Inconsistent Ontology Diagnosis: Evaluation

Stefan Schlobach*, Zhisheng Huang*, and Ronald Cornet#
(*Vrije Universiteit Amsterdam, #AMC, Amsterdam)

Abstract.
EU-IST Integrated Project (IP) IST-2003-506826 SEKT
Deliverable D3.6.2(WP3.6)
In this document, we evaluated the framework for inconsistent ontology diagnosis and repair as introduced in Sekt Deliverable D3.6.1, where we defined a number of new non-standard reasoning services to explain inconsistences.
The evaluation is done in two ways, first, we study the effectiveness of our proposal in a *qualitative* way with some practical examples. Secondly, in a *quantitative and statistical* analysis, we try to get a better understanding of the computational properties of the debugging problem and our algorithms for solving it.

Keyword list: ontology management, inconsistency dignosis, ontology reasoning

Document Id.	SEKT/2005/D3.6.2/v1.0
Project	SEKT EU-IST-2003-506826
Date	January 26, 2006
Distribution	PU

SEKT Consortium

This document is part of a research project partially funded by the IST Programme of the Commission of the European Communities as project number IST-2003-506826.

British Telecommunications plc.
Orion 5/12, Adastral Park
Ipswich IP5 3RE
UK
Tel: +44 1473 609583, Fax: +44 1473 609832
Contact person: John Davies
E-mail: john.nj.davies@bt.com

Jozef Stefan Institute
Jamova 39
1000 Ljubljana
Slovenia
Tel: +386 1 4773 778, Fax: +386 1 4251 038
Contact person: Marko Grobelnik
E-mail: marko.grobelnik@ijs.si

University of Sheffield
Department of Computer Science
Regent Court, 211 Portobello St.
Sheffield S1 4DP
UK
Tel: +44 114 222 1891, Fax: +44 114 222 1810
Contact person: Hamish Cunningham
E-mail: hamish@dcs.shef.ac.uk

Intelligent Software Components S.A.
Pedro de Valdivia, 10
28006 Madrid
Spain
Tel: +34 913 349 797, Fax: +49 34 913 349 799
Contact person: Richard Benjamins
E-mail: rbenjamins@isoco.com

Ontoprise GmbH
Amalienbadstr. 36
76227 Karlsruhe
Germany
Tel: +49 721 50980912, Fax: +49 721 50980911
Contact person: Hans-Peter Schnurr
E-mail: schnurr@ontoprise.de

Vrije Universiteit Amsterdam (VUA)
Department of Computer Sciences
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
Tel: +31 20 444 7731, Fax: +31 84 221 4294
Contact person: Frank van Harmelen
E-mail: frank.van.harmelen@cs.vu.nl

Empolis GmbH
Europaallee 10
67657 Kaiserslautern
Germany
Tel: +49 631 303 5540, Fax: +49 631 303 5507
Contact person: Ralph Traphöner
E-mail: ralph.traphoener@empolis.com

University of Karlsruhe, Institute AIFB
Englerstr. 28
D-76128 Karlsruhe
Germany
Tel: +49 721 608 6592, Fax: +49 721 608 6580
Contact person: York Sure
E-mail: sure@aifb.uni-karlsruhe.de

University of Innsbruck
Institute of Computer Science
Techikerstraße 13
6020 Innsbruck
Austria
Tel: +43 512 507 6475, Fax: +43 512 507 9872
Contact person: Jos de Bruijn
E-mail: jos.de-bruijn@deri.ie

Kea-pro GmbH
Tal
6464 Springen
Switzerland
Tel: +41 41 879 00, Fax: 41 41 879 00 13
Contact person: Tom Bösser
E-mail: tb@keapro.net

Sirma AI EAD, Ontotext Lab
135 Tsarigradsko Shose
Sofia 1784
Bulgaria
Tel: +359 2 9768 303, Fax: +359 2 9768 311
Contact person: Atanas Kiryakov
E-mail: naso@sirma.bg

Universitat Autonoma de Barcelona
Edifici B, Campus de la UAB
08193 Bellaterra (Cerdanyola del Vallès)
Barcelona
Spain
Tel: +34 93 581 22 35, Fax: +34 93 581 29 88
Contact person: Pompeu Casanovas Romeu
E-mail: pompeu.casanovas@uab.es

Executive Summary

In this document we evaluate our logical framework for debugging we had introduced in Sekt Deliverable 3.6.1. This evaluation is done in two ways: first, we study the effectiveness of our proposal in a *qualitative* way with some practical examples. Secondly, in a *quantitative and statistical* analysis, we try to get a better understanding of the computational properties of the debugging problem and our algorithms for solving it.

Instead of providing a wide and shallow evaluation for all types of debugging for all sorts of languages, we have opted for a very detailed analysis of a particular class of ontologies, namely the terminological part, and here we even focus on unfoldable ALC TBoxes. These restrictions have grown out of our own applications, and we have most experience in these cases. I believe that the lessons we learn can be extended to other cases.

In the qualititative part we discuss two simple incoherent terminologies to explain the functionality and particular application scenarios of our debugging framework. This framework was then applied to two incoherent terminologies which were used at the Academic Medical Center, Amsterdam, for the admission of patients to Intensive Care units. This evaluation is described in Chapter 4.

For the statistical parts we conducted three sets of experimments in order to evaluate the debugging problem and our algorithms and tools. First, we applied our two debuggers DION and MUPSter on a set of real-world terminologies collected from our applications and the WWW. Secondly, we translated an existing test-set for Description Logic satisfiability to an incoherence problem, and finally, we created our own benchmark.

The results are mixed: although debugging is useful in practice, we cannot guarantee that our tools will calculate debugs in reasonable time. The most important criteria are the size and complexity of the definitions and the number of modeling errors.

This deliverable partly overlaps with the previous Sekt Deliverable 3.6.1. in order to make it self-contained. This applies to Chapter 3, which is mostly copied from the previous Deliverable, and to Chapter 2, which contains mostly known material.

Contents

Chapter 1

Introduction

Ontologies play a crucial role in the Semantic Web (SW), as they allow "intelligent agents" to share information in a semantically umambiguous way, and to reuse domain knowledge (possibly created by external sources). However, this makes SW technology highly dependent of the quality, and, in particular, of the correctness of the applied ontology. Two general strategies for quality assurance are predominant, one based on developing more and more sophisticated ontology modeling tools, the second one based on logical reasoning. In this paper we will focus on the latter. With the advent of expressive ontology languages such as OWL and its close relation to Description Logics (DL), non-trivial implicit information, such as the `is-a` hierarchy of classes, can often be made explicit by logical reasoners. More crucially, however, state-of-the art DL reasoners can efficiently detect incoherences even in very large ontologies. The practical problem remains what to do in case an ontology has been detected to be incoherent.

Our work was motivated initially by the development of the DICE[1] terminology. DICE implements frame-based definitions of diagnostic information for the unambiguous and unified classification of patients in Intensive Care medicine. The representation of DICE is currently being migrated to an expressive Description Logic (henceforth DL) to facilitate logical inferences. Figure 1.1 shows an extract of the DICE terminology. In [4] the authors describe the migration process in more detail. The resulting DL terminology (usually called a "TBox") contains axioms such as the following, where classes (like BODYPART) are translated as concepts, and slots (like REGION) as roles:

$$\begin{aligned}
&\mathit{Brain} \mathrel{\dot\sqsubseteq} \mathit{CNS} \sqcap \exists \mathit{systempart.NervousSystem} \sqcap \\
&\quad \mathit{BodyPart} \sqcap \exists \mathit{region.HeadAndNeck} \sqcap \forall \mathit{region.HeadAndNeck} \\
&\mathit{CNS} \mathrel{\dot\sqsubseteq} \mathit{NervousSystem}
\end{aligned}$$

Developing a coherent terminology is a time-consuming and error-prone process. DICE defines more than 2400 concepts and uses 45 relations. To illustrate some of the

[1]DICE stands for "Diagnoses for Intensive Care Evaluation". The development of the DICE terminology has been supported by the NICE foundation.

Class	SuperClass	Slot	Slot-Value
Brain	BodyPart	region	Head and Neck
	CNS	system part	Nervous System
CNS	Nervous System		

Figure 1.1: An extract from the DICE terminology (frame-based).

problems, take the definition of a "brain" which is incorrectly specified, among others, as a "CNS" (central nervous-system) and "body-part" located in the head. This definition is contradictory as nervous-systems and body-parts are declared disjoint in DICE. Fortunately, current Description Logic reasoners, such as RACER [12] or FaCT [13], can detect this type of inconsistency and the knowledge engineer can identify the cause of the problem. Unfortunately, many other concepts are defined based on the erroneous definition of "brain" forcing each of them to be erroneous as well. In practice, DL reasoners provide lists of hundreds of unsatisfiable concepts for the DICE TBox and the debugging remains a jigsaw to be solved by human experts, with little additional explanation to support this process.

There are two main ways to deal with inconsistenct ontologies. One is to simply avoid the inconsistency and to apply a non-standard reasoning method to obtain meaningful answers. In [14, 15], a framework of reasoning with inconsistent ontologies, in which pre-defined selection functions are used to deal with concept relevance, is presented The notion of "concept relevance" can be used for reasoning with inconsistent ontologies.

An alternative approach to deal with logical contradictions is to resolve logical modeling errors whenever a logical problem is encountered. In this document, we will focus on evaluation of this *debugging* process. In Sekt Deliverable 3.6.1 we introduce a framework for debugging and diagnosis, more precisely the notions of *minimal unsatisfiability-preserving sub-TBoxes* (abbreviated MUPS) and *minimal incoherence-preserving sub-TBoxes* (MIPS) as the smallest subsets of axioms of an incoherent terminology preserving unsatisfiability of a particular, respectively of at least one unsatisfiable concept.

An orthogonal view on inconsistent ontologies is based on the traditional model-based *diagnosis (MBD)* which has been studies over many years in the AI community [22]. Here the aim is to find minimal fixes, i.e. minimal subsets of an ontology that need to be repaired or removed to render an ontology logically correct, and therefor usable again. In Reiter's terminology, MIPS and MUPS would be minimal conflict sets, which implies that, technically, calculating diagnosis depends on debugging. Therefore, we focus on the evaluation of debugging in this document.

There are basically two approaches for debugging, a bottom-up method using the support of an external reasoner, and a top-down implementation of a specialised algorithm. In this paper we describe one such approach each, the former based on the systematic enumerations of terminologies of increasing size based on selection functions on axioms, the latter on Boolean minimisation of labels in a labelled tableau calculus.

Both methods have been implemented as prototypes. The prototype for the informed bottom-up approach is called DION, which stands for a Debugger of Inconsistent ONtologies, the prototype of the specialised top-down method is called MUPSter. In SEKT deliverable 3.6.1, we discussed some implementation issue of both these systems, and provided a basic introduction on how to use the systems for debugging. What is missing is a detailed evaluation of the quality of the proposed framework, and this paper attempts to close this gap.

This is done in two ways: first, we first, we study the effectiveness of our proposal in a *qualitative* way with some practical examples. Secondly, in a *quantitative and statistical* analysis, we try to get a better understanding of the computational properties of the debugging problem and our algorithms for solving it.

In the qualititative part we discuss two simple incoherent terminologies to explain the functionality and particular application scenarios of our debugging framework. This framework was then applied to two incoherent terminologies which were used at the Academic Medical Center, Amsterdam, for the admission of patients to Intensive Care units. This evaluation is described in Chapter 4.

For the statistical parts we conducted three sets of experimments in order to evaluate the debugging problem and our algorithms and tools. First, we applied our two debugger DION and MUPSter on a set of real-world terminologies collect from our applications and the WWW. Secondly, we translated an existing test-set for Description Logic satisfiability to an incoherence problem, and finally, we created our own benchmark.

The results are mixed: although debugging is useful in practice, we cannot guarantee that our tools will calculate debugs in reasonable time. The most important criteria are the size and complexity of the definitions and the number of modeling errors.

The research underlying this report was driven by practical needs, as the MUPSter was implemented to debug the DICE ontology. This means that the top-down method to calculate MIPS and MUPS is implemented for unfoldable $\mathcal{ALC}$ TBoxes, only. This implies that this report focuses solely on debugging of $\mathcal{ALC}$ terminologies, and evaluates the methods with under these constraints. An extension to full ontologies in more expressive languages is left for future research.

Overview This paper is organized as follows. We will first introduce relevant background to the problem of debugging and the approach we have developed in Chapter 2. Furthermore we will remind the reader about the implementation of our methods in Chapter 3. This chapter repeats known results from Sekt Deliverable 3.6.1. The qualitative study of our methods is described in Chapter 4. The largest part of this report is the quantitative evaluation of the algorithms for debugging and diagnosis. Here, we first describe our evaluation method with three different types of evaluation in Chapter 5 and then the results in Chapter 6. We end this report with some concluding remarks.

Chapter 2

Debugging Inconsistent Terminologies

This chapter deals with debugging and diagnosis of inconsistent Description Logic ontologies. Description Logics are a family of well-studied set-description languages which have been in use for over two decades to formalize knowledge. They have a well-defined model theoretic semantics, which allows for the automation of a number of reasoning services.

2.1 Logical errors in Description Logic terminologies

We shall not give a formal introduction into Description Logics here, but point to the second chapter of the DL handbook [1] for an excellent introduction. Briefly, in DL concepts will be interpreted as subsets of a domain, and roles as binary relations. In a terminological component $\mathcal{T}$ (called TBox) the interpretations of concepts can be restricted to the *models* of $\mathcal{T}$. Let, throughout the paper, $\mathcal{T} = \{Ax_1, \ldots, Ax_n\}$ be a set of (terminological) axioms, where Ax_i is of the form $C_i \sqsubseteq D_i$ for each $1 \leq i \leq n$ and arbitrary concepts C_i and D_i. A TBox is called *unfoldable* if the left-hand sides of the axioms (the defined concepts) are atomic, and if the right-hand sides (the definitions) contain no direct or indirect reference to the defined concept [19].

Let $\mathcal{U}$ be a finite set, called the universe. A mapping $\mathcal{I}$, which interpretes DL concepts as subsets of $\mathcal{U}$ is a *model* of a terminological axiom $C \sqsubseteq D$, if, and only if, $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. A *model for a TBox* $\mathcal{T}$ is an interpretation which is a model for all axioms in $\mathcal{T}$. Based on this semantics a TBox can be checked for *incoherence*, i.e., whether there are *unsatisfiable* concepts: concepts which are necessarily interpreted as the empty set in all models of the TBox. More formally

1. A concept A is *unsatisfiable* w.r.t. a terminology $\mathcal{T}$ if, and only if, $A^{\mathcal{I}} = \varnothing$ for all models $\mathcal{I}$ of $\mathcal{T}$.

2. A TBox $\mathcal{T}$ is *incoherent* if there is a concept-name in $\mathcal{T}$, which is unsatisfiable.

$ax_1{:}A_1 \sqsubseteq \neg A \sqcap A_2 \sqcap A_3$	$ax_2{:}A_2 \sqsubseteq A \sqcap A_4$
$ax_3{:}A_3 \sqsubseteq A_4 \sqcap A_5$	$ax_4{:}A_4 \sqsubseteq \forall s.B \sqcap C$
$ax_5{:}A_5 \sqsubseteq \exists s.\neg B$	$ax_6{:}A_6 \sqsubseteq A_1 \sqcup \exists r.(A_3 \sqcap \neg C \sqcap A_4)$
$ax_7{:}A_7 \sqsubseteq A_4 \sqcap \exists s.\neg B$	

Table 2.1: A small (incoherent) TBox $\mathcal{T}_1$, where A, B and C are atomic and $A_1, \ldots, A_7$ defined concept names, and r and s are atomic roles.

Conceptually, these cases are *simple* modeling errors because we assume that a knowledge modeler would not specify something an impossible concept in a complex way.

Table 2.1 demonstrates this principle. Consider the (incoherent) TBox $\mathcal{T}_1$, where A, B and C are atomic and $A_1, \ldots, A_7$ defined concept names, and r and s are atomic roles. Satisfiability testing of the TBox by a DL reasoner returns a set of unsatisfiable concept names $\{A_1, A_3, A_6, A_7\}$. Although this is still of manageable size, it hides crucial information, e.g., that unsatisfiability of A_1 depends, among others, on unsatisfiability of A_3, which is in turn unsatisfiable because of the contradictions between A_4 and A_5. We will use this example later in this paper to explain our debugging methods.

In this chapter we study ways of *explaining* incoherence and unsatisfiability in DL terminologies, and inconsistency of ontologies by ways of debugging and diagnosis.

2.2 Framework for debugging and diagnosis

In Sekt Deliverable 3.6.1. we introduced a theory of debugging and diagnosis and linked it to description logic-based systems, in which case a diagnosis is a smallest set of axioms that needs to be removed or corrected to render a specific concept or all concepts satisfiable. Orthogonally, we considered the smallest set of axioms that still contained the logical contradictions.

In this we section describe both methods on the use of minimal conflict sets for explaining unsatisfiability and incoherence in DL-based ontologies. We will not elaborate on algorithms to construct such conflict sets, the interested reader is referred to Chapter 3, and to [24],[25] and [26].

We propose to simplify a TBox $\mathcal{T}$ in order to reduce the available information to the root of the incoherence. More concretely we first exclude axioms which are irrelevant to the incoherence and then provide simplified definitions highlighting the exact position of a contradiction within the axioms of this reduced TBox. We will call the former “axiom pinpointing”, the latter “concept pinpointing”. In this section we will formally introduce axiom and concept pinpointing for a general TBox without restrictions on the underlying description logic.

In our analogy to diagnosis, we consider the ontology to be the system, where the axioms are the components of the system. Satisfiability of a concept is taken as a measurement, where the system description states that all concepts are satisfiable.

