{size_{graph}}$$

where n is the number of parts of the partitioning, $size_{graph}$ is the sum of node weights in the graph and $size_{max}$ is the sum of the node weights in the largest part.

A perfectly balanced partitioning has an imbalance value of 1, the maximum imbalance of n is reached when one part contains all nodes and the other parts are empty. For defining the optimal partitioning, we would have to combine edge cut and imbalance into a single objective function. But, since computing an optimal solution is not possible for very large graphs anyway, available tools compute good solutions and do not define the relative importance of edge cut versus imbalance explicitly.
The common algorithms for this type of partitioning used by [31, 25] apply a coarsening-refinement approach. They repeatedly compute a simpler graph (coarsening), then partition the simpler graph and finally refine the graph stepwise back to the original graph.
For coarsening the graph, neighboring nodes are merged and the weight of the merged node is set to the sum of the weights of the original nodes, thus preserving the essential properties of the graph. Many graph partitioning methods are designed for nodes with coordinates, in our case we can use the greedy balance algorithm for partitioning the coarsest graph. One of the most popular methods for partitioning refinement is the iterative method proposed by Kernighan and Lin [32]. It starts with an arbitrary initial bisection of the graph and reduces the edge cut in each iteration by swapping a set of nodes from one part with a set of nodes from the other part. A more efficient variant is described in [21]. The method was extended to graph quadrisection [55] and to arbitrary numbers of sets, edge weights, and node weights [26]. The refinement technique is particularly relevant for the dynamic allocation described in the next chapter.

8.4.3 Islands Algorithm

The island algorithm was proposed for ontology partitioning in [52, 46, 54]. The idea is to create clusters such that the edges inside a cluster are stronger than edges connecting different clusters. In particular, the algorithm computes all maximal line islands [10]:

Definition 37 (Line Island).
A subset of nodes I of a graph is a line island in the graph, if and only if there is a spanning tree T over nodes in I such that

$$\max_{(u,v)\in E, v\notin T} w(u,v) < \min_{(u,v)\in T} w(u,v)$$

where u and v are nodes and E are the edges of the graph. (u,v) is the edge connecting u and v and $w(u,v)$ is the weight of this edge.

For the determination of the maximal spanning tree the direction of edges is not considered. In the application for ontology partitioning, [53] normalized the graph before computing the line island and added optimization for obtaining more balanced partitions. After normalization, the sum of the weights of connected edges equals 1 for every node in the graph. The computed line island partitioning often creates parts that consist of a single node, these are merged into a neighboring part. Furthermore, a high minimum edge weight in a cluster indicates a small size of the cluster. Hence, [53] provides an option for merging clusters with minimum edge weight above a specified threshold.

8.5 Partition Realization

In the previous sections, we showed how a partitioning of ontology symbols can be created. For running a distributed resolution task, this type of partitioning is sufficient. However, when different modules of the ontology are developed independently in collaborative maintenance or when modules are used independently of others, a modular ontology representation is required. For obtaining a connected set of ontology modules the axioms of the given ontology are partitioned into sets of axioms. I.e. we have to create a partitioning of axioms based on the partitioning of symbols.
Methods for computing this ontology network are described in terms of a mapping that assigns a set of symbols to each axiom. Each module corresponds to one part of the symbol partitioning and contains all axioms that are mapped to at least one of the symbols in the part.

Definition 38 (Partition Realization).
A partition realization function rf maps each axiom α to a set of symbols RS. A modular ontology $\{\mathcal{O}_i\}$ $(i = 1..n)$ is obtained from an ontology $\mathcal{O}$ and symbol allocation sa via the realization function:

$$\mathcal{O}_i = \{\alpha \in \mathcal{O} \mid \exists S \in rf(\alpha), i \in sa(S)\}$$

The best realization function depends on the application. The simplest realization results in a modular ontology where each module contains all axioms that contain one of the symbols allocated to the module.

Definition 39 (Local Comprehensive Realization).
The local comprehensive realization rf_1 maps an axiom to all symbols that are contained in the axiom.

$$rf_1(\alpha) = \{S \in Sig(\alpha)\}$$

If this realization is performed on the saturated ontology, each module contains most of the information about the ontology terms allocated to it. But, since the realization is a pure syntactical method, the modules are not completely self-contained. There may be axioms from the signature of one module, that are implied by the modular ontology but not by the module alone. Disadvantage of the first realization is the redundancy of the representation. One axiom may be contained in many modules. The duplicate free realization function creates a modular ontology where each axiom is contained in exactly one of the modules.

Definition 40 (Duplicate Free Realization).
The duplicate free realization $rf_2(\alpha)$ depends on the type of the axiom α:

$$\begin{aligned} rf_2(A \sqsubseteq C) &= A \\ rf_2(P \sqsubseteq Q) &= P \\ rf_2(A \equiv C) &= A \\ rf_2(C(a)) &= C \\ rf_2(P(a,b)) &= P \end{aligned}$$

where A is a concept name, C and D are concepts, P and Q are property names and a, b are constants.

The modules created by this realization are duplicate free, but less self-contained than modules created by the first realization function. For ontologies that contain general concept inclusions $C \sqsubseteq D$, we can extend the definition by choosing a symbol of the axiom randomly or normalizing the ontology. An ontology in definitorial form (Definition 2) does not contain axioms $C \sqsubseteq D$ with complex concept C, these are transformed into axioms $\top \sqsubseteq \neg C \sqcup D$. However, superconcepts of $\top$ are then all allocated to the same module which may cause imbalanced module sizes.
The described realization functions show the trade off between local completeness and redundancy of the representation. Depending on the requirements of an application we can define realizations that are more compact than rf_1 but provide more complete information in each module than rf_2.

8.6 Experiments

We tested different dependency graph and graph partitioning methods for deciding which methods are suitable for our distributed reasoning setting.

partitioning	runtime/sec	propagation	imbalance	derivation
c-greedy	7.7	94361	1.11	123478
d-greedy	8.3	93616	1.10	123439
round robin	9.88	93623	1.08	123439
c-edge	11.3	37117	1.38	123478
d-edge	20.3	5781	1.22	123439
pato-islands	255.8	930	3.89	123411

Table 8.1: Comparison of partitioning methods.

For the comparison depicted in Table 8.1, we saturated the NCI ontology distributed to 4 reasoners using different partitioning methods. Apart from the round robin allocation, each partitioning method consists of a combination of dependency graph creation and graph partitioning method. The dependency graph was created from the clauses (c-greedy, c-edge) or the derivation graph (d-greedy, d-edge) or from the OWL axioms (pato-islands). For the pato-islands partitioning, it is not possible to specify the number of created parts in advance. Pato created a partitioning consisting of 24 parts. The smaller parts were merged into one part to obtain a partitioning of 4 parts.

Applied graph partitioning methods are greedy balance (c-greedy, d-greedy), balanced edge cut (c-edge, d-edge) and islands algorithm. The round robin partitioning is the simplest strategy. It is based on the index number $i(S)$ of a symbol S, i.e. the allocation is $sa(S) = i(S)\%4$. For balanced edge cut partitioning, different algorithms are available. In preliminary investigations, we tested two free implementations, METIS[10] [31] and CHACO[11] [25]. The properties of the resulting partitionings are very similar [41], but only metis is able to create partitionings for arbitrary number of parts while chaco is limited to powers of two. Hence we used metis for balanced edge cut computation.

We recorded runtime of the saturation and number of propagated clauses and additionally counted the number of clauses d_m that where derived by each reasoner m. The depicted *derivation* is the sum $d_1 + d_2 + d_3 + d_4$. Consequently, *imbalance* is computed as $\frac{4}{derivation} \cdot \max_m(d_m)$ and the highest possible imbalance is 4.

