J Autom Reasoning (2007) 39:317–349
DOI 10.1007/s10817-007-9076-z

Debugging Incoherent Terminologies

Stefan Schlobach · Zhisheng Huang · Ronald Cornet · Frank van Harmelen

Received: 1 May 2006 / Accepted: 1 February 2007 / Published online: 10 July 2007

Abstract In this paper we study the diagnosis and repair of incoherent terminologies. We define a number of new nonstandard reasoning services to explain incoherence through pinpointing, and we present algorithms for all of these services. For one of the core tasks of debugging, the calculation of minimal unsatisfiability preserving subterminologies, we developed two different algorithms, one implementing a bottom-up approach using support of an external description logic reasoner, the other implementing a specialized tableau-based calculus. Both algorithms have been prototypically implemented. We study the effectiveness of our algorithms in two ways: we present a realistic case study where we diagnose a terminology used in a practical application, and we perform controlled benchmark experiments to get a better understanding of the computational properties of our algorithms in particular and the debugging problem in general.

Keywords Debugging · Diagnosis · Description logics

1 Introduction

Ontologies play a crucial role in the Semantic Web (SW), as they allow information to be shared in a semantically unambiguous way and domain knowledge to be reused (possibly created by external sources). As a result, however, SW technology is highly dependent on the quality and correctness of these ontologies. Two general strategies for quality assurance are predominant, the first based on developing more and more sophisticated ontology modeling tools, and the second based on logical reasoning. In

S. Schlobach (✉) · Z. Huang · F. van Harmelen
Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
e-mail: schlobac@few.vu.nl

R. Cornet
AMC, Universiteit van Amsterdam, Amsterdam, The Netherlands

this paper we focus on the latter. With the advent of expressive ontology languages such as OWL and its close relation to description logics (DLs), state-of-the-art DL reasoners can efficiently detect inconsistencies even in very large ontologies. The practical problem remains what to do if an ontology has been detected to be locally incorrect.

Inconsistent ontologies can be dealt with in two main ways. One is to simply "live with" the inconsistency and to apply a nonstandard reasoning method to obtain meaningful answers in the presence of inconsistencies. Such an approach is taken in [11]. An alternative approach is to resolve (or "debug") the error whenever an inconsistency is encountered. In this paper we focus on this *debugging* process and on the terminological part of ontologies (and talk about debugging of terminologies). We introduce the formal foundations for debugging and diagnosis of logically incorrect terminologies, 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.

Our approach to diagnosing incoherent terminologies is based on traditional *model-based diagnosis*, which has been studied for many years in the AI community [20]. Here the aim is to find minimal fixes, in our case minimal subsets of a terminology that need to be repaired or removed to render a terminology logically correct and therefore usable again. We will see that in Reiter's terminology, MIPS and MUPS correspond to minimal conflict sets.

We will describe two algorithms for debugging: a bottom-up method using the support of an external reasoner, and an implementation of a specialized top-down algorithm. The former is based on the systematic enumerations of terminologies of increasing size based on selection functions on axioms; the latter is based on Boolean minimization of labels in a labeled tableau calculus. Both methods have been implemented as prototypes. The prototype for the informed bottom-up approach is called DION (Debugger of Inconsistent ONtologies);[1] the prototype of the specialized top-down method is called MUPSTER.

We provide a detailed evaluation of our methods. This is done in two ways. First, we present a *case study* where we apply our algorithms to two medical terminologies that are used in a real application in the Academic Medical Center Amsterdam for the admission of patients to intensive care units. Second, we perform a set of *controlled benchmark experiments* to get a better understanding of the computational properties of the debugging problem and our algorithms for solving it.

The combined results of the case study and the controlled experiments show that, on the one hand, debugging is useful in practice but that, on the other hand, we cannot guarantee that our tools will always find explanations in a reasonable time. The most important criteria will turn out to be the size and complexity of the definitions and the number of modeling errors.

1.1 Related Work

Before presenting our own work, we briefly discuss the most important related work in the literature. In our earlier conference publications [23] we were the first to

[1] http://wasp.cs.vu.nl/sekt/dion.

propose the framework for debugging and diagnosing of terminologies that is defined in Section 2. There we also coined the term *pinpointing* as a means of reducing a logically incorrect terminology to a smaller one, from which a modeling error could be more easily detected by a human expert. In [22] we grounded notions of MIPS and MUPS in the well-established theory of model-based diagnosis [20].