First, we will define minimal conflict sets w.r.t. satisfiability of a concept. Next, we will define minimal conflict sets w.r.t. coherence of the ontology as a whole. Thereafter we describe a generalization method as a means of providing focus on those parts of axioms that lead to unsatisfiability of concepts.

In some situations, ontologies can contain a large number of unsatisfiable concepts. This can occur for example when ontologies are the result of a merging process of separately developed ontologies, or when closure axioms (i.e. disjointness statements and universal restrictions) are added to ontologies. Unsatisfiability propagates, i.e. one unsatisfiable concept may cause many other concepts to become unsatisfiable as well. As it is often not clear to a modeler what concepts are the root cause of unsatisfiability, we also describe a number of heuristics that help to indicate reasonable starting points for debugging an ontology.

2.2.1 Model-based Diagnosis

The literature on model-based diagnosis is manifold, but we focus on the seminal work of Reiter [22], and [11], which corrects a small bug in Reiter's original algorithm. We refer the interested reader to a good overview in [3]. A more formal description can also be found in [24].

Reiter introduced a diagnosis [22] of a system as the smallest set of components from that system with the following property: the assumption that each of these components is faulty (together with the assumption that all other components are behaving correctly) is consistent with the system description and observation. For example, a simple electrical circuit can be defined, consisting of a number of adders. Based on the description of the system and some input values, one can calculate the output of the system. If the observed output is different from the expected output, at least one of the components must be faulty, and diagnoses determine which components could have caused the error.

To apply this definition to a description logic ontology, the *system* is the ontology, and the *components* of the system are the axioms. The concepts and roles in a concept definition are regarded as *input values*, and the defined concepts as *output values*.

If we look at the example ontology from Table 2.1, the *system description* states that it is coherent (i.e. all concepts are satisfiable), but the *observation* is that A_1, A_3, A_6, and A_7 are unsatisfiable.

Reiter provides a generic method to calculate diagnoses on the basis of conflict sets and their minimal hitting sets. A conflict set is a set of components that, when assumed to be fault free, lead to an inconsistency between the system description and observations. A conflict set is minimal if and only if no proper subset of it is a conflict set. The minimal conflict sets (w.r.t. coherence) for the system in Table 2.1 are $\{ax_1, ax_2\}$, $\{ax_3, ax_4, ax_5\}$, and $\{ax_4, ax_7\}$.

A hitting set H for a collection of sets C is a set that contains at least one element of each of the sets in C. Formally: $H \subseteq \bigcup_{S \in C} S$ such that $H \cap S \neq \emptyset$ for each $S \in C$. A hitting set is minimal if and only if no proper subset of it is a hitting set. Given the conflict sets above, the minimal hitting sets are: $\{ax_1, ax_3, ax_7\}$, $\{ax_1, ax_4\}$, $\{ax_1, ax_5, ax_7\}$, $\{ax_2, ax_3, ax_7\}$, $\{ax_2, ax_4\}$, and $\{ax_2, ax_5, ax_7\}$.

Reiter shows that the set of diagnoses actually corresponds to the collection of minimal hitting sets for the minimal conflict sets. Hence, the minimal hitting sets given above determine the diagnoses for the system w.r.t. coherence.

In [7] diagnosis is extended by providing a method for computing the probabilities of failure of various components based on given measurements. Especially in cases where there are many diagnoses, additional observations (measurements) need to be made in order to determine the actually failing components. The method provided can also determine what observation has the highest discriminating power, i.e. needs to be performed to maximally reduce the number of diagnoses.

2.2.2 Debugging

As previously mentioned the theory of diagnosis is built on minimal conflict sets. But in the application of diagnosis of erroneous ontologies, these minimal conflict sets play a role of their own, as they are the prime tools for debugging, i.e. for the identification of potential errors. For different kind of logical contradictions we introduce several different notions based on conflict sets, the MUPS for unsatisfiability of a concept, the MIPS for incoherence of a terminology, and a number of heuristics.

Minimal unsatisfiability-preserving sub-TBoxes (MUPS) In [25] we introduced the notion of Minimal Unsatisfiability Preserving Sub-TBoxes (MUPS) to denote minimal conflict sets. Unsatisfiability-preserving sub-TBoxes of a TBox $\mathcal{T}$ and an unsatisfiable concept A are subsets of $\mathcal{T}$ in which A is unsatisfiable. In general there are several of these sub-TBoxes and we select the minimal ones, i.e., those containing only axioms that are necessary to preserve unsatisfiability. A TBox $\mathcal{T}' \subseteq \mathcal{T}$ is a MUPS of $\mathcal{T}$ if A is unsatisfiable in $\mathcal{T}'$, and A is satisfiable in every sub-TBox $\mathcal{T}'' \subset \mathcal{T}'$. We will abbreviate the set of MUPS of $\mathcal{T}$ and A by $mups(\mathcal{T}$, A). MUPS for our example TBox $\mathcal{T}_1$ and its unsatisfiable concepts are:

$mups(\mathcal{T}_1, A_1)$: $\{\{ax_1, ax_2\}, \{ax_1, ax_3, ax_4, ax_5\}\}$

$mups(\mathcal{T}_1, A_3)$: $\{ax_3, ax_4, ax_5\}$

$mups(\mathcal{T}_1, A_6)$: $\{\{ax_1, ax_2, ax_4, ax_6\}, \{ax_1, ax_3, ax_4, ax_5, ax_6\}\}$

$mups(\mathcal{T}_1, A_7)$: $\{ax_4, ax_7\}$

It can be easily proven that each MUPS($\mathcal{T}$, A) is a minimal conflict set w.r.t. satisfiability of concept A in TBox $\mathcal{T}$.

As explained in Section 2, a diagnosis is a minimal hitting set for a conflict set. Hence, from the MUPS, we can also calculate the diagnoses for satisfiability of concept A in TBox $\mathcal{T}$, which we will denote $\Delta_{\mathcal{T},A}$.

$\Delta_{\mathcal{T},A1}$: $\{\{ax_1\}, \{ax_2, ax_3\}, \{ax_2, ax_4\}, \{ax_2, ax_5\}\ \}$

$\Delta_{\mathcal{T},A3}$: $\{\{ax_3\}, \{ax_4\}, \{ax_5\}\}$

$\Delta_{\mathcal{T},A6}$: $\{\{ax_1\}, \{ax_4\}, \{ax_6\}, \{ax_2, ax_3\}, \{ax_2, ax_5\}\ \}$

$\Delta_{\mathcal{T},A7}$: $\{\{ax_4\}, \{ax_7\}\}$

Minimal incoherence-preserving sub-TBoxes (MIPS) MUPS are useful for relating sets of axioms to unsatisfiability of specific concepts, but they can also be used to calculate MIPS, which relate sets of axioms to incoherence of a TBox (i.e. unsatisfiability of any concept in a TBox).

In [25] we introduced Minimal Incoherence Preserving Sub-TBoxes (MIPS) as the smallest subsets of an original TBox preserving unsatisfiability of at least one atomic concept. The set of MIPS for a TBox $\mathcal{T}$ is abbreviated with $mips(\mathcal{T})$. For $\mathcal{T}_1$ we get 3 MIPS:

$mips(\mathcal{T}_1) = \{\{ax_1, ax_2\}, \{ax_3, ax_4, ax_5\}, \{ax_4, ax_7\}\}$

Analogous to MUPS, each MIPS($\mathcal{T}$) is a minimal conflict set w.r.t. coherence of TBox $\mathcal{T}$. Hence, from $mips(\mathcal{T})$, a diagnosis for coherence of $\mathcal{T}$ can be calculated, which we denote as $\Delta_{\mathcal{T}}$. From these definitions, we can determine the diagnosis for coherence of $\mathcal{T}_1$:

$\Delta_{\mathcal{T}1} = \{\{ax_1, ax_4\}, \{ax_2, ax_4\}, \{ax_1, ax_3, ax_7\}, \{ax_2, ax_3, ax_7\}, \{ax_1, ax_5, ax_7\}, \{ax_2, ax_5, ax_7\}\}$

Generalized MIPS The use of MUPS and MIPS provides a focus on potentially incorrect axioms by reducing the number of axioms, while retaining unsatisfiability of a specific concept or incoherence of the ontology as a whole. A further step towards pinpointing is to look into the axioms, aiming at determining the concept expressions within axioms that lead to unsatisfiability.

We will describe a procedural approach to generalization of MIPS. This approach is comparable to those used in structural subsumption algorithms. The first step is to represent the axioms in the MIPS in conjunctive normal form, i.e. $A \sqsubseteq X_1 \sqcap X_2 \sqcap \ldots \sqcap X_n$, where $X_1 \ldots X_n$ are concept names or concept expressions that do not contain conjunctions. The next step is to minimize the number of conjuncted concept expressions in the axioms, while retaining incoherence. The resulting axioms provide generalizations of the concepts from the MIPS, which we will call GMIPS. For the example $\mathcal{T}_1$ we get these generalized axioms:

GMIPS $\{ax_1, ax_2\}$: $\{ A'_1 \sqsubseteq \neg A' \sqcap A'_2 , A'_2 \sqsubseteq A' \}$
GMIPS $\{ax_3, ax_4, ax_5\}$: $\{ A'_3 \sqsubseteq A'_4 \sqcap A'_5 , A'_4 \sqsubseteq \forall s.B' , A'_5 \sqsubseteq \exists s.\neg B' \}$
GMIPS $\{ax_4, ax_7\}$: $\{ A'_4 \sqsubseteq \forall s.B' , A'_7 \sqsubseteq A'_4 \sqcap \exists s.\neg B' \}$

A'_1, A'_2 and A'_4 provide generalizations of the definitions of A_1, A_2 and A_4, respectively. The definitions of A_3 and A_5 could not be further generalized without losing incoherence.

2.2.3 Heuristics

Now that we have introduced the basic notion w.r.t. pinpointing, we will also provide a number of heuristics. These heuristics aim at indicating those axioms that are likely to be involved in larger numbers of unsatisfiable concepts. As was described earlier, unsatisfiability of a concept propagates to subsumees of that concept, and to concepts that use the unsatisfiable concept as the value of an existential restriction. In the example TBox $\mathcal{T}_1$, unsatisfiability of A_3 is one of the two causes for unsatisfiability of A_1 (as A_1 is subsumed by A_3). But it is also a cause for unsatisfiability of A_6. Firstly, because it renders A_1 unsatisfiable, secondly because it renders the conjunction $A_3 \sqcap \neg C \sqcap A_4$ unsatisfiable, which in turn leads to unsatisfiability of $\exists r.(A_3 \sqcap \neg C \sqcap A_4)$.

Hence, solving unsatisfiability of A_3 might also lead to satisfiability of A_1 and A_6. In this example, A_1 will actually remain unsatisfiable due to conflicting definitions of A_1 and A_2.

MIPS-weights Every MIPS is a subset of one or more MUPS. One indication of the effect of propagation is the number of MUPS that a MIPS is a subset of. The higher this number, the more likely it is that the axioms in the MIPS are the cause of unsatisfiability for more concepts.

We define the MIPS-weight as the number of MUPS of which a MIPS is a subset.

In the example ontology $\mathcal{T}_1$ we found six MUPS and three MIPS. The MIPS $\{ax_1, ax_2\}$ is equivalent to one of the MUPS for A_1, $\{ax_1, ax_2\}$, and a proper subset of a MUPS for A_6, $\{ax_1, ax_2, ax_4, ax_6\}$. Hence, the weight of MIPS $\{ax_1, ax_2\}$ is two. In the same way we can calculate the weights for the other MIPS: the weight of $\{ax_3, ax_4, ax_5\}$ is three, the weight of $\{ax_4, ax_7\}$ is one.

Intuitively, this suggests that the combination of the axioms $\{ax_3, ax_4, ax_5\}$ may be the cause of more than one unsatisfiable concept, whereas $\{ax_4, ax_7\}$ only leads to unsatisfiability of one concept, A_7.

Cores MIPS-weights provide an intuition of which combinations of axioms lead to unsatisfiability. Alternatively, one can focus on the occurrence of the individual axioms in MIPS, in order to predict the likelihood that an individual axiom is erroneous.

We define cores as sets of axioms occurring in several MIPS. The more MIPS such a core belongs to, the more likely its axioms will be the cause of contradictions. A non-empty intersection of n different MIPS in mips($\mathcal{T}$) (with $n \geq 1$) is called a MIPS-core of arity n (or simply n-ary core) for $\mathcal{T}$.

For our example TBox $\mathcal{T}_1$ we find one 2-ary core, ax_4. The other axioms in the MIPS are 1-ary cores.

Pinpoints Pinpoints were introduced in [26], as a computationally attractive alternative for diagnoses. As calculation of diagnoses is a so-called NP-complete problem (i.e. most likely not solvable in polynomial time), we use the cores described above to construct a pinpoint.

Pinpoints are constructed as follows. Take a core $\{ax\}$ of size 1 with maximal arity. Then, remove from the $mips$ all MIPS containing $\{ax\}$. Repeat these steps until there are no MIPS left. The cores form a pinpoint for the ontology.

For our example TBox $\mathcal{T}_1$ with $mips(\mathcal{T}_1)$ = $\{\{ax_1, ax_2\}, \{ax_3, ax_4, ax_5\}, \{ax_4, ax_7\}\}$ we first take 2-ary core, ax_4. Removing the MIPS containing this axiom leaves the MIPS $\{ax_1, ax_2\}$. Hence, two pinpoints can be defined: $\{ax_4, ax_1\}$ and $\{ax_4, ax_2\}$.

A note on terminology In the remainder of this report we will often refer to the calculation of MIPS and MUPS as the process of *debugging*, as opposed to the *diagnosis* process, which refers to the calculation of diagnoses in the proper sense of the word.

Chapter 3

Algorithms for debugging and diagnosis

3.1 Two algorithms for debugging

We present two general approaches to calculate explanations: a top-down method, which reduces the reasoning into smaller parts in order to explain a subproblem with reduced complexity, and an informed bottom-up approach, which enumerates possible solutions in a clever way. Both approaches will be represented for terminological reasoning only, but can, in principal, easily be extended to full ontology debugging.[1]

3.1.1 A top-down approach to explanation

In order to calculate minimal incoherence preserving sub-terminologies (MIPS) we first calculate the minimal unsatisfiability preserving sub-terminologies (MUPS) for each unsatisfiable concept. This is done in a top-down way: we calculate the set of axioms needed to maintain a logical contradiction by expanding a logical tableau with labels. This method is efficient, as it requires a single logical calculation per unsatisfiable concept. On the other hand, it is based on a variation of a specialised logical algorithm, and only works for Description Logics for which such a specialised algorithm exists and is implemented. At the moment, such a purpose-build method only exists for the DL $\mathcal{ALC}$, and a restricted type of TBoxes, namely *unfoldable* ones.

Debugging unfoldable $\mathcal{ALC}$-TBoxes Practical experience has shown that applying our methods on a simplified version of DICE can already provide valuable debugging information. We will therefore only provide algorithms for unfoldable $\mathcal{ALC}$-TBoxes [19] as

[1]The extension of the bottom-up method is trivial, as we only have to define a new selection function and systematic enumeration. For the top-down approach things are a bit more complicated, as we now have analyse forests rather than trees. However, this seems to be a technical rather than a conceptual problem.

$(\sqcap)$:	**if**	$(a : C_1 \sqcap C_2)^{label} \in B$, but not both $(a : C_1)^{label} \in B$ and $(a : C_2)^{label} \in B$
	then	$B' := B \cup \{(a : C_1)^{label}, (a : C_2)^{label}\}$.
$(\sqcup)$:	**if**	$(a : C_1 \sqcup C_2)^{label} \in B$, but neither $(a : C_1)^{label} \in B$ nor $(a : C_2)^{label} \in B$
	then	$B' := B \cup \{(a : C_1)^{label}\}$ and $B'' := B \cup \{(a : C_2)^{label}\}$.
(Ax)	**if**	$(a : A)^{label} \in B$ and $(A \sqsubseteq C) \in \mathcal{T}$
	then	$B' := B \cup \{(a : C)^{label \cup \{A \sqsubseteq C\}}\}$.
$(\exists)$:	**if**	$(a : \exists R_i.C)^{label} \in B$, $R_i \in N_R$ and all other rules have been applied on all formulas over a, and if $\{(a : \forall R_i.C_1)^{label_1}, \ldots, (a : \forall R_i.C_n)^{label_n}\} \subseteq B$ is the set of universal formulas for a w.r.t. R_i in B,
	then	$B' := \{(b : C)^{label}, (b : C_1)^{label_1 \cup label}, \ldots, (b : C_n)^{label_n \cup label}\}$ where b is a new individual name not occurring in B.

Figure 3.1: Tableau Rules for $\mathcal{ALC}$-Satisfiability w.r.t. a TBox $\mathcal{T}$ (with Labels)

this significantly improves both the computational properties and the readability of the algorithm.

The calculation of MIPS depends on the MUPS only, and we will provide an algorithm to calculate these minimal unsatisfiability-preserving sub-TBoxes based on Boolean minimisation of terminological axioms needed to close a standard tableau ([1] Chapter 2).

Usually, unsatisfiability of a concept is detected with a fully saturated tableau (expanded with rules similar to those in Figure 3.1) where all branches contain a contradiction (or close, as we say). The information which axioms are relevant for the closure is contained in a simple label which is added to each formula in a branch. A *labelled formula* has the form $(a : C)^x$ where a is an individual name, C a concept and x a set of axioms, which we will refer to as *label*. A labelled branch is a set of labelled formulas and a tableau is a set of labelled branches. A formula can occur with different labels on the same branch. A branch is closed if it contains a clash, i.e. if there is at least one pair of formulas with contradictory atoms on the same individual. The notions of open branch and closed and open tableau are defined as usual and do not depend on the labels. We will always assume that any formula is in *negation normal form* (nnf) and newly created formulas are immediately transformed. We usually omit the prefix “labelled”.