Table 8.1 shows the dependency graph created based on derivations did not perform better than the clause graph. When used in combination with the balanced edge cut algorithm, the saturation required almost twice the runtime of the clause graph based variant. Hence, the clause graph approximates actual costs good enough, the derivation graph is not a more

[10] `http://glaros.dtc.umn.edu/gkhome/views/metis`

[11] `http://www.sandia.gov/~bahendr/chaco.html`

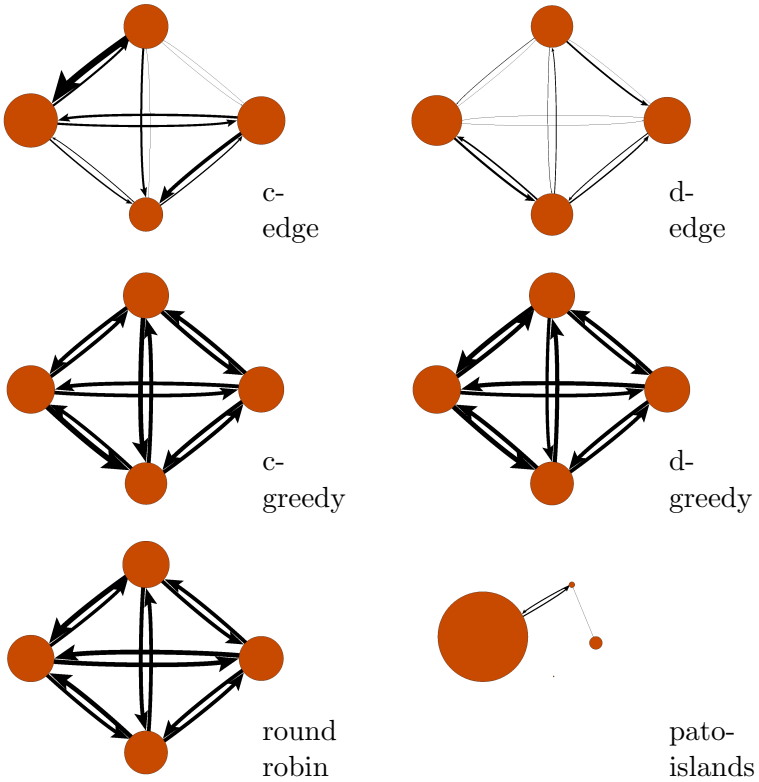

Figure 8.1: Derivation graphs of NCI saturation. Comparison of partitioning methods.

exact approximation. Furthermore, the greedy strategy and even the simple round robin outperformed the balanced edge cut partitioning. Actually, round robin is a bit similar to the greedy strategy because for computing the precedence, symbols are ordered according to their frequency in the input. Hence, the four most frequent symbols are allocated to different reasoners by both greedy and round robin partitioning.

Figure 8.1 gives are more detailed picture of balance and propagation for the different partitioning methods. The reasoners are depicted by nodes, with the size corresponding to the number of derivations of the reasoner. Arcs depict propagation, the width corresponds to the square root of the number of propagations. For c-greedy, d-greedy, and round robin partitioning, the graphs are very similar. Both node weights and edge weights are well balanced. For balanced edge cut partitioning, the number of propagations is much lower. Especially for c-edge, derivations and propagations are less balanced. The pato-islands graph shows extreme imbalance and extreme low propagation.

Variation in the number of derivations is very low. Even the highest number

of observed derivations is only 1‰ higher than the number of derivations without distribution.
Probably, the reason for bad performance of balanced edge cut is too much emphasis on the edge cut minimization. Apparently, the cost of propagation is not very high in our setting and hence it is more important to reduce imbalance than to reduce the number of propagations. In fact, the highest number of propagations was observed on the fastest saturation. However, it might be possible to improve the partitioning by taking into account the propagation. But, an adapted partitioning algorithm must put a strong focus on balance when reducing the edge cut. Note, that in distributed reasoning settings with slower network connection the results would be different. Depending on the network properties, algorithms with more focus on edge cut might be appropriate.
The partitioning created by pato-islands is not suitable for distributed resolution. Detailed investigation of the distributed process showed, that the merged part of the partitioning was saturated after only 3 seconds. Hence, the partitioning created by pato was considerably improved by merging the 24 parts into 4 parts. However, the results are much worse than for the simple round robin partitioning. There is no reason to expect good results when we use the pato dependency graph or the islands partitioning method in other combinations. The low propagation value is only caused by the extreme imbalance, one reasoner performed 97% of the total number of derivations.
The resource requirements for computing partitionings are very small compared to the runtime of the saturation. Only the pato and island methods require several minutes, but they are not suitable anyway. The most expensive of all other methods is computing the balanced edge cut which took less than 0.2 seconds.

To sum up, the results achieved with very simple partitioning strategies are quite good. Improving the simple partitioning by taking into account propagation requires a different balanced edge cut algorithm that puts more emphasis on the balance.

Chapter 9

Dynamic Allocation

The previous chapter investigated methods for creating the allocation of signature symbols to reasoners that is required for distributed resolution. While an arbitrary allocation is acceptable in theory, the performance of the reasoning process depends on the quality of the applied partitioning. There are a couple of methods for creating a symbol partitionings, for some ontologies a very simple round robin strategy can create an acceptable partitioning.

However, computing a suitable allocation in advance is not always possible. For some ontologies, the input clauses give a bad estimation of the workload caused by each symbol. For example, a property symbol P may occur only once in a single clause stating that P is a superproperty of property Q. But, the reasoning process will derive a clause with top symbol P from any clause with top symbol Q which can cause a very high workload for symbol P. Although in this special case we can adapt node weights of the clause graph to improve the partitioning, there is no way to give a reliable prediction of the workload of each symbol. In general, we cannot give any guarantee for the quality of a symbol partitioning before the reasoning task is finished. For improving the workload balance on reasoning tasks that start with an initial partitioning of bad quality, we need means to improve the partitioning at runtime.

Another problem with the partitioning methods discussed in the previous chapter arises when distributed reasoning is applied in a dynamic environment. Our distributed reasoning method envisions reasoning over very large ontologies composed of connected ontology modules. For very large ontology networks, computation of reasoning tasks may take several hours despite parallel computation. During the reasoning process, the availability of computation resources may change. Additional compute nodes may become available or used nodes may have to stop and continue working on a different task. For this setting, it is necessary to move workload between nodes.

Finally, the workload caused by a given symbol usually changes during rea-

soning time. For example, at the beginning of a task many given clauses have the top symbol S, but later given clauses with top symbol S are very rare. Hence, the runtime of the whole process might be decreased if we could use different symbol partitionings at the beginning and at the end.
Therefore, the extension to distributed reasoning proposed in this chapter aims at solving multiple problems. It enables improving the initial partitioning at runtime and adapting the partitioning to changing workload of the signature symbols. Furthermore it allows adding and removing compute nodes during computation.
Advantages of changing allocation at runtime are evident. But, changing the allocation of a clause at runtime could easily lead to flaws in the reasoning process, inferences could be skipped accidentally. Furthermore, temporary differences in the allocation tables of the reasoners could result in oscillating clauses that are send back and forth between reasoners, causing high network traffic. For solving these problems, we propose a representation of dynamic reallocation that structures a changing allocation into a sequence of modifications. After explaining the theory of reallocation, challenges imposed by delay in the communication are addressed.

9.1 Reallocation

Recall, that the allocation of clauses is induced by an allocation of symbols to reasoners. Hence, two main tasks have to be performed to change the allocation of clauses at runtime. First, the symbol allocation tables of every reasoner need to be updated to the new allocation. Second, the set of clauses with changing allocation must be moved from the reasoner that was previously responsible for these clauses to the reasoner that will be responsible for resolution on the clauses next. To keep the reallocation process simple, we first consider one reallocation task at a time.

Definition 41 (Reallocation Task).
A reallocation task is a tuple (sa_1, sa_2, m_1, m_2) *of two allocations and reasoners such that for all symbols* S*:*

- *either* $sa_1(S) = sa_2(S)$*, or*
- $sa_1(S) = m_1$*, and* $sa_2(S) = m_2$*.*

The symbols S *with* $sa_1(S) \neq sa_2(S)$ *are the reallocated symbols of the reallocation task.*

Hence, a reallocation task is a pair of allocations, such that one reasoner m_1 is responsible for all reallocated symbols before the change of allocation and a reasoner m_2 is responsible for the reallocated symbols after the change. The symbols that do not change allocation can be allocated to any reasoner. Dynamic allocation of clauses can be described as a series of reallocation tasks, where every task consists in changing the allocation of a set of symbols.

9.1.1 Completeness of Distributed Calculus

In theory, switching the allocation is performed instantly for all symbols and corresponding clauses. Correctness, completeness and termination of a distributed resolution calculus do not depend on any property of the allocation. In particular, it is not required to remain unchanged. The only assumption is that for each derivation the allocation symbols are allocated to a single well defined reasoner. In between two derivations, the allocation may change without any effect because a resolution rule is applicable to a pair of clauses if they share an allocation symbol. The specific allocation of the shared and unshared allocation symbols is not relevant.

Corollary 7 (Instant Reallocation)**.** *The distributed resolution calculus $R(ca)$ with clause allocation ca based on a dynamic symbol allocation sa is complete if the calculus $R(ca)$ is complete with static allocation.*

For the calculus, a change in the allocation is no problem. The challenge of dynamic reallocation is caused by physical distribution.

9.2 Dynamic Allocation Algorithm

It is not sufficient to change the local allocation tables, we also have to search for local clauses that are now allocated to a different reasoner and propagate them accordingly. Changing the physical location of clauses takes some time. Note, that in a setting with enough shared memory for storing all clauses we could avoid moving a clause to another location. But, changing the allocation still requires expensive updates of index structures and synchronization.
A simple implementation of dynamic reallocation is to stop all reasoners at the beginning of the main loop, just before picking a given clause. With the whole system at hold, all updates and propagations can be executed safely before the reasoners resume working on the updated clause sets with updated allocation tables.

9.2.1 Propagation

Clause propagation for dynamic allocation is considerably more complicated then with static allocation. Assume we have a *Wo* clause c in reasoner $sa(P)$ that is subject to propagation. The clause has two a-symbols, P and f, with different allocation $sa(P) \neq sa(f)$. Only f is a reallocated symbol. Then, we do not know if it is necessary to propagate c to the reasoner responsible for f. Maybe, c was send to $sa(f)$ already when it was derived. But, it is also possible that c was not propagated to $sa(f)$. To avoid unnecessary

propagation, we have to find an efficient way to record required information about previous allocations.
The solution to this problem is to record the type of allocation for each instance of a clause that has more than one a-symbol. I.e., if clause c is propagated to reasoner m_1 because m_1 is responsible for symbol S, we store the relevant allocation symbol $\{S\}$ with the clause. Then, we do not need to keep track of previous allocations. If c is a *Wo* clause that might require propagation, it is propagated to the reasoner responsible for S if $sa(S) \neq localID$.
For defining the relevant allocation symbols, we need to refer to different copies of the same clause.

Definition 42 (Relevant Allocation Symbols).
The relevant allocation symbol ra-symbol(c) *of a clause c is one of the allocation symbols* a-symbol(c). *A set of copies* $c_1, ..., c_n$ *of the same clause can be represented by one clause* c_m *with* ra-symbol(c_m) $= \bigcup_i^{1..n}$ ra-symbol(c_i). *Initially, for every input clause or derived clause c, all* a-symbol*s are relevant:* ra-symbol(c) = a-symbol(c).

Note that we have to record the relevant allocation symbols only for clauses that have multiple a-symbols, for other clauses ra-symbol(c) = a-symbol(c). With this information it is now possible to decide about propagation: Each clause c_i that is subject to propagation is only propagated to reasoners responsible for a symbol in ra-symbol(c_i). For extending the distributed resolution prover in Algorithm 2 to dynamic allocation, a call to the function updateAllocation(reallocationSymbols, oldAllocation, newAllocation) is inserted between line 1 and line 2 when a reallocation task is pending. Algorithm 3 depicts the updateAllocation function. It changes the allocation of the reallocated symbols to the reasoner specified by *newAllocation* and propagates the corresponding *Wo* clauses. Before returning, it waits until all reasoners have finished the allocation update.

9.2.2 Completeness of Algorithm

Corollary 7 shows the calculus allows changing the symbol allocation at any time. It remains to be shown that also the implementation in Algorithm 2 and 3 preserves completeness for dynamic allocation.

Theorem 8 (Dynamic Completeness)**.** *The satisfiability check implemented by Algorithm 2 extended by the dynamic allocation in Algorithm 3 is complete for first order clauses. For* $\mathcal{ALCHIQ}$ *clauses the algorithm terminates and decides satisfiability.*

For proving that the distributed reasoning algorithm preserves completeness, also when the allocation changes, we consider different copies of a clause for every a-symbol and show that *Wo* clauses are always saturated. The proof

Algorithm 3 Dynamic Extension for Distributed Resolution Prover

updateAllocation(*reallocationSymbols*, *oldAllocation*, *newAllocation*)

```
1: updateAllocationTable(reallocationSymbols, newAllocation)
2: if oldAllocation == localID then
3:   for clause in Wo do
4:     for symbol in ra-symbol(clause)∩ reallocationSymbols do
5:       sendto(clause,newAllocation)
6:       if ra-symbol(clause)\ reallocationSymbols == ∅ then
7:         delete(clause)
8:       end if
9:     end for
10:  end for
11: end if
12: if newAllocation == localID then
13:   Wo ← Wo∪ Receive()
14: end if
15: Wait()
```

requires talking about properties of the global clause sets that we define below.

Definition 43 (Global Clause Sets).
The global Wo set and Us set are the union of all the local clause sets of all reasoners m.

- $Wo_\cup = \bigcup_m Wo_m$
- $Us_\cup = \bigcup_m Us_m$
- $WU_\cup = Wo_\cup \cup Us_\cup$

According to Definition 42, there is a copy of an input clause for every a-symbol. Since only redundant clauses are deleted in the saturation, this property holds for any clause.

Corollary 8 (Relevant Symbol Invariant)**.** *For each non-redundant clause $c \in WU_\cup$ and each allocation symbol $S \in$ a-symbol(c) there exists a copy c_i of c with $S =$ ra-symbol$(c_i), c_i \in WU_\cup$.*

Worked off clauses are moved when the allocation of corresponding relevant allocation symbols changes. However, clauses with the same relevant allocation symbols are always moved together, hence they are always contained in the same *Wo* set.

Corollary 9 (Atomic Worked Off Sets)**.** *For every symbol S, Wo clauses c with* ra-symbol$(c) = S$ *are always contained in the Wo set of the reasoner responsible for S before a reallocation starts and after a reallocation is completed.*

Consequently, the global set of all *Wo* clauses is always saturated.

Lemma 1 (Worked Off Invariant)**.** *In distributed resolution with dynamic allocation, for every clause c with* $Wo_\cup \vdash_R c$*:*

- *either* $c \in WU_\cup$ *or*
- *c is redundant in* $WU_\cup$*.*

Proof. Assume that $Wo_\cup$ is not saturated, i.e., there are clauses $p, q \in Wo_\cup$ that are premises of an applicable resolution rule and the conclusion c is not redundant and not contained in $WU_\cup$. Then, there is a symbol S with $S \in (\text{a-symbol}(p) \cap \text{a-symbol}(q))$. According to Corollary 8 there are copies p', q' of p and q with $S = \text{ra-symbol}(p') = \text{ra-symbol}(q')$. Assume, without loss of generality, that p' was added to $Wo_\cup$ before q', and q' was picked as given clause and added to $Wo_\cup$ by reasoner m. Hence, m was responsible for symbol S at that moment. Consequently, p' was contained in the *Wo* set of reasoner m according to Corollary 9. Hence, q' would have been resolved with p' to obtain c. Clause c is only deleted if it is redundant, otherwise $c \in WU_\cup$. □

With this preparation, we can now prove the completeness of distributed resolution with dynamic allocation.

Proof of Theorem 8. Like in previous chapters, termination and correctness are not affected by extending distributed resolution. If the underlying calculus is correct and terminates, the same holds for the distributed dynamic algorithm. Of course, termination is not guaranteed for full first order logic, only for the decidable subsets $\mathcal{ALCHIQ}$.
Completeness is more difficult to prove, it follows from Lemma 1. Assume the input clauses are unsatisfiable. When distributed resolution returns, all reasoners are locally saturated, i.e. the local *Us* sets are empty. Since the applied calculus R is complete, we have $Wo_\cup \vdash_R \square$. Furthermore, the empty clause is never redundant. Consequently, $\square \in WU_\cup$ according to Lemma 1. Because $Us_\cup$ is empty, this implies $\square \in Wo_\cup$. Hence, every contradiction is detected by distributed resolution. □

Hence, the allocation of symbols can be changed at any time during the distributed reasoning process.

9.3 Subtask Coordination

The reallocation method described in the previous sections requires to halt the whole system for updating the allocation. Local reasoning continues only after all reasoners are finished with the reallocation task. For improving reallocation, we divide a reallocation task in a couple of subtasks, namely

updating the local symbol allocation table of every reasoner and propagating the corresponding reallocated clauses. Then we analyze the dependencies between subtasks, to obtain an appropriate order of the subtasks that avoids flaws, idle times and unnecessary network traffic.

R_I update: The reasoner specified in *newAllocation*, usually an idle reasoner, updates the local allocation table.

R_B update: The busy reasoner *oldAllocation* updates the local allocation table for delegating work to R_I.

R_o update: The other reasoners update the allocation tables in arbitrary order.

R_B send: R_B sends reallocated clauses to R_I.

R_I receive: R_I receives and inserts clauses into local clause set.

The simple implementation of reallocation is depicted in Table 9.1. First all reasoners are stopped, then the updates and propagations are performed, finally the reasoners continue reasoning. Obviously, the propagated clauses cannot be received before they are send and a reasoner has to update the allocation table before sending or receiving reallocated clauses.

R_I stop	R_I update	R_I receive	R_I continue
R_B stop	R_B update | R_B send		R_B continue
R_o stop	R_o update		R_o continue

Table 9.1: Partial order of subtasks for simple reallocation.