To calculate a minimal unsatisfiability-preserving TBox for a concept name A w.r.t. an unfoldable TBox $\mathcal{T}$ we construct a tableau from a branch B initially containing only $(a : A)^{\varnothing}$ (for a new individual name a) by applying the rules in Figure 3.1 as long as possible. The rules are standard $\mathcal{ALC}$-tableau rules with lazy unfolding, and have to be read as follows: assume that there is a tableau $T = \{B, B_1, \ldots, B_n\}$ with $n+1$ branches. Application of one of the rules on B yields the tableau $T' := \{B', B_1, \ldots, B_n\}$ for the $(\sqcap), (\exists)$ and (Ax)-rule, $T'' := \{B', B'', B_1, \ldots, B_n\}$ in case of the $(\sqcup)$-rule.

Once no more rules can be applied, we know which atoms are needed to close a saturated branch and can construct a minimisation function for A and $\mathcal{T}$ according to the rules in Figure 3.2. A propositional formula ϕ is called a *minimisation function for A and $\mathcal{T}$* if A is unsatisfiable in every subset of $\mathcal{T}$ containing the axioms which are true in

```
if rule = ($\sqcap$) has been applied to $(a : C_1 \sqcap C_2)^{label}$ and $B'$ is the new branch
  return min_function$(a, B', \mathcal{T})$;
if rule = ($\sqcup$) has been applied to $(a : C_1 \sqcup C_2)^{label}$ and $B'$ and $B''$ are the new branches
  return min_function$(a, B', \mathcal{T}) \wedge$ min_function$(a, B'', \mathcal{T})$;
if rule = ($\exists$) has been applied to $(a : \exists R.C)^{label}$, $B'$ is the new branch and $b$ the new variable
  return min_function$(a, B', \mathcal{T}) \vee$ min_function$(b, B', \mathcal{T})$;
if rule = (Ax) has been applied and $B'$ is the new branch
  return min_function$(a, B', \mathcal{T})$;
if no further rule can be applied
  return: $\bigvee_{(a : A)^x \in B, (a : \neg A)^y \in B} (\bigwedge_{ax \in x} ax \wedge \bigwedge_{ax \in y} ax)$;
```

Figure 3.2: *min_function*$(a, B, \mathcal{T})$: Minimisation-function for the MUPS-problem

an assignment making ϕ true. In our case axioms are used as propositional variables in ϕ. As we can identify unsatisfiability of A w.r.t. a set S of axioms with a closed tableau using only the axioms in S for unfolding, branching on a disjunctive rule implies that we need to join the functions of the appropriate sub-branches conjunctively. If an existential rule has been applied, the new branch B' might not necessarily be closed on formulas for both individuals. Assume that B' closes on the individual a but not on b. In this case *min_function*$(a, B, \mathcal{T}) = \bot$, which means that the related disjunct does not influence the calculation of the minimal incoherent TBox.

Based on the minimisation function *min_function*$(a, \{(a : A)^{\varnothing}\}, \mathcal{T})$ (let us call it ϕ) which we calculated using the rules in Figure 3.2 we can now calculate the MUPS for A w.r.t. $\mathcal{T}$. The idea is to use prime implicants of ϕ. A prime implicant $ax_1 \wedge \ldots \wedge ax_n$ is the smallest conjunction of literals[2] implying ϕ [21]. As ϕ is a minimisation function every implicant of ϕ must be a minimisation function as well and therefore also the prime implicant. But this implies that the concept A must be unsatisfiable w.r.t. the set of axioms $\{ax_1, \ldots, ax_n\}$. As $ax_1 \wedge \ldots \wedge ax_n$ is the smallest implicant we also know that $\{ax_1, \ldots, ax_n\}$ must be minimal, i.e. a MUPS.

From MUPS we can easily calculate MIPS, but we need an additional operation on sets of TBoxes, called *subset-reduction*. Let $M = \{\mathcal{T}_1, \ldots, \mathcal{T}_m\}$ be a set of TBoxes. The *subset-reduction* of M is the smallest subset $sr(M) \subseteq M$ such that for all $\mathcal{T} \in M$ there is a set $\mathcal{T}' \in sr(M)$ such that $\mathcal{T}' \subseteq \mathcal{T}$.

Let $\mathcal{T}$ be an incoherent TBox with unsatisfiable concepts $\Delta^{\mathcal{T}}$. A simple algorithm for the calculation of MIPS for $\mathcal{T}$ is then defined through the following equation: Then, $mips(\mathcal{T}) = sr(\bigcup_{A \in \Delta^{\mathcal{T}}} mups(\mathcal{T}, A))$. Checking elements of $mips(\mathcal{T})$ for cores of maximal arity requires exponentially many checks in the size of $mips(\mathcal{T})$. In practice, we therefore apply a bottom-up method searching for maximal cores of increasing size stopping once the arity of the cores is smaller than 2.

[2]Note that in our case all literals are non-negated axioms.

3.1.2 An Informed Bottom-up Approach to Explanation

In this section we propose an informed bottom-up approach to calculate MUPS by the support of an external DL reasoner, like RACER. The main advantage of this approach is that it can deal with any DL-based ontology if it has been supported by an external reasoner. Currently there exist several well-known DL reasoners, like RACER and FACT++. Those external DL reasoners have been proved to be very reliable and stable. They already support various DL-based ontology languages, including OWL. Thus, by the bottom-up approach we can obtain an OWL debugger almost for free, although the price is paid for its performance.

Given an unsatisfiable concept c and a formula set(i.e., an ontology) $\mathcal{T}$, we can calculate the MUPS of c by selecting a minimal subset Σ of $\mathcal{T}$ in which c is unsatisfiable in Σ. We use a similar selecting procedure which has been used in the system PION for reasoning with inconsistent ontologies[14]. In the PION approach, a selection function is designed to one which can extend selected subset by checking on axioms which are relevant to the current selected subset which starts initially with a query. Although the approach which is based this kind of relevance extension procedure may not give us the complete solution set of MUPS/MIPS, it is good enough to provide us an efficient approach for debugging inconsistent ontologies. We are going to report the evaluation of this informed bottom-up approach in the SEKT deliverable D3.6.2 entitled ”Evaluation of Inconsistent Ontology Diagnosis”.

Selection Function and Relevance Measure Given an ontology (i.e., a formula set) Σ and a query ϕ, a selection function s is one which returns a subset of Σ at the step $k > 0$. Let **L** be the ontology language, which is denoted as a formula set. We have the general definition about selection functions as follows:

Definition 3.1.1 (Selection Functions) *A selection function s is a mapping $s : \mathcal{P}(\mathbf{L}) \times \mathbf{L} \times N \rightarrow \mathcal{P}(\mathbf{L})$ such that $s(\Sigma, \phi, k) \subseteq \Sigma$.*

In this approach, we extend the definition of the selection function so that it starts from a concept c instead of from a query (i.e., a formula ϕ). As we have discussed above, select functions are usually defined by a relevance relation between a formula and a formula set. We will use a relevance relation as the informed message to guide the search strategy for MUPS.

Definition 3.1.2 (Direct Relevance Relation) *A direction relevance relation $\mathcal{R}$ is a set of formula pairs. Namely, $\mathcal{R} \subseteq \mathbf{L} \times \mathbf{L}$.*

Definition 3.1.3 (Direct Relevance Relation between a Formula and a Formula Set) *Given a direction relevance relation $\mathcal{R}$, we can extend it to a relation $\mathcal{R}^+$ on a formula and a formula set, i.e., $\mathcal{R}^+ \subseteq \mathbf{L} \times \mathcal{P}(\mathbf{L})$ as follows:*

$$\langle \phi, \Sigma \rangle \in \mathcal{R}^+ \textit{iff there exists a formula } \psi \in \Sigma \textit{ such that } \langle \phi, \psi \rangle \in \mathcal{R}.$$

We have implemented the prototype of the informed bottom-up approach. The prototype is called DION, which stands for a Debugger of Inconsistent Ontologies. DION uses a DIG data format as its internal data represenation format. Therefore, in the following, we define a direct relavance relation which is based on the ontology language DIG.

In DIG, concept axioms has only the following three forms: $impliesc(C_1, C_2)$, $equalc(C_1, C_2)$, and $disjoint(C_1, \cdots, C_n)$, which corresponds with the concept implication statement, the concept equivalence statement, and the concept disjoint statement respectively.

Given a formula ϕ, we use $C(\phi)$ to denote the set of concept names that appear in the formula ϕ.

Definition 3.1.4 (Direct concept-relevance) *An axiom ϕ is directly concept-relevant to a formula ψ, written $SynConRel(\phi, \psi)$, iff*
(i) $C_1 \in C(\psi)$ if the formula ϕ has the form $impliesc(C_1, C_2)$,
(ii) $C_1 \in C(\psi)$ or $C_2 \in C(\psi)$ if the formula ϕ has the form $equalc(C_1, C_2)$,
(iii) $C_1 \in C(\psi)$ or $\cdots$ or $C_n \in C(\psi)$ if the formula ϕ has the form $disjoint(C_1, \cdots, C_n)$.

Definition 3.1.5 (Direct concept-relevance to a set) *A formula ϕ is concept-relevant to a formula set Σ iff there exists a formula $\psi \in \Sigma$ such that ϕ and ψ are directly concept-relevant.*

For a terminology $\mathcal{T}$ and a concept c, we can define a selection function s in terms of the direct concept relevance as follows:

Definition 3.1.6 (Selection function on concept relevance)
(i) $s(\mathcal{T}, c, 0) = \{\psi \mid \psi \in \mathcal{T}$ and ψ is directly concept-relevant to $c\}$;
(ii)$s(\mathcal{T}, c, k) = \{\psi \mid \psi \in \mathcal{T}$ and ψ is directly concept-relevant to $s(\mathcal{T}, c, k-1)\}$ for $k > 0$.

In order to do so, we extend the definition of direct concept relevance so that we can say something like an axiom ψ is direcly concept-relevant to a concept c, i.e., $SynConRel(\psi, c)$. It is easy to see that it does not change the definition of direct relevance relation.

Algorithms We use an informed bottom-up approach to obtain MUPS. In logics and computer science, an increment-reduction strategy is usually used to find minimal inconsistent sets[8]. Under this approach, the algorithm first finds a set of inconsistent sets, then reduces the redundant axioms from the subsets. Similarlly, a heuristic procedure for finding MUPS consists of the following three stages:

Algorithm 3.1: Algorithm for $mups(\mathcal{T}, c)$

```
$k := 0$
$mups(\mathcal{T}, c) := \emptyset$
**repeat**
  $k := k + 1$
**until** $c$ unsatisfiable in $s(\mathcal{T}, c, k)$
$\Sigma := s(\mathcal{T}, c, k) - s(\mathcal{T}, c, k-1)$
**for all** $\Sigma' \in \mathcal{P}(\Sigma)$ **do**
  **for all** $\phi \in \Sigma/\Sigma'$ **do**
    **if** $\Sigma' \cup \{\phi\} \notin mups(\mathcal{T}, c)$ **then**
      $\Sigma'' := s(\mathcal{T}, c, k-1) \cup \Sigma'$
      **if** $c$ satisfiable in $\Sigma''$ and $c$ unsatisfiable in $\Sigma'' \cup \phi$ **then**
        $mups(\mathcal{T}, c) := mups(\mathcal{T}, c) \cup \{\Sigma'' \cup \{\phi\}\}$
      **end if**
    **end if**
  **end for**
**end for**
$mups(\mathcal{T}, c) := \mathbf{MinimalityChecking}(mups(\mathcal{T}, c))$
**return** $mups(\mathcal{T}, c)$
```

- **Heuristic Extension**: Using a relevance-based selection function to find two subsets Σ and S such that a concept c is satisfiable in S and unsatisfiable in $S \cup \Sigma$.
- **Enumeration Processing**: Enumerating subsets S' of S to obtain a set $S' \cup \Sigma$ in which the concept c is unsatisfiable. We call those sets *c-unsatisfiable sets*.
- **Minimality Checking**: Reducing redundant axioms from those c-unsatisfiable sets to get MUPSs.

The following is an algorithm for MUPSs. The algorithm first finds two subsets Σ and S of $\mathcal{T}$. Compared with $\mathcal{T}$, the set Σ is relatively small. The algorithm then tries to exhaust the powerset of Σ to get c-unsatisfiable sets. Finally, by the minimality checking it obtains the MUPSs. We can define the minimality checking as a sub-procedure as shown in the algorithm 3.2.

The complexity of the algorithm 3.1 is exponent to $|\Sigma|$. Although Σ is much smaller than $\mathcal{T}$, it is still not very useful in the implementation. One of the improvement is to do pruning. We can check the subsets of Σ with increasing cardinality of the subsets. Namely, we can always pick up the subsets with a less cardinality first, (i.e., the power set of Σ is sorted). First, set the cardinality $n = 1$, namely pick up only one axiom ϕ in Σ, check if c is unsatisfiable in $\{\phi\} \cup s(\mathcal{T}, c, k-1)$. If 'yes', then it ignores any superset S such that $\{\phi\} \subset S$. After all of the subsets with the cardinality n have been checked, increase n by 1. Moreover, we can do checking and pruning during the powerset is built.

Algorithm 3.2: Algorithm for the minimality checking on $mups(\mathcal{T}, c)$

```
for all $\Sigma \in mups(\mathcal{T}, c)$ do
  $\Sigma' := \Sigma$
  for all $\phi \in \Sigma'$ do
    if $c$ unsatisfiable in $\Sigma' - \{\phi\}$ then
      $\Sigma' := \Sigma' - \{\phi\}$
    end if
  end for
  $mups(\mathcal{T}, c) := mups(\mathcal{T}, c)/\{\Sigma\} \cup \{\Sigma'\}$
end for
return $mups(\mathcal{T}, c)$
```

That leads to the algorithm 3.3 in which we use the set S to book the c-satisfiable subsets.

Proposition 3.1.1 (Soundness of the Algorithms MUPS) *The algorithms for MUPS above are sound. Namely, they always return MUPSs.*

PROOF. It is easy to see that the concept c is always unsatisfiable for any element S in the set $mups(\mathcal{T}, c)$. Otherwise it is never added into the set. The minimality condition is achieved by the procedure of the minimality checking. Therefore, the algorithms for MUPSs are sound. □

Take our running example. To calculate $mups(\mathcal{T}_1, A_1)$, the algorithm first gets the set $\Sigma = \{ax_2\} = \{ax_1, ax_2\} - \{ax_1\}$. Thus, $mups(\mathcal{T}_1, A_1) = \{\{ax_1, ax_2\}\}$. We can see that the algorithm cannot find that $S_1 = \{ax_1, ax_3, ax_4, ax_5\} \in mups(\mathcal{T}_1, A_1)$. However, it does not change the result of MIPS, because we have a subset $\{ax_3, ax_4, ax_5\}$ of S_1, which will be in $mups(\mathcal{T}_1, A_3)$. To calculate $mups(\mathcal{T}_1, A_6)$, the algorithm first gets the set $\Sigma = \{ax_2, ax_5\} = \{ax_1, ax_2, ax_3, ax_4, ax_5, ax_6\} - \{ax_1, ax_3, ax_4, ax_6\}$. Thus, $mups(\mathcal{T}_1, A_6) = \{\{ax_1, ax_3, ax_4, ax_5, ax_6\}\}$. Again, the algorithm cannot find that $\{ax_1, ax_2, ax_4, ax_6\} \in mups(\mathcal{T}_1, A_6)$. However, it does not affect MIPS, because $\{ax_1, ax_2\} \in mups(\mathcal{T}_1, A_1)$. Therefore, the algorithms for MUPSs proposed above are sound, but not complete. As we have argued above, this informed bottom-up approach is efficient for inconsistent ontology diagnosis.

Sometimes it is useful to find just a single MUPS by using the relevance relation without referring to a selection function. Algorithm 3.4 uses the increment-reduction strategy to find a single MUPS for an unsatisfiable concept, without a selection function. The algorithm finds a subset of the ontology in which the concept is unsatisfiable first, then reduces the redundant axioms from the subset.

Algorithm 3.3: Algorithm for $mups(\mathcal{T}, c)$ with pruning

```
$k := 0$
$mups(\mathcal{T}, c) := \emptyset$
**repeat**
  $k := k + 1$
**until** $c$ unsatisfiable in $s(\mathcal{T}, c, k)$
$\Sigma := s(\mathcal{T}, c, k) - s(\mathcal{T}, c, k - 1)$
$S := \{s(\mathcal{T}, c, k - 1)\}$
**for all** $\phi \in \Sigma$ **do**
  **for all** $S' \in S$ **do**
    **if** $c$ satisfiable in $S' \cup \{\phi\}$ and $S' \cup \{\phi\} \notin S$ **then**
      $S := S \cup \{S' \cup \{\phi\}\}$
    **end if**
    **if** $c$ unsatisfiable in $S' \cup \{\phi\}$ and $S' \cup \{\phi\} \notin mups(\mathcal{T}, c)$ **then**
      $mups(\mathcal{T}, c) := mups(\mathcal{T}, c) \cup \{S' \cup \{\phi\}\}$
    **end if**
  **end for**
**end for**
$mups(\mathcal{T}, c) := \mathbf{MinimalityChecking}(mups(\mathcal{T}, c))$
**return** $mups(\mathcal{T}, c)$
```

We can obtain MUPSs for all unsatisfiable concepts. Based on those MUPSs, we can calculate MIPS, core, and pinpoints further, by using the standard algorithms which have been discussed in the previous chapter. Those data can be used for knowledge workers to repair the ontology to avoid unsatisfiable concepts [25, 26].