Starting from the simple reallocation implementation, we will find a more efficient method by defining an appropriate partial order of the subtasks and relaxing the restrictions on reasoner downtime. I.e. we stop the reasoners as late as possible and continue as soon as it is safe to do so. After analyzing the dependencies between subtasks, we can relax the restrictions of the simple reallocation method.

Lazy usable reallocation We do not need to reallocate *Us* clauses before restarting reasoners because they can be propagated when they are picked as given clause. The allocation may change again in the meantime, it is sufficient to decide propagation based on the relevant allocation symbols and current allocation when the given clause is processed. In the following, with 'R_B send' and 'R_I receive' we denote sending and receiving of *Wo* clauses.

R_I stop	R_I update R_B stop	R_B update	R_B send R_o update	R_I receive R_B continue	R_I continue

Table 9.2: Partial order of subtasks for efficient reallocation.

R_I update $\prec$ R_o update, R_B update $\prec$ R_o update Once the allocation tables of R_I and R_B are updated, delays in updating the other reasoners do not cause problems: If some reasoner R_{some} still uses an out-of-date allocation, it will send a clause that should go to R_I to R_B instead. But, R_B will just forward this clause to R_I, so apart from a short detour in the clause propagation, outdated partition tables of other reasoners do not cause problems. Hence, if we make sure the other reasoners are updated later, they do not have to stop reasoning at all.

R_I update $\prec$ R_B update For R_I and R_B we have to take care about the order of updates: Assume R_B is updated, but R_I not yet. In this case, R_B allocates clauses to R_I that R_I allocates R_B. Hence, as long as the update of R_I is delayed, clauses may be send back and forth between the two reasoners. If we make sure R_I is updated first, R_B may only delay sending some clauses until it is updated.

R_B update $\prec$ R_I send On the one hand, we would like to keep the S-clauses at R_B until the allocation is updated at R_B for resolving given clauses with the same top symbol. On the other hand, for the same reason we would like to have the S-clauses at R_I by the time the allocation of R_I is updated. We decide for the first option for two reasons: First, R_B is busy and likely to pick a given clause with modified allocation while R_I will more probably not do anything until the S-clauses are received. Second, other reasoners are updated later and still send S-clauses to R_B.

The partial order given by these considerations is depicted in Table 9.2. Reasoners that do not send or receive reallocated clauses do not have to wait at all, they just update the local allocation table. R_B does not need to stop either, if it sends the reallocated *Wo* clauses right after updating the local allocation table. Only reasoner R_I stops after updating. In most cases, R_I is idle anyway, hence it is no problem to wait for the reallocated clauses before going on.
Note that we could even continue R_I directly after updating the table if we record allocation changes. To avoid missing inferences in R_I, we have to skip given clauses that are reallocated clauses and put them back into the *Us* set until the *Wo* set is received.

We assume, reallocation tasks do not overlap, i.e. every subtask is completed before the first subtask of the next reallocation task starts. In theory, a much weaker requirement is sufficient to guarantee flawless dynamic allocation. We only have to make sure the reallocation symbol sets of every pair of overlapping reallocation tasks are disjoint. But, the communication between reasoners is more complicated when reallocation tasks overlap.

Now we have decided for an order of the subtasks, we have to synchronize the reasoner processes accordingly. Assume, a reallocation task (sa, sa', R_B, R_I) is to be executed with reallocated Symbols S. For every consecutive pair of subtasks that is performed by different reasoners, we have to send a message from the reasoner that performs the first subtask to the reasoner that performs the next subtask to ensure correct order of subtasks. The final process structure is depicted in Figure 9.1.

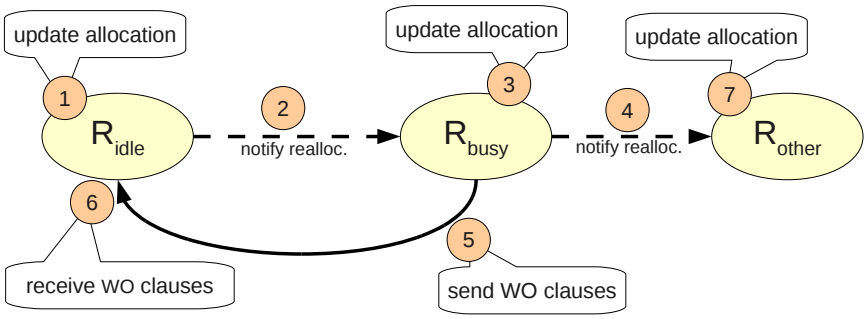

Figure 9.1: The process of reallocating a set of symbols S to an idle reasoner.

Note that up to subtask (6), the idle reasoner usually stays saturated. If it happens to receive propagated clauses from other reasoners and continues reasoning, the idle reasoner should not pick given clauses that are S-clauses because they have to be resolved with the *Wo* S-clauses that did not arrive yet. Hence, in this case the selection of given clauses has to be adapted until the *Wo* S-clauses are received.

9.4 Deciding About Reallocation

The idea of dynamic allocation is improving the allocation of symbols at runtime. We showed that this is possible in theory in the previous section, but, changing the allocation may also result in a less appropriate partitioning. The benefits from dynamic reallocation depend on the right choice of reallocation tasks. First, we have to decide which are the reasoners R_B and R_I that should rebalance their workload. Then, we have to select the reallocation symbols from the symbols R_B is currently responsible for.

9.4.1 Choose Reasoners

Obviously, we are searching for a reasoner R_B with heavy workload and a reasoner R_I that is idle or has very low workload. Relevant measurement for determining the workload is, first of all, the current number of *Us* clauses. If the *Us* set is small, the reasoner R_B will be idle very soon, and the effort for the reallocation would not pay off. All *Us* clauses have to be picked as given clauses and processed. But, when reallocations have been performed already, not every given clause is resolved with the *Wo* clauses. Some *Us* have been reallocated, i.e. the allocation changed since the clause was derived and it is propagated to the reasoner that is now responsible for the given clause.
For a more accurate estimation of workload than the number of *Us* clauses alone, we can take into account increase or decrease of the number of *Us* clauses per second. This measurement is additionally influenced by the number of *Wo* clauses, because, if there are many potential partners for a given clause, then it takes longer to process the given clause.
Wo clauses do not only affect workload but also the overhead of reallocation: Every propagated *Wo* clause is send and received and inserted into another *Wo* set.

Decision Communication

The decision, which reasoners adjust their workload depends on the situation of multiple reasoners. Furthermore, even with full information, the reasoners cannot decide independently, they have to agree on the reallocation to be performed. We propose different strategies for the decision process and address the advantages and disadvantages.

Local Communication One option is to decide about reallocation locally, by negotiation between two reasoners. When a reasoner has very high workload, it sends a reallocation request to a neighbor. The neighbor either rejects or replies with an "agree" message. If the neighbor has too much workload, it just ignores the request. After a specified timeout, the request expires, i.e. it is resent if the first reasoner is still busy. Note that a busy reasoner may send only one request at a time to avoid overlapping reallocations.
This method can be modified in various ways. Instead of repeating a request when the status of the busy reasoner remains the same, it could send an explicit "expire" message. This reduces the number of requests, but it takes longer to find an idle neighbor because the expiration and hence the request to the second neighbor may take longer.
Alternatively, the busy reasoner can send a third message to confirm the reallocation. Then, multiple requests may be pending at the same time. If there are two positive replies, the busy reasoner sends only one confirmation.

This speeds up the detection of possible reallocations, but it also increases network traffic. Another alternative is swapping busy and idle reasoner. In this variant, the idle reasoner sends a request (i.e. reallocation proposal) and the neighbors may accept or deny.
All local communication methods require a high number of messages between reasoners. In general, many reasoners have a high workload and will send a lot of requests to neighboring reasoners. Hence it is necessary to restrict R_B to reasoners with workload above some limit, e.g., defined by the number of *Us* clauses. Furthermore, the frequency of request can be limited. After a negative reply, R_B should wait for a specified period before asking the same reasoner again.

Central Communication For reducing the communication in the ontology network, a central control node can be appointed to decide about reallocation. Then, only the control node is notified about the current workload of every reasoner and decides if a reallocation is necessary. The control node chooses a reallocation task and notifies the reasoner R_I which in turn notifies R_B. With this communication design, only one message is send for every change of reasoner status and no repetition of the message is necessary. However, there is also a drawback to the centralized design. The control node imposes a communication bottleneck, all information is gathered there. Hence, in very large networks this can slow down the process.

9.4.2 Choose Symbols

The optimal choice of reallocation symbols mainly depends on the workload caused by each symbol. Furthermore, symbols that are connected to symbols allocated to R_I in the derivation graph should be preferred for reallocation. One option is to use the current partitioning of the of current derivation graph as initial partitioning for an incremental graph partitioning algorithm like the balanced edge cut discussed in Chapter 8. But, the experiments in Section 8.6 showed it is often enough to consider node weights. Depending on the difference in workload of the reasoners R_B and R_I, the total weight sum of the reallocated symbols is decided. A reasonable choice is to reallocate symbols with total weight w_r that corresponds to half of the workload difference.

$$w_r = w_B \cdot \frac{load_B - load_I}{2 \cdot load_B}$$

where w_B is the total weight sum of reasoner R_B and *load* is the workload of R_B and R_I respectively.

While the resulting reallocation task is probably a good choice, the effort for selecting the reallocation symbols is high. Apart from counting deriva-

tions and deletions for each symbol, we have to implement a communication protocol for notifying all reasoners about the choice of reallocation symbols. The communication is much simpler, when reallocation does not depend on symbol weights, but only on the previous allocation. In fact, it is possible to define a reallocation function that can be computed independently in every reasoner. I.e., the reallocation function computes a set of reallocation symbols for a given pair of reasoners.

Definition 44 (Simple Reallocation Function).
A simple reallocation function maps a tuple (sa, R_B, R_I) *of an allocation function* sa *and two reasoners* R_B, R_I *to a set* RS *of symbols.*

Before we define a simple reallocation function, we give an allocation function that is used as initial allocation.

Definition 45 (Chunk Allocation).
For a chunk allocation sa*, the symbols are numbered according to the precedence, the first symbol in the precedence is* S_0*. The symbol allocation* sa *is computed from the index number of the symbol.*

$$sa(S_i) = i\%C/step$$

where m *is the number of reasoners,* $C \geq m$ *is a constant and* $step = C/m$*. A set of symbols with the same value of* $i\%C$ *is called a chunk.*

Hence, there are C chunks corresponding to the C possible remainders of the division i/C. Without dynamic reallocation, the best value for C is the number m of available reasoners. In this case, chunk allocation is round robin allocation ($sa(S_i) = i\%C$) which leads to the best balance of workload when the precedence is based on symbol frequency. However, for enabling reallocation, C is set to a higher number, e.g. $4m$.
For simple reallocation, we propose the clock reallocation method. It is illustrated by arranging the symbols S_i in a circle. The circle consists of C chunks divided into m sections, each section is allocated to one reasoner. Now, the reallocation corresponds to moving a section border clockwise or counterclockwise as depicted in Figure 9.2. One chunk of the local symbols of reasoner R_B is reallocated to the idle reasoner R_I. In the first picture, chunk 11 is reallocated clockwise from reasoner R_0 to R_4. The second picture shows clockwise reallocation from R_2 to R_3.
The precise definition of clock reallocation is given below.

Definition 46 (Clock Reallocation).
Clock reallocation maps a tuple (sa, R_B, R_I) *of an allocation function* sa *and two reasoners* R_B, R_I *to a reallocation task* (sa, sa', R_B, R_I) *by defining the reallocated symbols* RS*:*

$$RS = \{S_i \mid sa(S_i) = R_B, sa(S_{i+1}) = R_I\}$$

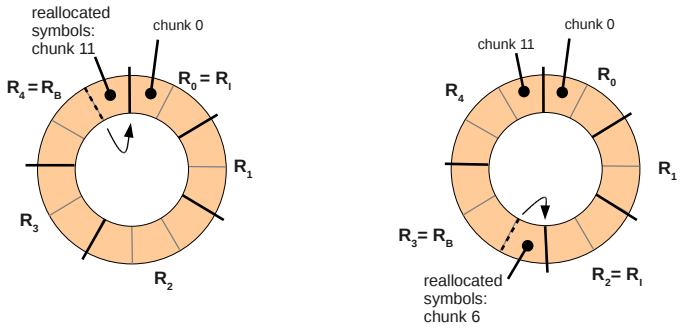

Figure 9.2: Clock reallocation with 12 chunks and 5 reasoners, clockwise and counterclockwise.

The new allocation sa' is defined based on the reallocated Symbols:

$$sa'(S) = \begin{cases} R_I & \text{for } S \in RS \\ sa(S) & \text{otherwise} \end{cases}$$

In general, the partitioning of symbols obtained by clock reallocation is not optimal, but the reallocation can be computed very efficiently.

9.5 Experiments

We tested dynamic distributed resolution by simulating a change in the availability of reasoners. The test data is the NCI ontology. All tests were executed on the same node of the Freiburg cluster of the bwGrid[6], using 2,83GHz CPUs with 16G memory.
First, the saturation starts with 4 reasoners and a chunk allocation with $C = 8$ chunks allocated to two reasoners. By using only half of the reasoners at the beginning, a increase in the number of available reasoners from 2 to 4 is simulated. Some time after the start of the saturation, reallocations are performed using different methods. For comparison, we first saturated the ontology without reallocation to obtain the baseline results depicted in the first line of Table 9.3. Tests with 4 reasoners, where only 2 are busy and the standard setting for 2 reasoners had similar results. The runtime values are larger than in Section 5.4, because the Esslingen cluster of the bwGrid was used in that section.
On first reallocation tests (not reported in the table), we observed that the reallocation of *Wo* clauses took several seconds for every reallocation task. This was caused by a very inefficient implementation of the function that deletes *Wo* clauses. Without distribution, this function is rarely called, hence it was not optimized for SPASS. For a simple solution to the prob-

setting	reallocations		t/sec	# deriv.	# prop.
baseline	-	-	41.34	123439	55983
manual	3 sec	0→3,1→2	34.87	123439	99778
manual	32 sec	0→3,1→2	37.58	123439	82814
auto	13 sec	0→3	33.80	123439	79494
auto	13 sec	0→3 ∗2	32.48	125354	91354
auto	13 sec	0→3 ∗3	33.32	130994	92107
auto	13 sec	0→3 ∗4	34.19	141468	91256

Table 9.3: Evaluation of NCI saturation for different dynamic settings with 2+2 reasoners. Performance depends on the time of the first reallocation, number of reallocations and concrete reallocation tasks.

lem, the deletion of propagated *Wo* clauses is skipped. This induces some redundant derivations, nevertheless the saturation is faster.
For the manual reallocation setting depicted in lines 2 and 3 of Table 9.3, the reallocation tasks (i.e. reasoners R_B and R_I) are hard coded. One chunk is reallocated from R_0 to R_3 and one chunk from R_1 to R_2. These reallocations are the straight forward choice for using the two additional reasoners if we do not consider workload. The second column in Table 9.3 shows the time the reallocations were performed. For early reallocation 3 seconds after start, the runtime was reduced by 6.5 seconds. For late reallocation, the runtime decrease is still 3.8 seconds compared to the baseline.
The number of propagations is higher for early reallocation. This is not surprising because early reallocation is more similar to static allocation to four reasoners, while late reallocation is more similar to static allocation to two reasoners. The number of propagations increases with the number of reasoners.
The automatic reallocation setting uses centralized communication for the reallocation decision. The reasoners message to the control node the current number of *Us* clauses, whenever the change in the number exceeds 300. A specified amount of time after the saturation start, the control node chooses the best reallocation task and decides if it is performed. I.e., of all reasoners that have an idle neighbor, R_B is the reasoner with the largest set of *Us* clauses. R_I is the idle neighbor of R_B. If the number of *Us* clauses of R_B is at least 500, the reallocation task is started. The total number of reallocations is limited by a hard coded value of 1 to 4.
For all tests with automatic reallocation, the first reallocation was performed about 13 seconds after start of the saturation. Depending on the number of performed reallocations, the runtime and number of derivations and propagation vary. If the number of reallocations is limited to one, the runtime is already shorter than for both manual allocation settings. This implies the reallocation 1→2 causes more overhead than improving the allocation. Also

with two reallocations, the automatic setting has a shorter runtime than the hard coded reallocation. Hence, the method for deciding about reallocation performed well. However, we had to set a limit for the number of allocations. Without this restriction, more than 20 reallocation are performed and the runtime increases to more than 70 seconds. The limit on *Us* clauses only avoids reallocations close to the end of the saturation.
If the limit is increased from 2 to 3 and 4 reallocations, the runtime increases by about a second for each reallocation. The best result is obtained for 2 automatic reallocations.
If there is more than one reallocation between the same reasoners, the number of derivations increases, because the propagated *Wo* clauses are not deleted. For 4 reallocations, the obtained partitioning was less balanced than for 2 reallocations and for 3 reallocations. On the one hand, bad balance can decrease the number of propagations, as we have already seen in Section 8.6. On the other hand, reallocations increase the number of propagations. This explains the non monotonic characteristics of propagation observed for automatic reallocation.