3.2 Calculating terminological diagnoses

Terminological diagnosis, as defined in [24], is an instance Reiter's diagnosis from first principles. Therefore, we can use Reiter's algorithms to calculate terminological diagnoses. What is required is a method to produce conflict sets, and we will discuss three different options for this. Let us first recall the basic methodology from [24]. Here, we simplify the technical details in order to make the presentation slightly more intuitive.

Given an incoherent terminology $\mathcal{T}$, a *conflict set* is any incoherent subset of $\mathcal{T}$.[3] Then, Reiter showed, that a set $\Delta \subseteq \mathcal{T}$ is a diagnosis for an incoherent terminology iff Δ is a minimal set such that $\mathcal{T} \setminus \Delta$ is not a conflict set for $\mathcal{T}$.

The basic idea to calculate diagnoses from conflict sets is based on minimal hitting

[3]A related notion can be defined for unsatisfiability of concepts w.r.t. a terminology. For readability reasons, we restrict ourselves to diagnosis of incoherence in terminologies.

Algorithm 3.4: Finding a MUPS for $\mathcal{T}$ in which a concept c is unsatisfiable

```
$\Sigma := \emptyset$
**repeat**
  **for all** $\phi_1 \in \mathcal{T} \setminus \Sigma$ **do**
    **if** $SynConcRel(\phi_1, c)$ or there is a $\phi_2 \in \Sigma$ such that $SynConRel(\phi_1, \phi_2)$ **then**
      $\Sigma := \Sigma \cup \{\phi_1\}$
    **end if**
  **end for**
**until** $c$ is unsatisfiable in $\Sigma$
**for all** $\phi \in \Sigma$ **do**
  **if** $c$ is unsatisfiable in $\Sigma - \{\phi\}$ **then**
    $\Sigma := \Sigma - \{\phi\}$
  **end if**
**end for**
```

sets. Suppose C is a collection of sets. A *hitting set* for C is a set $H \subseteq \bigcup_{S \in C} S$ such that $H \cap S \neq \varnothing$ for each $S \in C$. A hitting set is minimal for C iff no proper subset of it is a hitting set for C.

This gives the basis of Reiter approach to calculate diagnoses given the following theorem which is a direct consequence of Corollary 4.5 in [22].

Theorem 3.2.1 A set $\Delta \in \mathcal{T}$ is a diagnosis for an incoherent terminology $\mathcal{T}$ iff Δ is a minimal hitting set for the collection of conflict sets for $\mathcal{T}$.

To calculate minimal hitting trees Reiter introduces hitting set trees (HS-trees). For a collection C of sets, a HS-tree T is the smallest edge-labeled and node-labeled tree, such that the root is labeled by $\checkmark$ if C is empty. Otherwise it is labeled with any set in C. For each node n in T, let $H(n)$ be the set of edge labels on the path in T from the root to n. The label for n is any set $S \in C$ such that $S \cap H(n) = \varnothing$, if such a set exists. If n is labeled by a set S, then for each $\sigma \in S$, n has a successor, n_σ joined to n by an edge labeled by σ. For any node labeled by $\checkmark$, $H(n)$, i.e. the labels of its path from the root, is a hitting set for C.

Figure 3.3 shows a HS-tree T for the collection $C = \{\{1,2,3,4,5,6\}\{3,4,5\},\{1,2,4,6\},\{1,2\},\{4,7\}\}$ of sets. T is created breadth first, starting with root node n_0 labeled with $\{1,2,3,4,5,6\}$. For diagnostic problems the sets in the collection are conflict sets which are created on demand. In our case, conflict sets for a terminological diagnosis problem can be calculated by a standard DL engine (by definition each incoherent subset of $\mathcal{T}$ is a conflict set).

These calls are computationally expensive, which means that we have to minimize them. In Figure 3.3, those nodes are boxed, for which labels were created by calls to the prover. T reuses already calculated and smallest possible labels, and is pruned in a variety of ways, which are defined in detail in [22]. Just for example, node n_0 is relabeled with a subset $\{3, 4, 5\}$ of its label. We denote by $1^\times$, that element 1 is deleted. Note, that no successor for this element has to be created. Node n_6 has been automatically labeled with $\{4, 7\}$, because the intersection of its path $h(n_6) = \{2, 3\}$ is empty with an already existing conflict set in the tree.

Three ways of implementing diagnosis The generality of Reiter's algorithm has the advantage of giving some leeway for particular methodological choices. We implemented three ways of calculating conflict sets.

1. Use an optimized DL reasoner to return a conflict set in each step of the creation of the HS-tree. The only way to get conflict sets for an incoherent TBox $\mathcal{T}$ is to return $\mathcal{T}$ itself, i.e. the *maximal conflict set*.
2. Use an adapted DL reasoner to return *small conflict sets*, which it can derive from the clashes in a tableau proof.
3. Use a specialized method to return *minimal conflict sets*, e.g., using the algorithms of [25].

Diagnosis with maximal conflict sets The most general way to calculate terminological diagnosis based on hitting sets is to use one of the state-of-the-art optimized DL reasoner. The advantage is obvious: the expressiveness of the diagnosis is only restricted by the expressiveness of the DL reasoning implemented in the reasoner. We use RACER, which allows to diagnose incoherent terminologies up to $\mathcal{SHIQ}$ without restriction on the structure of the TBox. The algorithm to use RACER is simple: if $\mathcal{T}$ is incoherent, return $\mathcal{T}$, other return $\varnothing$. As RACER is highly optimized we can expect to get the maximal conflict sets efficiently.

The disadvantage of this naive approach is that the conflict sets are huge, and even with reusing of node labels and pruning, the HS-tree become quickly to large to handle. Take the incoherent TBox $\mathcal{T}^*$ where the related HS-tree already has 380 nodes, and needs 67 calls to RACER. We will see that the price we pay for the gain in expressiveness is too high, and that smaller conflict sets are required.

Diagnosis with small conflict sets The disadvantage of using a DL reasoner as a black-box is that they do not provide any information on which components contribute to the incoherence. Technically, this means which axioms contribute to the closure of the tableau. To show that already straightforward collecting of clash-enforcing axioms can dramatically improve the efficiency of diagnosis, we implemented a simple tableau calculus for unfoldable $\mathcal{ALC}$ TBoxes. This reasoner returns an unordered, and not necessarily minimal, list of axioms which are (indirectly) responsible for the clashes in the tableau. The basic idea is to label each formula with a set of axioms, which are added to a formula in

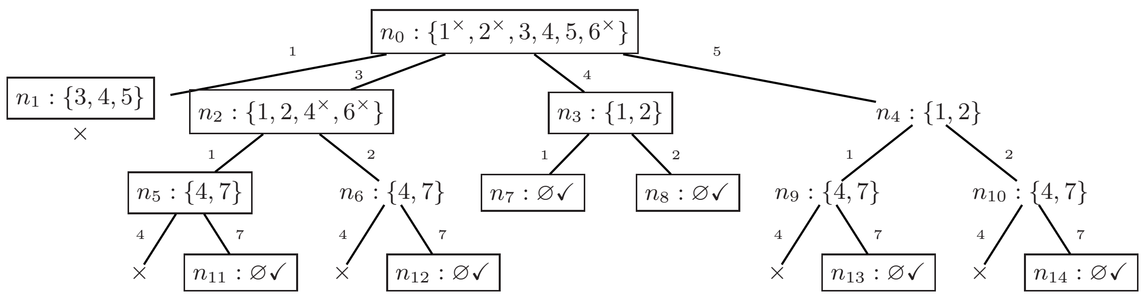

Figure 3.3: HS-Tree with small conflict sets

the tableau whenever they are used to “unfold” a defined concept. This algorithm is not optimized, but returns small conflict sets, and the sizes of the HS-Trees decrease dramatically. Figure 3.3 shows the hitting tree for the incoherence problem for $\mathcal{T}^*$ where small conflict sets have been collected from tableau proofs. Compared to the previous method, there were only 14 nodes created, and 11 calls to the DL reasoner necessary.

Diagnosis with minimal conflict sets Previously, we recalled the notion of minimal unsatisfiability (and incoherence) preserving sub-terminologies MUPS and MIPS, which were introduced in [25] for the debugging of terminologies.

The MUPSs of an incoherent terminology $\mathcal{T}$ and an unsatisfiable concept A are the minimal conflict sets of this unsatisfiability problem. It is easily checked that each MUPS $\{\{ax_1, ax_2\}, \{ax_1, ax_3, ax_4, ax_5\}\}$ for A_1 and $\mathcal{T}^*$ is indeed a minimal conflict set. This time, only 12 nodes were created. Based on the MUPS, it is straightforward to calculate MIPS, which are the minimal conflict sets for the incoherence problem.

Proposition 3.2.1 The MIPS of an incoherent terminology $\mathcal{T}$ are the minimal conflict sets for this incoherence problem.

Figure 3.4 shows the hitting tree for the incoherence problem of $\mathcal{T}^*$ where minimal conflict sets have been calculated as MIPS.[4]

[4]An axiom ax_i is represented by the number i.

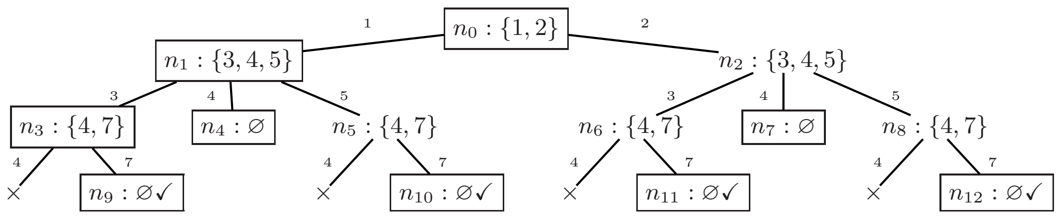

Figure 3.4: HS-Tree with minimal conflict sets

Chapter 4

A qualitative evaluation of the framework

4.1 Two motivating examples

To demonstrate the use of our debugging approach, we first take a well-described ontology. We use the Pizza ontology from the Protégé OWL tutorial[1]. Next, we will discuss the use of our approach to an extended version of the pizza ontology, which has a larger number of unsatisfiable concepts.

Pizza ontology This pizza ontology purposely contains two unsatisfiable concepts, IceCream and CheeseyVegetableTopping. Whereas the causes of the unsatisfiability are realistic, this ontology is a special case because the concepts that are unsatisfiable are fully unrelated, and the unsatisfiability does not propagate to other concepts.

$mups(Pizza,$ IceCream) = $\{$ IceCream $\sqsubseteq$ DomainConcept $\sqcap$ $\exists$ hasTopping. FruitTopping , disjoint(IceCream, Pizza), role hasTopping :domain Pizza $\}$

$mups(Pizza,$ CheeseyVegetableTopping) = $\{$ CheeseyVegetableTopping $\sqsubseteq$ CheeseTopping $\sqcap$ VegetableTopping , disjoint (CheeseTopping, VegetableTopping) $\}$

As the MUPS for both concepts contain exactly one set of axioms, the diagnoses for unsatisfiability of the concepts are the individual axioms:

$\Delta_{Pizza,IceCream}$ = $\{$ $\{$IceCream $\sqsubseteq$ DomainConcept $\sqcap$ $\exists$ hasTopping. FruitTopping $\}$, $\{$disjoint (IceCream, Pizza)$\}$, $\{$role hasTopping :domain Pizza$\}$ $\}$

$\Delta_{Pizza,CheeseyVegetableTopping}$ = $\{$ $\{$CheeseyVegetableTopping $\sqsubseteq$ CheeseTopping $\sqcap$ VegetableTopping $\}$, $\{$disjoint (CheeseTopping, VegetableTopping) $\}$ $\}$

It can be easily determined that

[1] `http://www.co-ode.org/ontologies/pizza/2005/05/16/`

$mips(Pizza)$ = $\{$ $\{$IceCream $\sqsubseteq$ DomainConcept $\sqcap$ $\exists$ hasTopping. FruitTopping , disjoint (IceCream, Pizza) , role hasTopping :domain Pizza$\}$, $\{$CheeseyVegetableTopping $\sqsubseteq$ CheeseTopping $\sqcap$ VegetableTopping , disjoint (CheeseTopping, VegetableTopping) $\}$ $\}$

and

Δ_{Pizza} = $\{$ $\{$IceCream $\sqsubseteq$ DomainConcept $\sqcap$ $\exists$ hasTopping. FruitTopping , CheeseyVegetableTopping $\sqsubseteq$ CheeseTopping $\sqcap$ VegetableTopping $\}$, $\{$IceCream $\sqsubseteq$ DomainConcept $\sqcap$ $\exists$ hasTopping. FruitTopping , disjoint (CheeseTopping, VegetableTopping) $\}$, $\{$disjoint (IceCream, Pizza) , CheeseyVegetableTopping $\sqsubseteq$ CheeseTopping $\sqcap$ VegetableTopping $\}$, $\{$disjoint (IceCream, Pizza) , disjoint (CheeseTopping, VegetableTopping) $\}$, $\{$role hasTopping :domain Pizza , CheeseyVegetableTopping $\sqsubseteq$ CheeseTopping $\sqcap$ VegetableTopping $\}$, $\{$role hasTopping :domain Pizza, disjoint (CheeseTopping, VegetableTopping) $\}$ $\}$

For the generalized MIPS, only the definition of IceCream can be generalized, the other definitions remain unchanged.

$gmips(Pizza)$ = $\{$ $\{$IceCream$'$ $\sqsubseteq$ $\exists$ hasTopping. FruitTopping$'$, disjoint(IceCream$'$, Pizza$'$) , role hasTopping :domain Pizza$'$ $\}$, $\{$CheeseyVegetableTopping$'$ $\sqsubseteq$ CheeseTopping$'$ $\sqcap$ VegetableTopping$'$, disjoint (CheeseTopping$'$, VegetableTopping$'$) $\}$ $\}$

As all MIPS are equivalent to MUPS, they all have a MIPS-weight of 1.

As there are no axioms that occur in both of the MIPS, there are only MIPS-cores of arity 1. As a result, any diagnosis is also a pinpoint.

This example shows that for this ontology the MUPS are a useful reduction of the ontology as a whole to small sets of concepts. The other measures are of limited interest, due to the isolation of the unsatisfiable concepts and the absence of propagation of unsatisfiability.

Extended Pizza ontology Suppose that within the Pizza ontology also 5 subsumees of IceCream were defined, IceCream$_1$ … IceCream$_5$ and three subsumees of Pizza, CheeseyVegetablePizza$_1$ … CheeseyVegetablePizza$_3$, which are defined as CheeseyVegetablePizza$_i$ $\sqsubseteq$ Pizza $\sqcap$ $\exists$ hasTopping. CheeseTopping $\sqcap$ $\forall$ hasTopping. VegetableTopping.

Now, the resulting ontology, which we will call $Pizza'$ will have 10 unsatisfiable concepts, IceCream, CheeseyVegetableTopping, plus the eight newly defined concepts.

Then the following MUPS are added:

$mups(Pizza'$, IceCream$_i)$ = $\{$ IceCream$_i$ $\sqsubseteq$ IceCream , IceCream $\sqsubseteq$ DomainConcept $\sqcap$ $\exists$ hasTopping. FruitTopping , disjoint (IceCream, Pizza) , role hasTopping :domain Pizza $\}$

$mups(Pizza'$, CheeseyVegetablePizza$_i)$ = $\{$ CheeseyVegetablePizza$_i$ $\sqsubseteq$ Pizza $\sqcap$ $\exists$ hasTopping. CheeseTopping $\sqcap$ $\forall$ hasTopping. VegetableTopping , disjoint (CheeseTopping, VegetableTopping) $\}$

Additional MIPS can now be calculated for the extended ontology. The $mips$ for the new ontology is:

$mips(Pizza')$ = $\{$ $\{$IceCream $\sqsubseteq$ DomainConcept $\sqcap$ $\exists$ hasTopping. FruitTopping , disjoint (IceCream, Pizza) , role hasTopping :domain Pizza $\}$,
$\{$CheeseyVegetableTopping $\sqsubseteq$ CheeseTopping $\sqcap$ VegetableTopping , disjoint (CheeseTopping, VegetableTopping) $\}$,
$\{$CheeseyVegetablePizza$_1$ $\sqsubseteq$ Pizza $\sqcap$ $\exists$ hasTopping. CheeseTopping $\sqcap$ $\forall$ hasTopping. VegetableTopping , disjoint (CheeseTopping, VegetableTopping) $\}$,
$\{$CheeseyVegetablePizza$_2$ $\sqsubseteq$ Pizza $\sqcap$ $\exists$ hasTopping. CheeseTopping $\sqcap$ $\forall$ hasTopping. VegetableTopping , disjoint (CheeseTopping, VegetableTopping) $\}$,
$\{$CheeseyVegetablePizza$_3$ $\sqsubseteq$ Pizza $\sqcap$ $\exists$ hasTopping. CheeseTopping $\sqcap$ $\forall$ hasTopping. VegetableTopping , disjoint (CheeseTopping, VegetableTopping) $\}$ $\}$

For the sake of brevity, we will not present the diagnoses and the generalized mips for this example.

The MIPS-weight for $\{$IceCream $\sqsubseteq$ DomainConcept $\sqcap$ $\exists$ hasTopping. FruitTopping , disjoint (IceCream, Pizza), role hasTopping :domain Pizza $\}$ is now 6. The other MIPS have a weight of 1.

The core with the highest arity is now the axiom “disjoint (CheeseTopping, VegetableTopping”), which has arity 4.

This core is used as the starting point for constructing a pinpoint. We can combine this core with an axiom from the first MIPS. A pinpoint for this ontology now is: $\{$disjoint (CheeseTopping, VegetableTopping), disjoint (IceCream, Pizza)$\}$.

In this example, we see that MIPS-weights and the cores indicate axioms that lead to unsatisfiability of more concepts.