In summary, the results show that dynamic allocation is possible and can decrease runtime. More investigations are necessary for developing better strategies for the reallocation decision based on more detailed information on the gradient of the number of *Us* clauses and the workload connected to specific symbols.

Part IV

Conclusion

Chapter 10

Future Work

In this chapter we discuss additional options for improving distributed resolution. In particular, we propose allocating a symbol to multiple reasoners for improving the balance of workload. Furthermore, the supported expressivity can be extended and there are many options for improving the performance of the distributed reasoning system. Finally, investigations on the general applicability of distributed resolution would facilitate adapting distributed resolution to other rule based calculi.

10.1 Subdivided Symbols

We have only one restriction imposed on allocation of clauses: It is based on an allocation of symbols with each symbol allocated to exactly one reasoner. On the first sight this is no severe restriction, but there are cases where we want to avoid this restriction. Some symbols, like for example the "part-of" property can cause very high workload. If a large amount of the derivations are clauses with the same top symbol, the distribution may be imbalanced even if one of the reasoners is only responsible for the problematic symbol. For these cases, subdividing responsibility for one symbol to multiple reasoners improves balance and thereby may decrease runtime.
Although the simplicity of our approach is based on allocating a symbol to only one reasoner, the approach can be extended to use multiple reasoners for resolving clauses with the same top symbol. To illustrate the idea of this extension, we first consider a situation with three reasoners used for the same symbol S. We assume an allocation function that allocates some clauses with top symbol S to $R1$ and the others to $R2$. A suitable allocation of S-clauses can be defined, for example, based on the top symbol of the next literal.
The saturation process for a symbol S allocated to multiple reasoners consists of two steps. First, the sets Wo_1 and Wo_2 are generated by two reasoners responsible for S. For this step, Algorithm 2 is used just like for reasoners

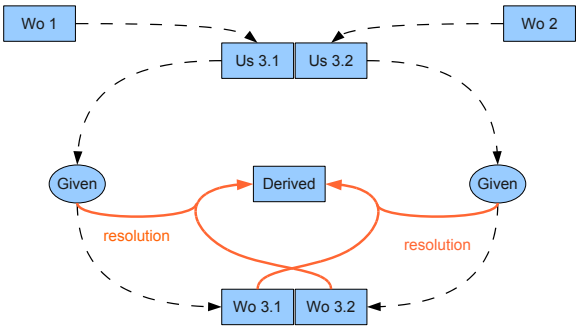

Figure 10.1: Saturating two worked-off clause sets with shared a-symbol.

that do not share responsibility for a symbol. Step 2 is the saturation of the set $Wo_1 \cup Wo_2$ without repeating the already performed inferences.
The modified saturation is depicted in Figure 10.1. Algorithm 4 specifies the corresponding procedure for n reasoners responsible for the same symbol. We use a third reasoner $R3$ for saturating the union, it is also possible to use $R1$ or $R2$ (the reasoner that is locally saturated first). $R3$ can start as soon as Wo clauses are available from $R1$ or $R2$. In difference to Algorithm 2, $R3$ maintains two distinct sets of Us clauses and two distinct sets of Wo clauses to avoid repeating inferences that where already performed by $R1$ or $R2$.
$R3$ picks a given clause from one of the usable sets, e.g. $Us_{3.1}$ that contains Wo clauses from $R1$. The given clause is resolved with partner clauses from $Wo_{3.2}$ (i.e. Wo clauses from $R2$) and then added to $Wo_{3.1}$. All derived clauses are propagated by $R3$, the allocation function allocates no clause to $R3$. In difference to Algorithm 2, two clauses from $Us_{3.1}$ are never resolved with each other. Note that factoring is only applied by $R1$ and $R2$, but not $R3$ because all clauses derivable by factoring are already contained in the Us sets of $R3$.
The method could be extended to also use multiple reasoners for the second step of the saturation. Adding reduction to the Algorithm 4 is less important than in Algorithm 2 since the input is partially saturated already.

Hence, with a small modification to the local reasoning algorithm, it is possible to distribute the workload caused by a single symbol to multiple reasoners.

10.2 Expressivity

The ordered resolution calculi presented in Chapter 5 and Chapter 7 are complete for first order logic but are not distributed efficiently for every decidable subset of first order logic. We addressed several decidable logics

Algorithm 4 Distributed Resolution Prover 2

isSatisfiable(KB)

1: $Wo \leftarrow \{\emptyset, \emptyset, ...\}$
2: $Us \leftarrow KB$
3: **while** $Us_i \neq \emptyset$ **for some** i **do**
4: $Given \leftarrow$ choose(Us$_i$)
5: $Us_i \leftarrow Us_i \setminus \{Given\}$
6: $Wo_i \leftarrow Wo_i \cup \{Given\}$
7: $New \leftarrow \emptyset$
8: **for** $j = 1$ **to** n **do**
9: **if** $j \neq i$ **then**
10: $New \leftarrow New \cup$ resolve(Given, Wo_j)
11: **end if**
12: **end for**
13: **for** *clause* **in** New **do**
14: **for** *reasonerID* **in** a(*clause*) **do**
15: send(*clause*,*reasonerID*)
16: **end for**
17: **end for**
18: $Us \leftarrow Us \cup$ receive()
19: **if** $\square \in Us$ **then**
20: **for** *reasonerID* $\in$ M **do**
21: send($\square$, *reasonerID*)
22: **end for**
23: **return** false
24: **end if**
25: **if** $Us == \emptyset$ **and** globally saturated **then**
26: **return** true
27: **end if**
28: **end while**

that are relevant for knowledge representation, but for deciding satisfiability of any OWL-DL ontology, additional work is necessary.

Equalities While the theoretical investigation on distributed resolution for the description logic $\mathcal{ALCHIQ}^-$ is based on basic superposition, our experiments were executed with the less efficient superposition. For larger ontologies an extension of the implementation is necessary. A simple extension would be to add a reduction rule that removes clauses that are not on the list of clause types in Table 7.1. For example, we can remove all clauses that contain a literal with three nested function symbols. However, the best solution is a complete implementation of basic superposition.

Transitivity Transitive properties like "has-part" are required by many ontologies. In Chapter 6 we proposed a distributed resolution calculus for BSHE that enables efficient reasoning with transitive properties. However, BSHE does not contain function symbols, hence the corresponding description logic subset does not allow existential restrictions. For applying distributed resolution to ontologies that contain transitive properties and existential restrictions, we have to extend the calculus.

Datatypes Datatypes in ontologies allow expressing that e.g. an adult is a person with an age of at least 18. In theory, datatypes do not add expressivity, as they can be translated to the abstract domain. But, *hasValue* restrictions are used frequently on datatype properties and translate to nominals. Our approach can be extended to datatypes with the approximation of nominals described below. In practice however, reasoning on the translated datatypes is not very efficient. The better choice is modifying the reasoner to deal with datatypes directly.

Nominals Nominals ($\mathcal{O}$), i.e. references to instances in axioms describing classes, are a known source of scalability problems but not used in many ontologies. Nominals can be approximated in the description logic $\mathcal{ALC}$ by common classes. For example the axiom $Photoionization \sqsubseteq \exists hasStar.Sun$ can be replaced by the axioms $Photoionization \sqsubseteq \exists hasStar.Sun\_Class$ and $Sun\_Class(Sun)$, with a new class Sun_Class. The information that Sun must be the only instance of Sun_Class is missing in this approximation. Actually, the clause $\neg Photoionization(x) \vee hasStar(x, Sun)$ is the correct translation of the nominal axiom. However, this is an additional clause type (type $\mathcal{O}1$) not included in Table 5.1 or Table 7.1. Adding this clause type to the $\mathcal{ALC}$ clauses introduces another new clause type $\mathcal{O}2$: $\mathbf{P_1}(x) \vee \mathbf{P_2}(c)$ (obtained from resolution with type 3). Unfortunately, literals in this clause are not completely ordered, two literals ($P_1(x)$ and $P_2(c)$) are maximal and none is selected. Furthermore, clauses derived from this type

can have arbitrary numbers of different Variables. Hence, the calculus is not guaranteed to terminate because it may generate an infinite number of clauses. We cannot extend our distributed resolution methods to nominals that easily. Adding the nominal clauses to $\mathcal{ALCHIQ}$ raises even more clause types with multiple resolvable literals. Hence, other methods are necessary for dealing with nominals in distributed resolution.

10.3 Performance

There are many possibilities for improving the performance of distributed resolution. First of all, modified data structures and additional indexes would speed up retrieval and deletion of clauses with given top symbol. In our experiments, the whole *Wo* clauses of a reasoner were scanned for potential partner clauses of a given clause. If a separate list of *Wo* clauses is maintained for every top symbol, scanning *Wo* clauses that are no possible partners of the current given clause would be completely avoided. Furthermore, this improvement would speed up dynamic allocation because it enables selecting the *Wo* clauses for propagation without scanning through all *Wo* clauses.