4.2 Applying the framework to real-life terminologies

In the previous section we presented the results of the approach on ontologies that were specifically designed to demonstrate unsatisfiability. The outcomes demonstrated that the cause(s) for unsatisfiability were clearly pinpointed.

We now present results from the implementation and application of the approach to the real-world medical ontologies DICE [6] and FMA [23].

The current implementation of our approach can handle unfoldable TBoxes in $\mathcal{ALC}$ [24]. An unfoldable TBox implies that all definitions are simple (defining only atomic concepts), unique (only one definition for each atomic concept exists), and acyclic (meaning the definition of a concept has no reference to the definiendum, either directly or indirectly). The language $\mathcal{ALC}$ is a description logic where the allowed constructors are the ones mentioned in Section 2.

Our implementation partly uses RACER to perform reasoning, and partly implements algorithms to calculate $mups$, $mips$, $gmips$, and the heuristics. These algorithms are implemented in Java, so it can run on the platforms that are supported by RACER (currently available for 32bit versions of Windows, Linux or Mac OS X, for Sun and other branded UNIX workstations with 32bit or 64bit, as well as 64bit Linux environments).

DICE The DICE knowledge base[2], which is under development at the Academic Medical Center in Amsterdam, contains about 2500 concepts. Each concept is described in both Dutch and English by one preferred term, and any number of synonym(s) for each language. In addition to about 1500 reasons for admission, DICE contains concepts regarding anatomy, etiology and morphology.

DICE originally has a frame-based representation, and is migrated to DL in order to be able to perform auditing w.r.t. incorrect definitions and missed classification. In the DL-based representation many closure axioms are used in order to be able to find incorrect definitions. These closure axioms include disjointness of sibling concepts, and universal restrictions. As a result of the migration process, various concepts become unsatisfiable.

A recent version of DICE was classified, resulting in 65 unsatisfiable concepts. For one concept, calculation of MUPS failed in the current implementation, for yet unknown reasons. For the remaining 64 concepts, a total of 175 MUPS are found.

142 MIPS are found, with the following distribution of MIPS-weights:
MIPS of weight 1: 121
MIPS of weight 2: 10
MIPS of weight 3: 0
MIPS of weight 4: 11

The pinpoint of this ontology consisted of the following five axioms, which form the largest cores, according to the procedure defined in Section 2.

Core of arity 60: Disjointness of children of “Act”
Core of arity 56: Disjointness of children of “Dysfunction/Abnormality”
Core of arity 15: Disjointness of children of “System”
Core of arity 7 (43 in the full $mips$): “Heart valve operations”
Core of arity 4: Disjointness of children of “Toxical substance”

The arity of 7 for “Heart valve operations” is the arity in the remaining $mips$ after removal of all MIPS containing the disjointness statements mentioned earlier. The arity of this core in the full $mips$ is 43.

Based on these results one can determine where to start the debugging process. Either the disjointness statements mentioned can be verified, or one can further analyze the definition of and references to the concept “Heart valve operations”.

We will discuss the concept “Heart valve operations”, also in order to demonstrate the use of generalized MIPS.

The concept is defined as:

[2]Development of DICE is supported by the National Intensive Care Evaluation (NICE) foundation.

$$
\begin{aligned}
\text{HeartValveOperations} \sqsubseteq\ & \text{HeartProcedures} \sqcap \\
& \exists\ \text{HasSystemInvolvement. CirculatorySystem} \sqcap \\
& \forall\ \text{HasSystemInvolvement. CirculatorySystem} \sqcap \\
& \exists\ \text{LocalizedIn. HeartValveStructure} \sqcap \\
& \forall\ \text{InvolvesDysfunction. (Thrombosis} \sqcup \text{Insufficiency} \sqcup \text{Stenosis)} \sqcap \\
& \exists\ \text{InvolvesAct. Replacement} \sqcap \\
& \exists\ \text{InvolvesAct. Resection} \sqcap \\
& \exists\ \text{InvolvesAct. Repair} \sqcap \\
& \exists\ \text{InvolvesAct. Excision} \sqcap \\
& \exists\ \text{InvolvesAct. Inspection} \sqcap \\
& \forall\ \text{InvolvesAct. (Replacement} \sqcup \text{Resection} \sqcup \\
& \qquad\qquad \text{Repair} \sqcup \text{Excision} \sqcup \text{Inspection)}
\end{aligned}
$$

As mentioned earlier, the DL-based representation is generated from a frame-based representation [5]. In this migration process, it was decided that universal restrictions were added as default, as is shown by the $\forall$ constructors in the definition.

Inspection of this statement reveals incorrect semantics: five values for the InvolvesAct role are required, whereas generally these operations involve one of these acts. After correcting this modeling error, which resulted from an incorrect assumption in the migration process, we can reapply our approach. The resulting ontology still has 62 unsatisfiable concepts, resulting in 111 MIPS.

Now, the pinpoint only contains disjointness axioms. However, the definition for Heart valve operations is still a core with an arity of 12, so we continue to focus on that concept. One MIPS contains "Valve commissurotomy" (the definition of which contains $\exists$ InvolvesAct. Incision) and disjointness of Incision and Replacement, Resection, Repair, Excision, and Inspection. This indicates that the definition of Heart valve operation should be adjusted to include Incision in the disjunctions of the $\forall$ InvolvesAct restriction.

In this way, we can iterate the process of making changes to the ontology and determining $mups$ and $mips$. Alternatively, we could have focused on specific unsatisfiable concepts, using the $mups$.

FMA To test how our approach can be applied to other ontologies, we have used the Foundational Model of Anatomy (FMA)[3]. FMA, developed by the University of Washington, provides about 69000 concept definitions, describing anatomical structures, shapes, and other entities, such as coordinates (left, right, etc.). The FMA Knowledge Base, which is implemented as a frame-based model in Protégé[4], has been migrated to DL.

Due to its large size we were not able to classify the full FMA ontology with RACER. We hence limited the case study to "Organs", which comprises a convenient subset that is representative for the FMA. Of the 3826 concept definitions, 181 were found to be

[3]`http://sig.biostr.washington.edu/projects/fm/`
[4]`http://protege.stanford.edu/`

unsatisfiable. Interestingly, this resulted in the single pinpoint “Organ”. This could be explained by the definition of Organ:

$$\begin{aligned}\text{Organ} \sqsubseteq\ & \text{AnatomicalStructure} \sqcap \\ & \exists\ \text{RegionalPartOf. OrganSystem} \sqcap \\ & \forall\ \text{RegionalPartOf. OrganSystem} \sqcap \\ & \exists\ \text{PartOf. OrganSystem} \sqcap \\ & \forall\ \text{PartOf. OrganSystem}\end{aligned}$$

In FMA, the unsatisfiable concepts were defined as part of some organ, for example

$$\begin{aligned}\text{Periodontium} \sqsubseteq\ & \text{SkeletalLigament} \\ & \exists\ \text{RegionalPartOf. Tooth} \sqcap \\ & \forall\ \text{RegionalPartOf. Tooth} \sqcap \\ & \exists\ \text{PartOf. Tooth} \sqcap \\ & \forall\ \text{PartOf. Tooth} \sqcap \\ & \exists\ \text{SystemicPartOf. Tooth} \sqcap \\ & \forall\ \text{SystemicPartOf. Tooth}\end{aligned}$$

Periodontium and Tooth are subsumed by Organ, and according to the definition of Organ, Tooth should be an OrganSystem. Hence, it would be more correct to specify that an Organ is also an allowed value for the (Regional)PartOf role, i.e. defining Organ as follows:

$$\begin{aligned}\text{Organ} \sqsubseteq\ & \text{AnatomicalStructure} \sqcap \\ & \exists\ \text{RegionalPartOf. (Organ} \sqcup \text{OrganSystem)} \sqcap \\ & \forall\ \text{RegionalPartOf. (Organ} \sqcup \text{OrganSystem)} \sqcap \\ & \exists\ \text{PartOf. (Organ} \sqcup \text{OrganSystem)} \sqcap \\ & \forall\ \text{PartOf. (Organ} \sqcup \text{OrganSystem)}\end{aligned}$$

This example clearly shows how one axiom can lead to unsatisfiability of a large number of concepts. The pinpoint properly detects this single axiom.

4.3 Qualitative evaluation

The implementation of our approach provides a useful contribution to the debugging process. The heuristics (especially the Pinpoint) provide a good starting point for debugging of the axioms. MUPS and generalized MIPS provide understandable explanations for causes of unsatisfiability. In our experiments, performing the calculations for the MUPS, MIPS and heuristics was a matter of minutes (on a 2.4 GHz PC with 1 GB memory).

The current implementation provides text-based output. This requires a modeler to browse the output, and find for example the MUPS that are related to a MIPS. Apart from further optimizing the algorithms and supporting more expressive logics, the various steps in the debugging process can be further integrated to provide more support.

Debugging of ontologies gets increasing attention, which is driven by research in the area on the semantic web on the one hand, and the need for robust ontologies on the other hand. We will first look into truth maintenance systems, a research field in artificial intelligence that has not yet been discussed in this paper. These systems provide, at least theoretically, a possibility for explaining reasoning. Next we will describe two implementations of other approaches to explanation, Swoop/Pellet and the OWL-debugger plug-in for Protégé. These implementations differ in the approach followed. The Pellet reasoner, described later in this section, uses a “glass-box” approach, that offers a debugging mode in which explanation is provided as a result of the reasoning process. The Protégé OWL-debugger, like our approach, uses a black-box approach. It looks for common modeling errors as explanation for inconsistency, as is described in last Section of this Chapter.

Truth Maintenance Systems Truth maintenance systems (TMSs) [9] are designed to keep track of inferences made in knowledge bases. For example, the implication $P \Rightarrow Q$ might have been used to add Q to a knowledge base that contains P. One approach is a justification-based truth maintenance system (JTMS). In such a system, each sentence in the knowledge base is provided with a justification consisting of the set of sentences from which it was inferred. In the example above, Q is provided with the justification $\{P, P \Rightarrow Q\}$.

TMSs generally serve three distinct purposes: handling retraction of (incorrect) information, speeding up analysis of multiple hypothetical situations, and providing a mechanism for generating explanations. In the example above, the justification provided with Q is also an explanation for why Q holds. Should P be retracted (for example because it was found that P does not hold), then from the justifications it can be determined that Q should also be retracted, as P provided a justification for Q. Similar to our diagnoses, these explanations should be minimal, i.e. no proper subset of an explanation should also be an explanation.

As discussed in [2], a problem with implementing truth maintenance systems in DL reasoners is the fact that efficiency of the highly optimized tableaux-based reasoning algorithms may suffer, as is also experienced in Pellet. Therefore, a black-box approach providing post-hoc explanation is a reasonable alternative to truth maintenance.

Explanation in Pellet One of the most elaborate implementations of debugging known to the authors is the Pellet reasoner, combined with the SWOOP web ontology editor[5], which is described in [20],[17]. It provides a combination of black-box and glass-box

[5]`http://www.mindswap.org/2005/debugging/`

approaches.

Glass-box techniques are used to support two forms of debugging of unsatisfiable concepts: presenting the root cause of the contradiction and determining the relevant axioms in the ontology that are responsible for the clash (the so-called minimal sets of support).

The black-box methods focus on detecting dependencies between unsatisfiable classes. Two types of unsatisfiable classes are recognized: root classes, and derived classes. A root class is defined as an unsatisfiable class in which a clash or contradiction found in the class definition (axioms) does not depend on the unsatisfiability of another class in the ontology. A derived Class is an unsatisfiable class in which a clash or contradiction found in a class definition depends on the unsatisfiability of another class.

The advantage of the Pellet/Swoop environment is that debugging clues are presented in an integrated way. A drawback of the glass-box techniques is that it makes debugging dependent to a specific reasoner. A comparison between the results of debugging with the use of Swoop and Pellet is planned to be performed.

Explanation by the Protégé OWL-debugger Another example of debugging support is the OWL-debugger[6], which is a plug-in for Protégé [28, 27]. This debugger provides a black-box approach, and is heuristic as it is based on experience, from which a set of commonly made mistakes have been identified. Being based on heuristics, the debugger is not complete, but it provides explanation to a majority of cases of unsatisfiability.

The process performed by the OWL-debugger resembles that of the Swoop/Pellet approach. First, an unsatisfiable core is identified, consisting of the smallest set of conditions of a concept that render that concept unsatisfiable. A number of rules is then applied to the unsatisfiable core, in order to determine the debugging super conditions (DSC), which include all concepts that are involved in the unsatisfiable core, for example all superconcepts. A most general conflicting class set is generated from the DSC, which is used to produce an explanation for unsatisfiability.

The OWL-debugger also presents explanation in an integrated way, as part of the Protégé ontology modeling environment. As it is a fully black-box implementation, it is independent of the reasoner being used.

[6]`http://www.co-ode.org/community/debugging.php`

Chapter 5

Principals of a quantitative evaluation

In Chapter 4 we presented a qualitative discussion of the framework for debugging and diagnosis, which was primarily based on our own experience with incoherent terminologies in a medical domain. In principle, these results were encouraging, as we claim that that the MUPS, MIPS, diagnoses and the like, do indeed help. What remains to investigate is whether we have effective algorithms to calculate those debugging operators, in which cases a particular method will be more appropriate than another, and how much can indeed stretch our method in practice. In the next two chapters we attempt to answer these questions given some practical experiments, in which we will run our tools MUPSter and DION against a number of benchmarks. First, however, let us discuss which research questions we want to answer based on such a benchmark, and which basic requirements this benchmark need to fulful.

5.1 Evaluating algorithms for debugging and diagnosis

Even though our experience showed that our approach to debugging and diagnosis is useful in practice, we regularly encountered computational problems in the day to day application. In this report we want to get a better understanding of the challenge, the state-of-the-art of our tools, and the differences between the methods. Basically, there are a number of questions:

- *Can debugging be performed efficiently?* More concretely, given an incoherent terminology, can our tools support a practitioner effectively?
- *What makes an incoherent terminology difficult to debug?* We will see that the answer to the previous question is not necessarily positive. Even worse, we know that for most Description Logics our problem is exponentially hard[1], which means

[1]Debugging is at least as hard as checking satisfiability, which is itself PSPACE for most DLs (see. e.g., Chapter 3 of [1]).

that we will never be able to guarantee termination in realistic time for any arbitrary input. There might be particular classes of terminologies that are more difficult than others, and it is important to study these difficulties to improve the methods.

- *Which are the most appropriate methods to calculate?* Not only will there be classes that are more difficult than others, there will also be classes of TBoxes for which one method is more appropriate than the other. To answer this question might allow the user to choose the appropriate implementation according to his/her needs.

Based on these research questions, there are a number of criteria for a good test-set. Most importantly, a test-set should be prototypical, so that it represents a larger class or realistic problems. Also, it has to be systematical, so that influence of particular properties of classes of TBoxes can be evaluated. Finally, a test-set should be statistically viable, i.e the results of an experiment for a particular class of TBoxes should indeed have some significance w.r.t. the properties it is meant to evaluate.

Moreover there are some criteria that are inherent to our evaluation, most importantly, that we are restricted by the evaluated tools to incoherent, unfoldable $\mathcal{ALC}$ TBoxes.

5.2 Three types of benchmark experiments

In order to fulfill the above described criteria, and to be able to address the asked research questions we propose (and conducted) three different types of benchmark experiments, first, an evaluation of the methods with real-world terminologies, secondly, an evaluation using an adapted benchmark set from the DL literature, and finally, some experiments with our own purpose-build data-set.

5.2.1 Evaluation with real-world terminologies

The ideal case of an evaluation is to use a number of The first approach to evaluation of debugging methods is to consider a number of publicly available Description Logic terminologies.

We split our test terminologies in three groups, ordered by how they were built. As examples for terminologies created through migration we consider an older version of the anatomy fragment of DICE (we abbreviate DICE-A), with 534 axioms and 76 unsatisfiable concepts, and a previous full version of full DICE (abbreviated DICE). The incoherence of DICE-A has two distinct causes: first, this is a snap-shot from the terminology in its creation process, i.e. it contains real modeling errors. Moreover, the high number of contradictions is specific for migration as a result of stringent semantic assumptions. MGED and Geo are variants of ontologies which are incoherent because they have disjointness statements artificially added for semantic enrichment (as suggested in [26]). MGED provides

	#ax	#unsat	#mips	|mips|	length of mD
DICE-A	534	76	16	3	3
DICE	4995	27	55	4	-
MGED	406	72	38	4	3
Geo	417	11	22	2.6	8
S&C	6382	923	-	-	-
MadC	69	1	-	-	1
	1	2	3	4	5

Table 5.1: Real-world terminologies

standard terms for the annotation of micro-array experiments to enable structured queries on those experiments; and Geo an ontology of geography made available by the Teknowledge Corporation. The third category contains the merged terminologies of SUMO and CYC, two well-known upper ontologies. As they are topic-related, and as CYC provides disjointness statements, there is a high number of unsatisfiable concepts.

We constructed simplified $\mathcal{ALC}$ versions for all five terminologies. Without loss of unsatisfiability, we removed, for example, numerical constraints, role hierarchies and instance information. All terminologies, however, were non-cyclic and could be transformed to an unfoldable format.

For the last example, the MadC ontology, this is not the case. This ontology was constructed to illustrate language features of Description Logics, and we use it to illustrate Reiter's generic method works for expressive formalisms, where both other methods fail. MadC is incoherent with unsatisfiable concept *MadCow*.

Benchmarking with real-life ontologies is obviously a most natural way of evaluating the quality of debugging algorithms. On the other hand, there is only a limited amount of realistic terminologies available that are incoherent. This has several reasons, first, published ontologies usually have undergone a careful modeling process, and it should be expected that logical modeling errors have already been eliminated (without the help of automatic methods). Secondly, current ontologies often still use quite inexpressive languages, for example, and avoid inconsistencies by not stating disjointness of classes. The consequence is that the set of testing examples is limited, and it becomes difficult to make a systematic evaluation of our algorithms as the bias of the data is simply too dominant.

As an alternative it is common to use systematically created test-sets for benchmarking, and such sets also exists to evaluate Description Logic reasoning.

5.2.2 Benchmarking with (adapted) existing test-sets

The issue of benchmarking Description Logic systems has been addressed exhaustively in the literature over the last 10 years, mostly using adaptations of modal logic test-sets, [18]. In most of these studies the purpose was to evaluate the runtime of DL reasoner with respect to the complexity of the used language (mostly the modal depths). As they were often planned to include comparisons with SAT- [10] or resolution-based systems [16] most of the resulting test-sets are for concept satisfiability.

Unfortunately, we are bound by the expressiveness of our tools for the benchmarking, which essentially means to restrict the experiments to test-sets with unfoldable $\mathcal{ALC}$ terminologies. Finding a good benchmark for unfoldable terminologies ontologies is difficult, though, as existing extensions to ontologies usually go beyond these requirements.

For this reason, we adapted an existing test-set for $\mathcal{ALC}$ concept satisfiability, by translating each unsatisfiable concept into an incoherent terminology. For this purpose, we used the satisfiability tests from the well-known DL benchmark from the 1998 system comparison. We took 9 sets of unsatisfiable concepts usually denoted by: DL98=$\{$k_branch_p, k_d4_p, k_dum_p, k_grz_p, k_lin_p, k_path_p, k_ph_p, k_poly_p, k_t4p_p$\}$. The test procedure works as follows: each set contains 21 concepts with exponentially increasing computational difficulty. The measure of the speed of the DL system was then the highest concept in the list that could still be solved in 100 seconds. These test-sets were build in the early days of the latest generation of Description Logic tools, which did not have as many optimisations then as they have now. Nowadays, these test-sets are outdated, as they are too easily solved by all existing DL systems. For our purpose, however, they are still relevant, as the implemented top-down work (in the current version) without optimisations.[2]

We translated each of the unsatisfiable concepts in the test-sets into one unfoldable incoherent $\mathcal{ALC}$ terminology in the following way: let C be an unsatisfiable concept, we build an initial terminology $A_C \sqsubseteq C$. Then each sub-concept S that is in the scope of an odd number of negations is (recursively) replaced by an atom A_S, and an axiom $A_S \dot{\sqsubseteq} S$ is added to the terminology, where A_S and A_C are new names not occurring in the concept. Then the resulting TBox is incoherent. Let us illustrate the method with an example, rather than give a formal definition. Suppose we have an unsatisfiable concept $C = \exists r.(A \sqcup B) \sqcap \forall r.(\neg A \sqcap \neg B)$. We first build a terminology $\mathcal{T} = \{A_C \sqsubseteq C\}$, and replace the outermost subformulas of C by atoms A_1 and A_2. $\mathcal{T}$ is now $\{A_C \sqsubseteq A_1 \sqcap A_2, A_1 \dot{\sqsubseteq} \exists r.(A \sqcup B), A_2 \dot{\sqsubseteq} \forall r.(\neg A \sqcap \neg B)\}$. Next we transform the definitions of A_1 and A_2, which leads to the following TBox $\{A_C \sqsubseteq A_1 \sqcap A_2, A_1 \dot{\sqsubseteq} \exists r.A_3, A_2 \dot{\sqsubseteq} \forall r.A_4, A_3 \dot{\sqsubseteq} A \sqcup B, A_4 \dot{\sqsubseteq} \neg A \sqcap \neg B\}$,

[2]There is an experimental implementation, which trades completeness of the method with a number of optimisations. In this paper we restrict our attention to the more robust, and complete, standard implementation of MUPSter.

and so on. The resulting terminology for $C = \exists r.(A \sqcup B) \sqcap \forall r.(\neg A \sqcap \neg B)$ is then:

$$\begin{array}{rcl} \{A_C & \dot{\sqsubseteq} & A_1 \sqcap A_2, \\ A_1 & \dot{\sqsubseteq} & \exists r.A_3, \\ A_2 & \dot{\sqsubseteq} & \forall r.A_4, \\ A_3 & \dot{\sqsubseteq} & A_5 \sqcup A_6, \\ A_4 & \dot{\sqsubseteq} & A_7 \sqcap A_8, \\ A_5 & \dot{\sqsubseteq} & A \\ A_6 & \dot{\sqsubseteq} & B \\ A_7 & \dot{\sqsubseteq} & \neg A \\ A_8 & \dot{\sqsubseteq} & \neg B\} \end{array}$$

For each of the sets of unsatisfiable concepts in DL98 we created an incoherent terminology given the above mentioned method for the first three concepts. The resulting terminologies are called: k_branch_p_tbox1, k_branch_p_tbox2, k_branch_p_tbox3 and similarly for all other sets in DL98.

An interesting phenomenon given this new test-set is that the data-sets are heavily engineered to make reasoning hard, and *to punish non-optimised reasoning*. We will see that this has drastic effects on the run-times of the different algorithms. On the other hand, this reasoning-centered view does not really coincide with average structure of the realistic terminologies, that are out in practise. Mostly these terminologies are relatively flat in structure, but large; and often the definitions strongly dependent on other definitions. To account for this, we decided to build our own benchmark for evaluating algorithms for debugging.

5.2.3 Benchmarking with purpose-built test-sets

In order to build a test-set for benchmarking our algorithms and implementations for debugging and diagnosis several basic requirements have to be fulfilled: first, the resulting TBoxes have to be unfoldable and in $\mathcal{ALC}$, and they have to be incoherent. Also, as discussed in the beginning of this Chapter, the benchmark should be systematically constructed, so that classes of problems can be studied and general statements over properties of TBoxes can be made.

These basic requirements leave a number of choices to create such a test-set: e.g. creating an incoherent terminology could be achieved through systematic construction of logical contradictions, or through random choice of operators and names. The first choice has been made in the previously mentioned DL98 benchmark set, whereas in our test-set we opted for the second option. This way we hope to get a stronger similarity with realistic terminologies, while still retaining some control over parts of the structure of the TBoxes.

Building such a test-set for debugging and diagnosis of incoherent terminologies is difficult, because there is a plethora of parameters that could influence the complexity of reasoning, and the difficulty for debugging. In our case we decided to fix a number of parameters, and vary others.

Fixed Parameters The fixed parameters are those parameters which will not change in any setup of the experimental analysis, i.e. for all terminologies we consider in the experiments, we can consider the same properties. The most important fixed parameters are the constant term-depth of the concept definitions, the fixed ratio between modal and Boolean operators, and the ratio between defined and primitive concepts, and the restriction to atomic negation.

- *Constant term-depth.* The term-depths of a concept is the maximal number of operators that an atomic concept is "away from" under the scope of. More formally, the term-depth $td(C)$ of an $\mathcal{ALC}$ concept C is defined recursively as follows:

  $$\begin{array}{rl} td(C) = 0 & \text{if } C \text{ is an atom} \\ td(C) = max(td(C_1), td(C_2)) & \text{if } C = C_1 \sqcup C_2 \\ td(C) = max(td(C_1), td(C_2)) & \text{if } C = C_1 \sqcap C_2 \\ td(C) = td(C_1) + 1 & \text{if } C = \exists r.C_1 \\ td(C) = td(C_1) + 1 & \text{if } C = \forall r.C_1 \\ td(C) = td(C_1) + 1 & \text{if } C = \neg C_1 \end{array}$$

  We say that a concept has constant term-depth, if $td(C_1) = td(C_2))$ for all subconcepts $C_1 \sqcap C_2$ and $C_1 \sqcup C_2$. Intuitively, this implies that all paths of the term representation of a concepts to the leaves are of the same length.

  In logical terms this is not really a restriction, as for every $\mathcal{ALC}$ concept there is a logically equivalent concept with constant term-depth.

- *Ratio between modal and Boolean operators.*[3] When creating a concept definition we decided create boolean and modal operators with the same probability. This corresponds more or less to our experience with existing ontologies. Though they often contain more modal quantifiers than Booleans the latter are not binary, which is the way we create our concepts.

- *Ratio between primitive and defined concept names.* In an unfoldable TBox a defined concept name is one that occurs on the left-hand side of an axiom. Semantically, the interpretation of the defined concepts can be derived from the interpretation of the base interpretation of the primitive concepts. For this test-set we decided to have the same number of primitive and defined concept names. Evaluating the influence of this parameter might be subject to further experiments but was out of the scope of this research.

- *Restriction to atomic negation.* It is well known that it is possible to translate each $\mathcal{ALC}$ concept into an equivalent one in Negation Normal Form (NNF) in linear time,

[3]By modal operators we understand the Box- and Diamond-like DL operators $\exists$ and $\forall$.

i.e. a concept where negations only occur atomically. This means that this technical restriction does not really have an influence of the benchmarking, it should make no significant difference whether debugging and diagnosis is performed on TBoxes in NNF or not.

Variable Parameters The variable parameters will be those parameters we will do experimental evaluation over, i.e. the will influence properties of the resulting TBoxes, which we believe can have a significant influence on the practical complexity. For the current experiments we have four such variable parameters, the size of the TBox, the concept-length, the ratio of disjunctions versus conjunctions, and the chance of an atom being negated or not.

- *Size of the TBox.* We vary the size of the TBox from 10 to 200 axioms. This number corresponds to the number of defined concepts.
- *Concept length.* As previously mentioned we restrict ourselves to TBoxes with constant term-depths. In our experiments we will consider concept definitions of term-depth 3 to 7. Intuitively, this means that each concept-name occurs in the scope of 3 to 7 operators.
- *Ratio of disjunction versus conjunction.* As discussed before, when creating a concept there is a 50% chance that the operator is Boolean. We then have to determine whether this operator will be a disjunction or a conjunction. In our experiments we vary this ratio from 10% to 30% up to 50%.[4]
- *Ration of negated versus non-negated atoms.* Once the full term-depth is reached in our creation process, an atomic concept or its negation has to be created at random. In the experiments we vary the probability of an atom to be negated from 30 to 70%.

Building a set of benchmark TBox Finally, we have to describe how to build an unfoldable $\mathcal{ALC}$ terminology. Suppose we have fixed the variable parameters TBox size to #t with concept size to #s. To create the i-th axiom (i.e. to defined the i-th concept-name) we create a concept of term-depth #s from all the primitive concept-names, and the remaining concept-names that were not define in the first $i-1$ axioms. The construction of the concept is then simply a construction based on random choice with the given probability distribution.

Given this method, we constructed 1611 unsatisfiable terminologies. Note that this method does not automatically deliver incoherent TBox. On the contrary, the overall rate of incoherence given this method is less than 30%. To consider only incoherent TBoxes

[4]We originally considered a similar variation of percentages for the ration between the choice of existential versus universal quantifier, but this seemed to give no new insights.

we therefore simply run an optimised DL reasoner to delete all coherent TBoxes from the testset.

It should be noticed, the choice of parameters greatly influences the ratio of satisfiable versus unsatisfiable terminologies, particularly the *disjunction/conjunction* ratio. More concretely, given the likelihood that a disjunction was chosen in 50% of the cases, only 5% of the resulting TBoxes were incoherent, as opposed to almost 50% of the TBoxes when the likelihood was just 10%. The consequence of this is easy to see: assume that we decide to create a fixed number of TBoxes for testing per parameter value, we will have only 10% of incoherent TBoxes in our test-set with high disjunction ratio. The most simple solution to this problem was to account for the satisfiability/unsatisfiability ratio, and to create an accordingly larger number of TBoxes in the first place. In this way we believe to have created a test-set of TBoxes, where each class of TBoxes corresponding to a particular parameter choice is as likely to occur in the TBox as any other. Both the test-set and the generator will be made available on our website.

Chapter 6

Experimental evaluation

In the previous chapter, we described three different test-sets we use for evaluating the run-times, first, a set of 6 real-life ontologies we collected from the Internet, secondly, an extension of an existing benchmark for evaluation of Description Logic systems, and finally, a purpose build benchmark consisting of 1611 systematically constructed incoherent terminologies. Interestingly, the results vary a lot, and it is worth looking into some details to get a better understanding of the pros and cons of each of the algorithms.

6.1 Experiments to evaluate algorithms for debugging

The main notion for debugging are the minimal unsatisfiability and incoherence preserving sub-terminologies, the MUPS (of a TBox and an unsatisfiable concept) and MIPS (of a TBox). Calculating MIPS can be done quite easily from MUPS. However, there are conceptually very different ways of calculating MUPS, and we have implemented two different strategies: a top-down method, which uses a specialised algorithm based on DL tableau, and a bottom-up method which uses heuristics to guide a brute-force enumeration method.

The theoretical complexity of the problem and the algorithms for unfoldable $\mathcal{ALC}$-TBoxes is known. First, finding MIPS is PSPACE complete: obviously, finding MIPS is at least as hard as checking satisfiability, and there is a simple brute-force way of calculating all the MIPS using only polynomial space in terms of the size of the TBox (trial and error). DION, implements a variant of such an algorithms, whereas MUPSter's algorithm needs exponential space w.r.t. the size of the TBox.[1]

While the MUPSter is the older method, the second one has some significant advantages: because it uses existing purpose-build DL systems for the reasoning part via a `DIG` interface its expressiveness only depends on the underlying prover. At the moment, this

[1]This is because finding prime implicants is NPCOMPLETE w.r.t. a formula, which might need exponential space in relation to the TBox.

	Top-Down: Mupster	Bottom-up: DION	RacerPro
	time for all MIPS	time for all MIPS	time for coherence check
DICE-A	12 sec	timed out	4.3 sec
DICE	54 sec	32 sec	16.62sec
MGED	5 sec	timed out	3.38 sec
Geo	20 sec	11 sec	1.82 sec
Sumo&Cyc	timed out	timed out	4.7 sec
MadCow	not applicable	0.66 sec	0.22 sec
	1	2	3

Table 6.1: Comparing top-down and bottom-up methods

means that DION can debug arbitrary ontologies in very expressive Description Logics, such as $\mathcal{SHIQ}$ and the like. MUPSter, on the other hand, is restricted to unfoldable $\mathcal{ALC}$ TBoxes, and therefore severely less expressive than DION. On the other hand, DION has a seemingly heavy disadvantage, in that its heuristics make it theoretically incomplete. Here, we will try to find out whether there is price for expressiveness or completeness, and whether we can identify cases in which one of the algorithms might be preferable to the other.

6.1.1 Experiments with existing ontologies

The first set of experiments was performed on a dual AMD Athlon MP 2800+ with 2 GB memory. Both MUPSter and DION were given the six terminologies described in Section 5.2.1. Both were timed-out if the programs had not calculated the set of MIPS within one hour.

Results Figure 6.1 summarises the run-times of the two systems on the 6 terminologies in columns 1 and 2. As a reference we also give in column 3 the runtime RacerPro 1.8.1 needs to find the unsatisfiable concepts. For the MadCow terminology, MUPSter could not be applied as this is not an unfoldable terminology.

Whereas MUPSter manages to calculate all the MIPS within the time-limit for all but the Sumo&Cyc terminology, DION also fails in two more cases, the anatomy part of DICE, and the MGED TBox. The results of the experiments show a scalability problem for both algorithms with real big terminologies, as both methods failed to calculate MIPS in the case of Sumo&Cyc, and DION also needs at least more than one hour for two more terminologies. On the other hand, the run-times of DION are faster for the two examples where it finds a solution, even though MUPSter can solve two more problems in the given time.

Finally, it has to be noted, as a proof of concept more than anything else, that DION is indeed capable of calculating MIPS for the more complex MadCow terminology.

Analysis The first conclusion from the experiments can easily be drawn: debugging is computationally difficult, and with the current software it will be likely that calculating MUPS and MIPS will in some cases fail no matter what algorithms is used. On a more optimistic note, however, it is important to note, that solutions can be found in fairly complex terminologies, such as DICE or MGED, even though this cannot be guaranteed in general.

This remark is in line with reasoning in Description Logics in general, where the worse case complexity also suggests that reasoning is in principle intractable. But despite this very high computational complexity in the worse case, there are very promising results in practical complexity (both for classical reasoning and debugging), which are based on powerful optimisations.

Out of the four cases where our tools find solutions, two interesting, and antagonistic, pieces of information stick out: the shorter runtime DION as compared to MUPSter in the two cases where DION finds a solution, and the apparent unrelatedness of the run-times of DION and RacerPro. As DION uses RacerPro as underlying reasoning engine, a correlation could have been expected. However, even though there is almost no difference in the time to check satisfiability for DICE and DICE-A, DION fails to find MIPS for the latter. Similarly, the relatively small run-time of RacerPro on Sumo&Cyc would suggest that DION should be able to find MIPS, which it does not.

One way of explaining this behaviour is to consider the number of unsatisfiable concepts in the 6 ontologies. Here might be a correlation, as both DICE-A and MGED have a very high number of unsatisfiable concepts. As DION implements the calculation of MIPS as a series of calculations of MUPS, it might simply need too many communication steps to the DIG interface to solve the MIPS problem in time.

However, it remains to be investigated in more detail, whether there are other criteria that could be responsible for these relatively surprising results. For this purpose we conducted some experiments on two sets of systematically built benchmarks.

6.1.2 Experiments with existing benchmarks

The analytic value of the previous set of experiments is relatively limited, as the results could be severely influenced by some particularity of the chosen ontologies. Therefore, we conducted a second set of experiments based on the benchmark set from the DL systems competition at DL 98.

The hope of running experiments against a more systematically built set of incoherent TBoxes was to get some better understanding of the influence of specific properties of terminologies on the computational properties of the respective algorithms.