Propagation and Parsing Another working point for performance of propagation is parsing and serialization of clauses. In our experiments, propagated clauses are written to a string and parsed like input clauses. Parsing is more efficient then the original parser because the symbol index number is written instead of the symbol name, but still optimization is possible. The propagation can be realized more efficiently by sending a clause object directly, then the clause structure is received with the clause and parsing can be skipped.

Assertion Summary Although distributed reasoning is applicable to ontologies with instances (i.e. constants), it is not designed for large amounts of assertional data. Especially, in combination with general concept inclusions, large amounts of data will slow down the reasoning process because saturation materializes implied assertions. For increasing the performance for this type of ontologies, distributed resolution should be extended with approaches for efficient assertional reasoning. For example, [22] propose a method that replaces groups of similar instances by representatives. Thereby, the assertional data is reduced and the reasoning is faster. In the result, the representatives are replaced again by the original instances.

Shared Memory Furthermore, the performance would be improved by taking benefit of available shared memory. If enough shared memory is available for two reasoners, the communication between these reasoners does

not require TCP connections. The clauses can be stored in shared memory, available to both reasoners.

Caching Our focus on satisfiability checking is motivated by the fact that all relevant queries can be reduced to satisfiability. However, for some queries the direct reduction to satisfiability is not very efficient. In theory, the concept hierarchy of an ontology is computed by a series of subsumption test that in turn correspond to satisfiability checks. But, testing every pair of concepts for subsumption is not the most efficient method, it is possible to combine some tests and thereby reduce the number of tests. Also, the order of the subsumption tests matters, because depending on the result of a subsumption test, other test may be skipped. Hence, more investigations of the saturation process are necessary for designing a classification method based on distributed resolution. Apart from classification, there are other queries that require special optimizations, like conjunctive queries.
Furthermore, we have to consider the repeated execution of similar queries. Even when standard translation to a satisfiability check is appropriate, caching techniques should be added to improve query runtime. The simplest caching method is to store a saturated version of the input ontology. Then, only the query clauses have to be saturated, most of the work is done off line. However, additional methods are necessary for removing again the query and its implications before starting the next query. Otherwise, the saturated ontology has to be loaded for every query.

10.4 Generalization

We showed distributed resolution is applicable to a couple of different calculi. This raises the question, which properties of a calculus determine applicability for distribution and if it would be possible to automatize distribution. In particular, we would like to automatically define the a-symbol function and automatically modify rules of the calculus if necessary. Then, given a calculus like the calculus for $\mathcal{EL}+$ classification proposed by [5], we could automatically compute a distributed variant of the calculus.
Automatic distribution requires to first give a general formal description of calculi that takes into account the required connections between premises. The first step is to investigate the properties of calculi that are necessary and sufficient for efficient distribution.

Chapter 11

Summary

This work investigates a method for distributed reasoning on ontologies. The increasing size of ontologies used for knowledge representation requires using multiple processors for computation of a reasoning task. Hence, reasoning methods are necessary that divide the computation into several processes that can be executed in parallel.
We proposed a distributed reasoning method that checks satisfiability of different subsets of first order logic, with a focus on subsets of OWL-DL. The method is sound and complete and for decidable fragments it terminates. The distribution of the reasoning process is based on a distribution of the ontology axioms. Distribution reduced the runtime of satisfiability tasks considerably. In some settings, duplicating the number of reasoners reduced runtime by more than 50%. Theoretical investigation and experiments answered the research questions posed in the introduction:

Q1 Which reasoning method is a good basis for distributed reasoning on description logic ontologies?

Resolution is a good basis for distributed reasoning, it allows dividing the input into subsets of axioms that can be processed largely independently. Actually, distribution is almost inherent in resolution. A given clause is only resolved with a subset of the worked off clauses and these subsets can be separated. In contrast, tableaux reasoning, which is one of the most popular methods for description logic reasoners, is not that compatible with distribution.

Q2 Is it possible to preserve soundness, completeness and termination of this reasoning method when distributing computation to a set of parallel processes?

In Chapter 5 we identified the allocation symbols of a clause that are the connection between premises of a resolution inference. The allocation symbols determine whether two clauses can be resolved with each other and

guide our allocation method. Any partitioning of the ontology signature defines an allocation of clauses to reasoners. We proved that for all these allocations, soundness, completeness and termination of different resolution calculi are preserved by the proposed distributed resolution method.

Q3 Is the distribution efficient, the runtime decreased?

The theoretical properties are essential, but for applications the viability of distributed resolution depends on the actual decrease in runtime that can be achieved. We tested the method on different ontologies and observed a runtime decrease that exceeded our expectations. In some settings, doubling the number of reasoners decreased the runtime by more than 50 %.

Q4 Does the distributed reasoning method scale?

For investigating scalability of the approach, we recorded the maximum number of reasoners that can be successfully used for a reasoning task. This number increased with the size of the ontology and runtime of the saturation.

Q5 What is the expressivity that can be supported by distributed reasoning?

Distributed resolution can be applied to full first order logic, but then the representation may contain multiple copies of each clause because the number of allocation symbols is not limited. Consequently, the number of propagations may be very high. We showed that efficient distribution is possible for the description logics $\mathcal{ALCHI}$ and $\mathcal{ALCHIQ}$ and the BSHE class.

Q6 What is the best method for computing a distribution of input axioms?

We tested different methods for computing a partitioning of ontology symbols that is the basis for the clause allocation of distributed resolution. The results in Chapter 8 showed that simple partitioning strategies based on only the frequency of symbols performed good enough. Even a round robin allocation may be acceptable. More sophisticated methods that take into account propagation must put a heavy emphasis on balance. Otherwise, reducing propagation is traded by high imbalance and results in bad performance of the partitioning for distributed resolution.

Q7 Is it possible to change the distribution of axioms at runtime?

The number of processors that are available may change at runtime, if a reasoning task on a large ontology takes several hours. For this setting, we propose runtime modification of the clause allocation. The investigations in Chapter 9 show that soundness, completeness and termination are preserved, if the subtasks of the allocation change are executed in a specific order. Experiments showed that dynamic allocation can reduce runtime of distributed resolution.

Q8 What optimizations of the method are necessary and/or possible?

The first optimization we propose aims at improving balance of workload. In Section 10.1 we proved that subdividing symbols for allocation is possible and does not cause duplication of inferences. Also important for applications of distributed resolution is built-in support for reasoning on datatypes and the combination of transitivity and equalities. A simple but effective performance enhancement can be achieved by adapting the representation of *Wo* clauses. Separation into disjoint sets corresponding to the relevant allocation symbols would speed up local reasoning and propagation.

To sum up, distributed resolution is a promising method for scaling up reasoning on large ontologies. Experiments proved the concept feasible. Additional work is necessary for investigation and implementation of extensions and optimizations.

Bibliography

[1] Philippe Adjiman, Philippe Chatalic, Francois Goasdoué, Marie-Christine Rousset, and Laurent Simon. Distributed reasoning in a peer-to-peer setting: Application to the semantic web. *Journal of Artificial Intelligence Research*, 25:269–314, 2006.

[2] Gene Myron Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

[3] Eyal Amir and Sheila McIlraith. Partition-based logical reasoning for first-order and propositional theories. *Artificial Intelligence*, 162(1-2):49–88, 2005.

[4] Grigoris Antaniou and Frank van Harmelen. Web Ontology Language: OWL. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, pages 67–92. Springer, 2009.

[5] Franz Baader, Carsten Lutz, and Boontawee Suntisrivaraporn. Cel—a polynomial-time reasoner for life science ontologies. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR'06)*, pages 287–291. Springer, 2006.

[6] Franz Baader, Carsten Lutz, and Boontawee Suntisrivaraporn. Efficient reasoning in $\mathcal{EL}^+$. In *Proceedings of the 2006 International Workshop on Description Logics (DL2006)*, CEUR-WS, 2006.

[7] Franz Baader and Werner Nutt. Basic description logics. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors, *The Description Logic Handbook*, pages 43–95. Cambridge University Press, 2003.

[8] Leo Bachmair and Harald Ganzinger. Ordered chaining calculi for first-order theories of transitive relations. *J. ACM*, 45:1007–1049, November 1998.

[9] Leo Bachmair, Harald Ganzinger, Christopher Lynch, and Wayne Snyder. Basic paramodulation. *Information and Computation*, 121(2):172–192, 1995.

[10] Vladimir Batagelj. Analysis of large networks - islands. Presented at Dagstuhl seminar 03361, August 2003.

[11] Maria Paola Bonacina. The clause-diffusion theorem prover peers-mcd (system description). In *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249 of *Lecture Notes In Computer Science*, pages 53–56, London, UK, 1997. Springer.