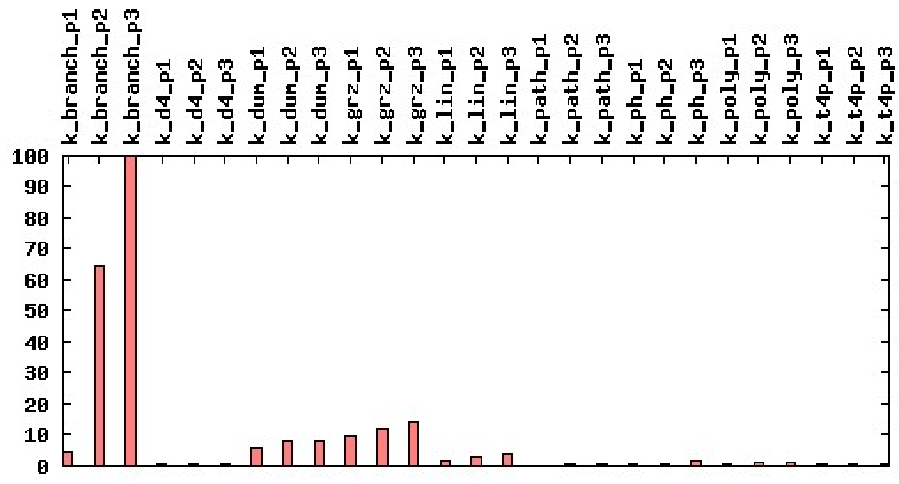

Figure 6.1: Runtime of DION on DL98 testset

To this avail we run both MUPSter and DION against a set of 27 TBoxes, where the translation from the DL98 test-bench of the first three concepts were considered. The results for MUPSter are summarised in Figure 6.2, those of DION in Figure 6.1

The graphics show the runtime per problem up to the time-out of 100 seconds. Hereby, it can be expected that with an increasing number of the description of each problem also the runtime goes up by construction of the problem.

We kept the relatively short time-out of 100 seconds from the original DL experiments. This has to do with the nature of the experiments. Remember that the DL benchmark was created in such a way that it show the weakness of an algorithm in an exponential way, so adding more time will not change anything substantially. DION might solve one class more, and MUPSter over the first two levels in some cases, but the general picture will be the same.

Results The results of the experiments with the adapted DL benchmark set are almost diametric to the results described in the previous section, as they show high failure rate of the MUPSter tool to solve the more difficult cases, but very good computational behaviour of DION.

More concretely, it can be noticed that MUPSter fails in all of the 9 different classes to calculate even the 3 problem (out of a possible 21), whereas DION only fails in a single case (k_branch_p) to terminate the debugging process. However, in 6 out of 9 possible

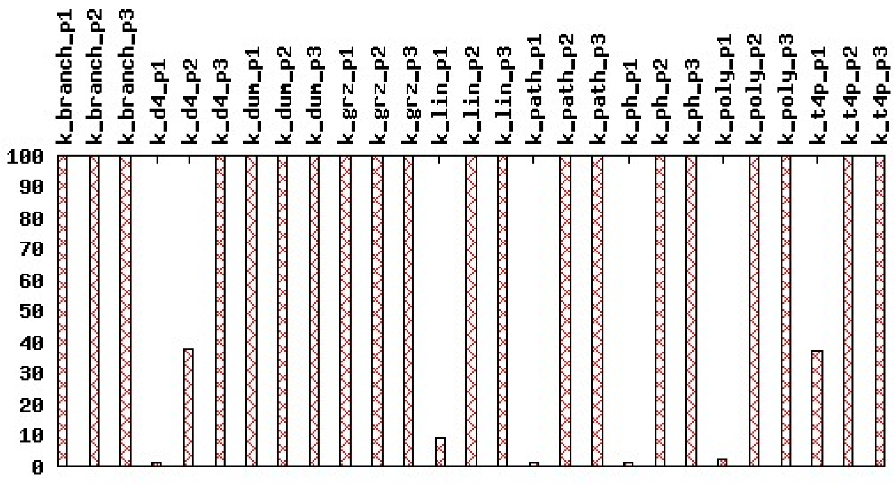

Figure 6.2: Runtime (in sec) of MUPSter on DL98 testset

cases, MUPSter manages to find debugs for the most simple case. These results are in line with the computational difficulty of the problems.

Analysis The results show clearly that MUPSter fails to debug terminologies that are based on complex unsatisfiability problems, such as the ones used in the DL98 evaluation. These problems have been purpose-build to point to computational difficulties in satisfiability checking, i.e. they explicitly exploit structural properties of formulas and provers. More concretely, experience showed that the DL systems of the 90s failed in these cases, because they used naive tableau algorithms, and the complexity of the test-set forced their calculations to be “lost in the search space”.

This allows the conclusion that basic optimisations can significantly improve the performance of MUPSter, because techniques such as lemmatising and caching can avoid the mentioned computational traps. The price in this case is a loss of theoretical correctness. More precisely, the terminologies returned by our algorithm might not be minimal any more. We doubt whether this is a problem in practise, as minimality is a nice, but not strictly required feature of the MIPS.

As DION uses optimised reasoners, which can nowadays solve problems like those of the DL98 test-set within milliseconds, via its DIG interface, it solved the problems easily. This also has to do with the structure of the test-set: as we created the TBoxes from concepts in a deterministic way, each axiom is related to the part of the concept it was created from. But this syntactic structure corresponds one-to-one to DION’s search strategy: in

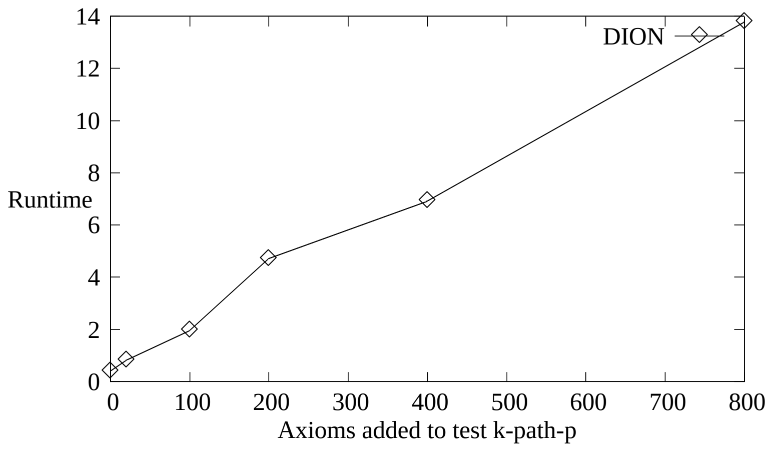

Figure 6.3: Runtime (in seconds) of DION depending on the number of (unrelated) axioms to k_path_p1

DION we basically start with the definition on an unsatisfiable concept, and include the next following axioms in our test procedure on the basis of a syntactic relation (occurrence of the defined name). Each calculation of MIPS therefore mirrors the construction of the TBox from the unsatisfiable concept.

To counter this process we decided, in order to create a more realistic test-set, to add a number of irrelevant axioms to the test-set. We arbitrarily chose the first TBox k_path_p and added 100, 200, 400 and 800 random axioms.[2] Figure 6.3 summarise the run-times of this experiments.

Although the problem is in principle exponential in the size of the TBox, DION shows linear behaviour in this experiment. This also corresponds to the increasing run-time RacerPro needs for checking coherence. This has an relatively simple explanation: it shows that the strategy underlying DION controls the inherent complexity of the search process in a nice ways. But it also poses a question with regard to our first set of experiments with the real-world ontologies. Remember that we had noted there, that the failure of DION to calculate any MIPS for two ontologies was independent of the computational complexity to find unsatisfiable concepts (as shown by RacerPro).

One reason for this discrepancy could be that the created TBoxes have a very particular structure, as they usually only have a single unsatisfiable concepts. A more detailed study of the original test-set is needed to be able to properly assess the reasons for those differences.

[2]We took arbitrary coherent sets of axioms from our purpose-build test-set.

6.1.3 Experiments with purpose-build benchmark

The final set of experiments to evaluate algorithms of debuggers was conducted on our own purpose built benchmark that was described in Section 5.2.3. Here the focus was not on creating difficult reasoning problems, but on identifying features of terminologies that would make debugging hard or easy. Furthermore, we intended our constructed TBoxes to be as close as possible to the structures of realistic terminologies.

In these experiments we constructed classes of TBoxes according to some varying parameters, and compared the runtimes of our tools for these classes with other classes. This allows us to get some more insights into the influence of structural peculiarities of TBoxes on the time needed for debugging.

The parameters we varied were the ratio of negated versus unnegated atoms, the number of disjunctions versus conjunctions, and the size of the concepts and of the TBox.

We will first summarise the results, before trying to provide some explanations, and a critical discussion, for the sometimes unexpected results.

Results For the 1611 TBoxes tested in these experiments we checked the run-times of the three tools DION and MUPSter for debugging, and RacerPro for consistency checking. Again, the experiments were performed on a dual AMD Athlon MP 2800+ with 2 GB memory. The overall average run-times in seconds are summarised in the following table:

	MUPSter	DION	RacerPro
average in sec	2.05	1.68	1.35

These numbers show DION as the clear overall winner over MUPSter, which is in line with the results of the previous experiments. It is worth noticing, that the overall runtimes of MUPSter contain a call to RacerPro for a list of unsatisfiable concepts. Since this external run-time takes on average about 70% of MUPSter's runtime, we expect a strong correlation between RacerPro and MUPSter for those cases where MUPSter takes reasonably short time. Any difference in the run-time behaviour of these two tools therefor points to an additional source of complexity in MUPSter's core algorithm.

On the other hand, DION's algorithm is completely based on calls to an external reasoner, in our experiments RacerPro. Therefore, also a correlation between the times of these tools can be expected.

The results of our experiments to identify computational properties of particular classes of TBoxes are summarised in a number of figures, which we will discuss in the following. In each of the following figures we show the average run-time in seconds for each tool given a particular instantiation of values for the varying features.

Figure 6.4 shows the runtime (in seconds) of the three tools given a varying ratio of 30 to 70% of negated versus non-negated atoms in the axioms of the TBox. There is an

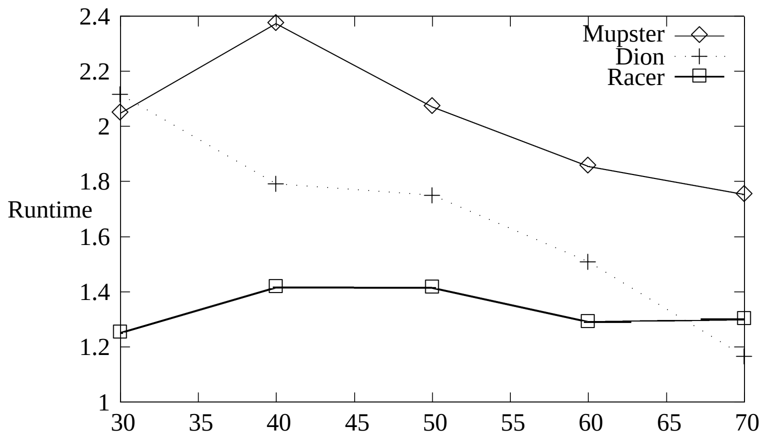

Figure 6.4: Average runtime (in seconds) depending on the ratio negative/positive atoms (in %)

interesting discrepancy between the run-time of RacerPro and MUPSter, on the one side, and DION on the other, as the former peak on a ratio of 40%, whereas the latter has its highest value in the class of TBoxes with a probability of 30% that the atoms are negated.

This is a surprising result, and it might be worth to remember the construction of the test-set. It is easy to see that the probability of TBoxes constructed with a negation ratio of 40 or 50% are more likely to be incoherent. This has also been shown in some of our secondary experiments. However, as we only consider incoherent TBoxes we would not expect the difference between coherence and incoherent to influence the run-time behaviour. Moreover, the run-times of Racer do not support the view that reasoning for the class of TBoxes with 30% negated atoms might be harder than for the other cases.

A similar result can be seen in the study of the ratio between disjunctions and conjunctions. Again, there is an almost equivalent behaviour of MUPSter and RacerPro, and a diverging pattern for DION. Here, the difference between lower and higher number of disjunctions has significant influence on the runtimes of DION, much more than for Racer and MUPSter.

Much more in line with expectation is the following experiment, where we vary the size of the concepts in the definitions. Figure 6.6 summarises the results, where we compare the run-times of our three tools on classes of TBoxes with concept-length 3 to 7.

In the case of varying concept-size, the result shows a clear correlation between the three methods. There is a slight dip in computational difficulty of the concepts of size 4, followed by an increasing run-time for all tools. Interestingly, the curve for DION is the steepest, and for concepts of size 7, DION already takes more time than MUPSter.

The most natural results can be seen in the experiments were the size of the TBox is varied. Here, we take TBoxes of different length into account varying from 10 to 100 in

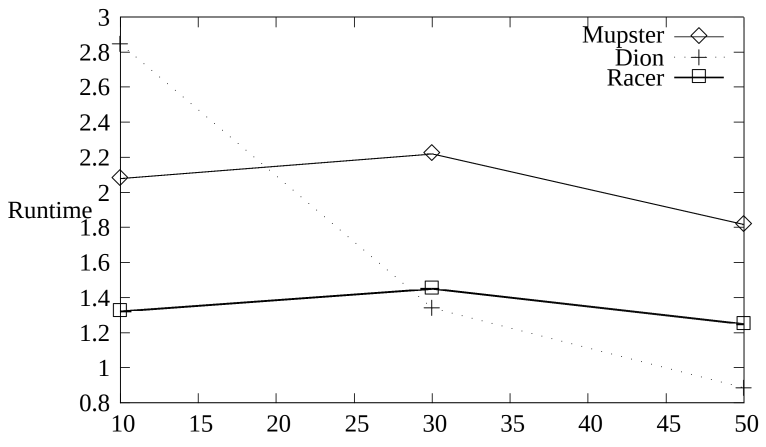

Figure 6.5: Average runtime depending on the disj/conj ratio

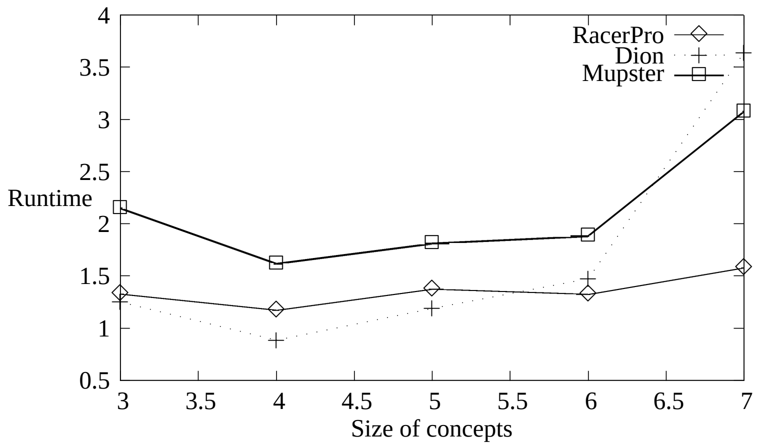

Figure 6.6: Runtime (in seconds) depending on the size of the concepts

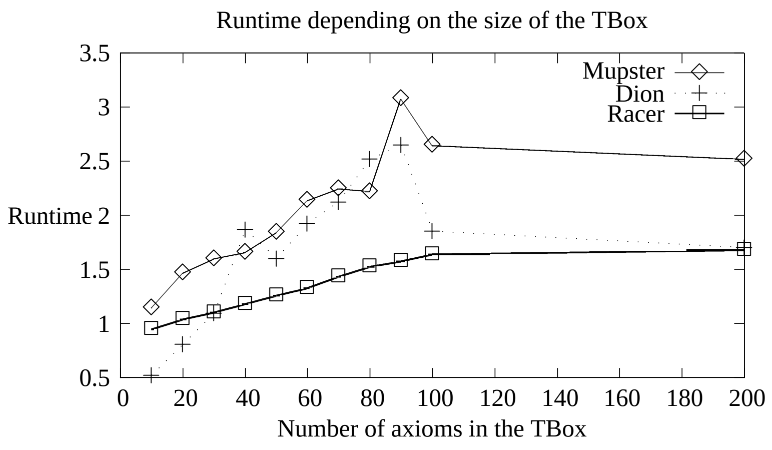

Figure 6.7: Runtime (in seconds) depending on the size of the TBox

steps of 10, comparing it with a single value for TBoxes of size 200. Here the general line is an almost linear increase in run-time for all three tools up to 90 concepts, followed by a slight decrease toward longer TBoxes.

Analysis Unfortunately, none of the initial experiments seem to indicate a clear explanation for the run-time behaviour of particular classes of TBoxes. Even worse, almost each experiments shows some very odd results.

In the two experiments where we vary the disjunction and negation ratios DION's run-times are not in line with either MUPSter or RacerPro. There are some possible explanations, either these results correspond to some (unknown) properties of the DION algorithm or implementation or to some peculiarities in our test-data. More experiments (with a more detailed analysis of the benchmark set) are needed, before we can explain this behaviour.

More conclusive are the next two experiments, in which a clear tendency can be seen that debugging becomes more difficult with increasing TBox and concept-size. From these two experiments, another conclusion might be drawn. Figure 6.6 shows an increase in the runtime of DION which is much steeper than those of the other two tools. This has its reasons in the DION algorithm, where the size of the search tree is determined by the length of the concepts of the definitions (because of the choice of selection function). The experiment described in Figure 6.7 suggests that the increase in complexity of the TBox is controlled by the heuristics in a better way.

All the previously described experiments depend on the features of the TBoxes that are known previous to debugging. This could be useful if we want to decide automatically which tool to use for which terminology. A second class of features are those that are inherent to debugging, i.e., which can only be determined by running a debugger on the

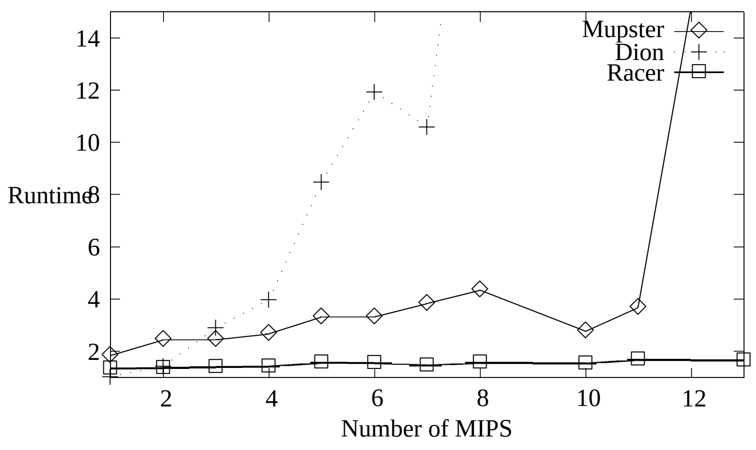

Figure 6.8: Runtime (in seconds) depending on the number of MIPS

data-sets. One such feature of our TBoxes that might be relevant to better understand the runtime behaviour of our three tools is the number of MIPS for an incoherent terminology.

Figure 6.8 shows the runtimes of the three tools with respect to the number of MIPS of the incoherent terminologies. The results are highly interesting, as they differ strongly from the previous ones, and show a very distinct behaviour for each of the three tools. Whereas RacerPro's runtime remains almost constant for increasing MIPS-size, and MUPSter shows an almost linear increase (up to the number 11), is there a clearly exponential behaviour visible for DION.

For the values above 10, the numbers have to be read with care, as there are too few examples to be statistically valid (only 6 in total). Nevertheless the data is interesting, as it shows clearly DION's difficulties to deal with incoherent TBoxes with large numbers of MIPS.

	Name of TBox	Run-time	Run-time	incoherence
#MUPS		DION	MUPSter	RacerPro
13	tbox_60_7_1_3_7_v1	74.59 sec	26.40 sec	1.64 sec
11	tbox_40_7_1_3_6_v1	46.07 sec	4.76 sec	1.38 sec
11	tbox_80_7_1_5_5_v1	38.33 sec	2.58 sec	1.97 sec
10	tbox_40_6_1_3_6_v1	15.65 sec	1.84 sec	1.20 sec
10	tbox_80_5_1_4_7_v1	10.18 sec	2.67 sec	1.30 sec
10	tbox_90_7_1_6_7_v1	46.67 sec	3.79 sec	2.10 sec

How can we explain the increasing run-time of the debugger, particularly of DION, for an increasing number of MIPS? Both MUPSter and DION implement the same method to calculate MIPS from MUPS. As an increasing number of MIPS is directly related to an increasing number of MUPS, the difference can be found here: remember that MUPSter calculates all MUPS at the same time from a fully expanded tableau. This means that calculating several MUPS is not more difficult than calculating fewer MUPS, as the initial time-consuming act is the expansion of the tableau. However, the number of MUPS is directly related to the breadth of the search-tree of DION, which adds one exponential factor for each additional branch in the tree.

Remember, that the number of MIPS is also correlated with the number of “modelling errors”, in the sense that it is more likely that there are more MIPS if there are more erroneous axioms. This could explain the odd behaviour of DION for the class of TBoxes with the smallest number (10%) of disjunctions as described in Figure 6.5. For concepts with fewer branches, there will be more possible contradictions than for concepts with more disjunctions. This implies that there will be more MIPS in the class of TBoxes with a 10% probability that a Boolean operator is a disjunction, than in the class where the probability is 30% or 50%, which in turn explains the run-time behaviour of DION.

As the latest experiments show, studying the properties of the results of the debug seems to be a fruitful line, which could lead to an explanation of the run-time behaviour of DION on the realistic terminologies studied in Section 6.1.1.

There are a number of open questions, that we have to leave for future research:

- What is the influence of other properties of TBoxes, i.e. the number of MUPS or the average size of MIPS and MUPS.
- What is the relation between the a-posteriori features (#MIPS etc) and the data-set? Can we explain the peculiarities of some results by secondary properties of the benchmark?

In this report we focused on a-priori properties of TBoxes, i.e. properties that can be determined before diagnosis and debugging, because one of our initial research goals was to determine the best tool for a particular class of incoherent terminologies.

Answering the research questions

Let us end this Section by answering the research questions we presented in Section 5.1. More concretely, there are three questions regarding the computational properties of debugging, our tools and particular subclasses of incoherent terminologies, which we can now answer on the basis of the experiments of the previous section.

Can debugging be performed efficiently? Realistic ontologies can be debugged in some, but not all, cases. Overall MUPSter shows better performance on our real-world examples than DION, which probably has to do with the large number of MIPS. On the other hand, the other two benchmark show a more fine-grained picture: given our own benchmark we can conclude that both methods scale quite well with the overall size of the terminology and even the average length of the concepts. Finally, the DL benchmark where complex reasoning is required show that optimised reasoners can be used for efficient (if incomplete) debugging, but also that non-optimised techniques (such as the one employed by MUPSter, which are naive w.r.t. the logical complexity of the reasoning) necessarily fail. It is an open question whether there are any complete approaches that might scale up in this case.

Diagnosis is a much harder problem, as it requires the calculation of all minimal conflict sets and, on top of this, an NPCOMPLETE minimisation procedure.

What makes an incoherent terminology difficult to debug? The larger an incoherent terminology, the more difficult it becomes. It seems that the increase in run-time is linear with increasing TBox size, but exponential in the average size of the concepts. On the other hand, the results of our experiments with our purpose-built benchmark regarding the influence of the ratio of negated versus non-negated atoms and disjunction versus conjunction do not show coherent influence on the run-time, and point to differences in the two computational approaches rather than in the general difficult to debug an incoherent terminology. Finally, the runtime w.r.t. the number of MUPS gives a clear indication that the complexity increases with an increase in the number of modelling errors.

Which are the most appropriate methods to calculate? There are two critical cases: first, the complexity of the standard reasoning is so that MUPSter's unoptimised tableau calculus fails. Then, DION is a good choice, as it uses optimised DL reasoner. In the second case, there are multiple-errors, which might hamper DION's efficiency. In this case, MUPSter can often be more efficient, as it uses a single procedure which is independent on the number of errors. In all other cases, our results seem to indicate that both methods perform comparably.

6.2 Experiments to evaluate terminological diagnosis

With a number of experiments we studied the feasibility of diagnosis. We implemented the three techniques described in the previous section in JAVA,[3] and applied them to the terminologies introduced in Section 5.2.1

Table 6.2 summarizes the quantitative results of diagnosis on the 6 incoherent terminologies introduced above. All experiments were performed on a Pentium III, 1.3.GHz, RedHat Linux.

The first 4 columns summarize information about the terminologies, the number of axioms, unsatisfiable concepts and MIPS, as well as the average size of the MIPS. Column 5 gives the length of the smallest diagnosis, Column 6 the time RACER needs to check for incoherence of the terminology. The diagnostic results are split in three pairs: the first two columns 7 and 8 ((labeled Maximal CS) give

[3]Implementations and test sets will be made publicly available.

	Maximal CS		Small CS		Minimal CS	
	#D/hr	timeD1	#D/hr	timeD1	#D/hr	timeD1
DICE-A	0	-	4	1622 s	27	151 s
MGED	0	-	10	40 s	58	31 s
Geo	-	-	8	114 s	115	62 s
S&C	0	-	0	-	0	-
WINE	6	37 s	/	/	/	/
MadC	4	12 s	/	/	/	/
	1	2	3	4	5	6

Table 6.2: Comparing Diagnosis with different types of conflict sets

- the number of diagnoses calculated per hour (#D/hr), and
- the time to calculate the first diagnosis (timeD1)

based on maximal conflict sets. Similarly for columns 9 and 10, for small, and 11 and 12 for minimal conflict sets (calculated using MIPS). The runtime in column 12 contains calculation of unsatisfiability using RACER, the calculation of the MIPS, and, finally, of the hitting sets.

Analysis The most significant result is the almost complete failure to calculate diagnoses using the naive maximal hitting set approach. Only for the toy examples of the WINE and MadC terminologies are any diagnoses found. The reason for this is the length of the minimal diagnosis and the number of axioms. As all but one axioms belong to the maximal conflict sets, there are (#ax-1) branches at first level, and (#ax-1)*(#ax-2) branches at level 2. To find a diagnosis of length 3, a branch of depth 3 has to be explored, which means, e.g., for DICE-A a total number of 100 million branches. Only small ontologies with small diagnoses can be debugged in this most general way.

Things look better for the other methods. Both detect diagnoses of size up to 8 for large terminologies such as DICE-A or GEO. Again, all depends on the size of the diagnoses and the number of axioms. Still the results were unsatisfactory: there was not a single algorithm that determined any diagnoses for the merged SUMO and CYC ontology, and only once did the algorithm terminate within an hour, namely when checking DICE-A using minimal conflict sets.

For this latter method (based on minimal conflict sets) the computational difficulty lies in the fact that constructing minimal hitting sets from MIPS corresponds to calculating prime implicants for a propositional formula, and is thus an NP-COMPLETE problem. Although our prototypical implementation uses some optimization it is not efficient enough to build and search very large HS-Trees. The intermediate implementation based on small conflict sets could significantly be made faster by using an optimized reasoner to return conflict sets more efficiently. From manual inspection we believe that the size of the conflict sets (and thus the size of the HS-Tree) would not be much smaller, but the time to find the small conflict sets could be significantly lower. In both cases, however, the theoretical (and practical) complexity is very high.

A word of caution It should be mentioned that the run-times of diagnosis and debugging cannot be compared as the experiments were performed on different computers. Also, the evaluation of diagnosis has to be studied with care. Given the algorithms based on HS-Trees diagnoses are calculated with increasing size, the given numbers can be slightly misleading: remember that timeD1 denotes the time needed to calculate the first diagnosis, and #D/hr the number of diagnosis calculated in one hour. The evaluation of diagnosis as presented here is meant as a comparison of the three different methods (based on small, minimal and maximal conflict sets), and not for comparison with debugging. This has to be left for future research.

Chapter 7

Concluding remarks

In previous work we had presented a framework for debugging and diagnosis, which was published as Sekt Deliverable 3.6.2. This research was continued for this report, in which we evaluate the previously defined methods in some detail.

This is done in two ways: first, we first, we study the effectiveness of our proposal in a *qualitative* way with some practical examples. Secondly, in a *quantitative and statistical* analysis, we try to get a better understanding of the computational properties of the debugging problem and our algorithms for solving it.

In the qualititative part we discuss two simple incoherent terminologies to explain the functionality and particular application scenarios of our debugging framework. This framework was then applied to two incoherent terminologies which were used at the Academic Medical Center, Amsterdam, for the admission of patients to Intensive Care units. This evaluation is described in Chapter 4.

For the statistical parts we conducted three sets of experimments in order to evaluate the debugging problem and our algorithms and tools. First, we applied our two debugger DION and MUPSter on a set of real-world terminologies collect from our applications and the WWW. Secondly, we translated an existing test-set for Description Logic satisfiability to an incoherence problem, and finally, we created our own benchmark.

The results are mixed: in general we can conclude that our proposed framework for debugging is useful in practice. Our experience showed that a number of heuristics and additional measures, such as the Generalised MIPS, the weight of MIPS or the pinpoints are crucial in practical applications.

In terms of computational behaviour our results are slightly less positive: for each of the described methods there are relatively simple cases, where debugging fails in reasonable time. For example, the DION tool, which performs slightly better on average, fails to compute MIPS for a number of real-world terminologies, where MUPSter comes up with a solution with minutes. On the other hand, the more complex reasoning gets, the worse is MUPSter's performance.

Our experiments gave us some better understanding of the properties of incoherent terminologies that influence the debugging time, most notably the complexity of the definitions, and the number of logical modeling errors. There are, however, still several open questions, mostly related to the combination of properties. This means that an estimation of the run-time of the debuggers on the basis of the structure of the incoherent terminology is still very difficult.

It has to be mentioned we restrict our attention purely to the evaluation of logical debugging methods. Nowadays, there is much research on conceptual approaches to debugging, which are out of the scope of this paper. Also, our evaluation has some severe technical restrictions, most importantly the restriction to unfoldable $\mathcal{ALC}$ terminologies.

In future research the methods for debugging have to be extended so that they can deal with general ontologies. But the general properties we analysed in this paper will remain the same: debugging will be possible, but computationally difficult. An interesting result is the comparison of the two methods: top-down versus bottom-up, which both have their merits in different cases. This might be an argument to extend the MUPSterapproach to more expressive languages, and non-restricted ontologies.

Bibliography

[1] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[2] Alex Borgida, Enrico Franconi, Ian Horrocks, Deborah McGuinness, and Peter Patel-Schneider. Explaining $\mathcal{ALC}$ subsumption. In Patrick Lambrix, editor, *International Workshop on Description Logics, Linköping, Sweden*, volume 22, pages 37–40. CEUR-WS, 1999.

[3] Luca Console and Oskar Dressler. Model-based diagnosis in the real world: Lessons learned and challenges remaining. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1393–1400. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[4] R. Cornet and A. Abu-Hanna. Evaluation of a frame-based ontology. A formalization-oriented approach. In *Proceedings of MIE2002.*, volume 90, pages 488–93, 2002.

[5] R. Cornet and A. Abu-Hanna. Description logic-based methods for auditing frame-based medical terminological systems. *Artificial Intelligence in Medicine*, 34(3):201–17, 2005.

[6] N. F. de Keizer, A. Abu-Hanna, R. Cornet, J. H. Zwetsloot-Schonk, and C. P. Stoutenbeek. Analysis and design of an ontology for intensive care diagnoses. *Methods of Information in Medicine*, 38(2):102–12, 1999.

[7] J de Kleer and B C Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[8] Maria Garcia de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding all minimal unsatisfiable subsets. In *Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 32–43. ACM, 2003.

[9] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, 1979.

[10] Enrico Giunchiglia and Armando Tacchella. System description: *SAT: A platform for the development of modal decision procedures. In *Conference on Automated Deduction*, pages 291–296, 2000.

[11] R. Greiner, B. A. Smith, and R. W. Wilkerson. A correction to the algorithm in reiters theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[12] Volker Haarslev and Ralf Möller. Description of the racer system and its applications. In *Proceedings of the International Workshop on Description Logics (DL-2001)*, pages 132–141. Stanford, USA, August 2001.

[13] I. Horrocks. The FaCT system. In *TABLEAUX 98*, pages 307–312, 1998.

[14] Z. Huang, F. van Harmelen, and A. ten Teije. Reasoning with inconsistent ontologies. In *Proceedings of the International Joint Conference on Artificial Intelligence - IJCAI'05*, 2005.

[15] Z. Huang, F. van Harmelen, A. ten Teije, P. Groot, and C. Visser. Reasoning with inconsistent ontologies: a general framework. Project Report D3.4.1, SEKT, 2004.

[16] Ullrich Hustadt and Renate Schmidt. MSPASS: Modal reasoning by translation and first-order resolution. In Roy Dyckhoff, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (T ABLEAUX 2000)*, volume 1847 of *LNAI*, pages 67–71. Springer, 2000.

[17] Aditya Kalyanpur, Bijan Parsia, and Evren Sirin. Black box techniques for debugging unsatisfiable concepts. In I. Horrocks, U. Sattler, and F. Wolter, editors, *Proceedings of the International Workshop on Description Logics DL2005, Edinburgh, Scotland, UK*, volume 147. CEUR-WS, 2005.

[18] Fabio Massacci and Francesco M. Donini. Design and results of tancs-2000 non-classical (modal) systems comparison. In *TABLEAUX '00: Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 52–56, London, UK, 2000. Springer-Verlag.

[19] B. Nebel. Terminological reasoning is inherently intractable. *AI*, 43:235–249, 1990.

[20] Bijan Parsia, Evren Sirin, and Aditya Kalyanpur. Debugging OWL ontologies. In *Proceedings of the 14th International World Wide Web Conference (WWW2005), Chiba, Japan*, pages 633–640, 2005.

[21] W.V. Quine. The problem of simplifying truth functions. *American Math. Monthly*, 59:521–531, 1952.

[22] R. Reiter. A theory of diagnosis from first principles. *Artif. Intelligence*, 32(1):57–95, 1987.

[23] Cornelius Rosse and Jose L. V. Mejino, Jr. A reference ontology for biomedical informatics: the foundational model of anatomy. *Journal of Biomedical Informatics*, 36(6):478–500, 2003.

[24] S. Schlobach. Diagnosing terminologies. In Manuela Veloso and Subbarao Kambhampati, editors, *AAAI, Pittsburgh, PA*, 2005.

[25] S. Schlobach and R. Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In *Proceedings of the eighteenth International Joint Conference on Artificial Intelligence, IJCAI'03*. Morgan Kaufmann, 2003.

[26] Stefan Schlobach. Debugging and semantic clarification by pinpointing. In A. Gómez-Pérez and J. Euzenat, editors, *ESWC, Heraklion, Greece*, volume 3532, pages 226–240. Springer, 2005.

[27] Hai Wang, Matthew Horridge, Alan Rector, Nick Drummond, and Julian Seidenberg. Debugging OWL-DL ontologies: A heuristic approach. In Brahmananda Sapkota, editor, *Proceeding of the 4th International Semantic Web Conference (ISWC2005), Galway, Ireland*. Springer, 2005.

[28] Hai Wang, Matthew Horridge, Alan Rector, Nick Drummond, and Julian Seidenberg. A heuristic approach to explain the inconsistency in OWL ontologies. In N. F. Noy, editor, *Proceeding of the Protégé Conference, Madrid*, pages 65–68. Stanford KSL, 2005.