[12] Maria Paola Bonacina. A taxonomy of parallel strategies for deduction. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):223–257, 2000. Published in February 2001.

[13] Maria Paola Bonacina and Jieh Hsiang. Parallelization of deduction strategies: An analytical study. *J. Autom. Reasoning*, 13(1):1–33, 1994.

[14] Alex Borgida and Luciano Serafini. Distributed description logics: Assimilating information from peer sources. *Journal of Data Semantics*, 1:153–184, 2003.

[15] Susan E. Conry, Douglas J. MacIntosh, and Robert A. Meyer. Dares: A distributed automated reasoning system. In *Proceedings of AAAI-90*, pages 78–85, 1990.

[16] Bernardo Cuenca Grau, Ian Horrocks, Yevgeny Kazakov, and Ulrike Sattler. Modular reuse of ontologies: Theory and practice. *Journal of Artificial Intelligence Research (JAIR)*, 31:273–318, 2008.

[17] Bernardo Cuenca Grau, Bijan Parsia, and Evren Sirin. Combining owl ontologies using e-connections. *Journal Of Web Semantics*, 4(1), 2005.

[18] Mathieu d'Aquin d'Aquin d'Aquin d'Aquin, Anne Schlicht, Heiner Stuckenschmidt, and Marta Sabou. Ontology modularization for knowledge selection: Experiments and evaluations. In *18th International Conference on Database and Expert Systems Applications (DEXA)*, 2007.

[19] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] Francesco Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. Reasoning in description logics. In G. Brewka, editor, *Principles of Knowledge Representation and Reasoning*, Studies in Logic, Language and Information, pages 193–238. CLSI Publications, 1996.

[21] Charles M. Fiduccia and Robert Marcel Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference DAC '82*, 1982.

[22] Achille Fokoue, Aaron Kershenbaum, Li Ma, Edith Schonberg, and Kavitha Srinivas. The summary abox: Cutting ontologies down to size. In *Proceedings of ISWC-06*, pages 343–356. Springer, 2006.

[23] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[24] Peter Haase and Yimin Wang. A decentralized infrastructure for query answering over distributed ontologies. In *Proceedings of The 22nd Annual ACM Symposium on Applied Computing (SAC)*, 2007.

[25] Bruce Hendrickson and Robert Leland. *The Chaco User's Guide, Version 2.0*, 1995.

[26] Bruce Hendrickson and Robert Leland. A multi-level algorithm for partitioning graphs. In *Supercomputing. Proceedings of the IEEE/ACM SC95 Conference*, 1995.

[27] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[28] Thomas Hofweber. Logic and ontology. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, fall 2011 edition, 2011.

[29] Ian Horrocks and Peter F. Patel-Schneider. Reducing owl entailment to description logic satisfiability. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):345—-357, 2004.

[30] Ullrich Hustadt, Boris Motik, and Ulrike Sattler. Reasoning in Description Logics by a Reduction to Disjunctive Datalog. *Journal of Automated Reasoning*, 39(3):351–384, 2007.

[31] George Karypis and Vipin Kumar. *METIS A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4.0*, 1998.

[32] Brian Wilson Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(1):291–307, 1970.

[33] Douglas B. Lenat. Cyc: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33, 1995.

[34] Ewing L. Lusk, William W. McCune, and John Slaney. Roo: A parallel theorem prover. In *Proceedings of the 11th CADE*, volume 607 of LNAI, pages 731–734. Springer, 1992.

[35] Carsten Lutz, Dirk Walther, and Frank Wolter. Conservative extensions in expressive description logics. In *Twentieth International Joint Conference on Artificial Intelligence IJCAI-07*, 2007.

[36] Boris Motik. *Reasoning in Description Logics using Resolution and Deductive Databases*. PhD thesis, Universität Karlsruhe (TH), Karlsruhe, Germany, January 2006.

[37] Raghava Mutharaju, Frederick Maier, and Pascal Hitzler. A MapReduce Algorithm for EL+. In *Workshop on Description Logics (DL2010)*, pages 464–474, 2010.

[38] Robert Nieuwenhuis and Albert Rubio. Theorem proving with ordering and equality constrained clauses. *Journal of Symbolic Computation*, 19:321–351, 1995.

[39] Philipp Nowakowski. Dire: Implementation of a distributed reasoner for description logic. Master's thesis, Universität Mannnheim, 2010.

[40] Eyal Oren, Spyros Kotoulas, George Anadiotis, Ronny Siebes, Annette ten Teije, and Frank van Harmelen. Marvin: A platform for large-scale analysis of semantic web data. In *Proceedings of the International Web Science Conference*, 2009.

[41] Ivo Popov. Graph partitioning for parallel computing. Technical report, University of Mannheim, 2008.

[42] Rob Raskin. Knowledge representation in the semantic web for earth and environmental terminology (SWEET), 2005.

[43] Alan Rector. The galen high level ontology. *Studies in health technology and informatics*, 1996.

[44] Nicholas Rescher. Leibniz's interpretation of his logical calculi. *The Journal of Symbolic Logic*, 19(1):1–13, 1954.

[45] Anne Schlicht and Heiner Stuckenschmidt. Distributed resolution for $\mathcal{ALC}$ - first results. In *ESWC Workshop on Advancing Reasoning on the Web*, 2008.

[46] Anne Schlicht and Heiner Stuckenschmidt. A flexible partitioning tool for large ontologies. In *International Conference on Web Intelligence and Intelligent Agent Technology (WI/IAT)*, 2008.

[47] Anne Schlicht and Heiner Stuckenschmidt. Towards distributed ontology reasoning for the web. In *International Conference on Web Intelligence and Intelligent Agent Technology (WI/IAT)*, 2008.

[48] Anne Schlicht and Heiner Stuckenschmidt. Distributed resolution for expressive ontology networks. In *Web reasoning and rule systems : Third International Conference, RR 2009*, 2009.

[49] Anne Schlicht and Heiner Stuckenschmidt. Peer-to-peer reasoning for interlinked ontologies. *International Journal of Semantic Computing, Special Issue on Web Scale Reasoning*, 4(1), March 2010.

[50] Anne Schlicht and Heiner Stuckenschmidt. Mapresolve. In *Proceedings of the Fifth International Conference on Web Reasoning and Rule Systems, RR 2011*, 2011.

[51] Julian Seidenberg and Alan Rector. Web ontology segmentation: Analysis, classification and use. In *Proceedings of the 15th international World Wide Web Conference*, Edinburgh, Scotland, 2006.

[52] Heiner Stuckenschmidt and Michel Klein. Structure-based partitioning of large concept hierarchies. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *Proceedings of the Third International Semantic Web Conference (ISWC 2004)*, pages 289–303, Hiroshima, Japan, November 2004.

[53] Heiner Stuckenschmidt and Maarten R. Menken. Tool Support for Dependency-Based Partitioning of OWL Ontologies. Technical report, Vrije Universiteit Amsterdam, 2005.

[54] Heiner Stuckenschmidt and Anne Schlicht. Structure-based partitioning of large ontologies. In *Modular Ontologies*. Springer, 2009.

[55] Peter R. Suaris and Gershon Kedem. An algorithm for quadrisection and its application to standard cell placement. *IEEE Transactions on Circuits and Systems*, 35(3):294–303, March 1988.

[56] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 697–706, New York, NY, USA, 2007. ACM.

[57] Martin Suda, Christoph Weidenbach, and Patrick Wischnewski. On the saturation of yago. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2010.

[58] Tanel Tammet. *Resolution methods for Decision Problems and Finite Model Building*. PhD thesis, Chalmers University of Technology and University of Göteborg, 1992.

[59] Herman J. ter Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):79–115, 2005.

[60] Dmitry Tsarkov, Alexandre Riazanov, Sean Bechhofer, and Ian Horrocks. Using vampire to reason with owl. In *International Semantic Web Conference (ISWC)*, 2004.

[61] Jacopo Urbani, Spyros Kotoulas, and Jason Maassen. OWL reasoning with WebPIE: calculating the closure of 100 billion triples. *The Semantic Web*, pages 213–227, 2010.

[62] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and F. van Harmelen. Scalable distributed reasoning using mapreduce. *The Semantic Web-ISWC 2009*, pages 634–649, 2009.

[63] Christoph Weidenbach. Combining superposition, sorts and splitting. In Alan Robinson und Andrei Voronkov, editor, *Handbook of Automated Reasoning*, volume II, chapter 27. Elsevier, 2001.

[64] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobalt, and Dalibor Topić. SPASS version 2.0. In Andrei Voronkov, editor, *Automated deduction - 18th International Conference on Automated Deduction*. Springer, 2002.