The area of debugging inconsistent ontologies has received much attention since the publication of [23]. In this section we do not aim to give a general literature survey, but discuss some of the most influential pieces of work, in particular the work by the MINDSWAP group and the work based on belief revision.

Debugging in the DL Community The MINDSWAP group at the University of Maryland has done significant work in this area, culminating in the recent thesis of Kalyanpur [12]. The work investigates two approaches: one based on modifying the internals of a DL reasoner (the "glass-box" approach), and one based on using an unmodified external reasoner (the "black-box" approach).

The glass-box approach is closely related to our work in Section 3.1 and (just as our work) is based on the techniques in [2]. The work deals with OWL-Lite, except for max cardinality roles, and is efficient because it avoids having to do full tableau saturation (details are in [14]). The work in [2] is particularly noteworthy: Although that paper is about a different topic (computing extensions for a certain class of default description logics), it turns out that one of the algorithms is very similar to the one described in Section 3.1. The main difference is that Baader et al. consider ABoxes instead of TBoxes and that their algorithm is for computing default extensions rather than computing diagnoses.

The black-box approach (i.e., detecting inconsistencies by calling an unmodified external DL reasoner) is based on Reiter's hitting set algorithm (similar to our work in [22]) and is also closely related to a proposal of Friedrich et al. [5], who (as we do in Section 2.2.1) bring the general diagnostic theories from [20] to bear for diagnosing ontologies. An interesting difference with our work is that Friedrich et al. use generic diagnoses software. As we do in our bottom-up method, they use a DL reasoner as oracle.

Kalyanpur also proposes a method for "axiom pinpointing,"[2] which rewrites axioms into smaller ones and then debugs the resulting ontology after rewriting, with the effect that a more precise diagnosis is obtained. Early results have been reported in [13].

A second pinpointing technique, called "error pinpointing" by Kalyanpur, is similar to what we call pinpointing here. Kalyanpur has performed user studies that reveal that a combination of axiom pinpointing (i.e., breaking large axioms into smaller ones) and error pinpointing (i.e., finding the errors that lie at the root of a cascading chain of errors) seems to be the cognitively most efficient support for users.

A significant extension to our work in [23] was published in [17], where the authors extend our saturation-based tableau calculus with blocking conditions, so that general TBoxes can be handled.

[2]A different use of the word pinpointing from our use in Section 2.2.3 and even from the identical term "axiom pinpointing" in [23].

Belief Revision Much of the work in the belief revision community over the past 20 years has focused on dealing with inconsistency, and significant advances have been made [7]. Nevertheless, there are significant differences that cause this work to be not directly applicable to ontology revision. First of all, most of the work on belief revision is phrased in terms of a "belief set," a deductively closed set of formulas. Much of the interest in dealing with inconsistent ontologies is to deal with sets of axioms that are not deductively closed and on which deduction has to be performed in order to find out inconsistencies and their causes (e.g., our Section 3.1). Furthermore, theories of belief revision typically assume a preference ordering among all models of a belief set, representing an order of implausibility among all situations. Such an approach is also taken in [16], which imposes a stratification on the knowledge base and employing this stratification to select a suitable repair.

Some work on belief revision does not rely on deductively closed belief sets and hence is more relevant to our work. One example is [8], from whom we have taken the notion of a syntactic relevance function. (Such a syntactic relevance function makes sense only when abandoning the notion of deductively closed belief sets, since an immediate consequence of working with such sets is that equivalent formulas should be treated equally, a.k.a. the principle of irrelevance of syntax.) Our work in Section 3.2 can be seen as a specialization to ontologies of the general framework presented in [8].

Contributions of This Paper In summary, the contributions of this paper are as follows. We present a formal characterization for debugging and diagnosis (briefly repeated from earlier publications); we define algorithms for all tasks described in our debugging framework; we study the effectiveness of our proposal in a realistic setting on two life-size terminologies; and we perform a set of controlled experiments to analyze the computational properties of the debugging problem and our different algorithms for solving it.

Structure of This Paper This paper is organized as follows. In Section 2 we define the formal notions that underlie our approach to debugging and diagnosis. In Section 3 we present the required algorithms. Section 4 briefly presents a realistic case study where we deployed these algorithms. Sections 5 and 6 are devoted to a set of controlled benchmark experiments: Section 5 describes the experimental setup, and Section 6 presents the results of our experiments. We end this paper with some concluding remarks about these results.

2 Formal Definitions

Description logics are a family of well-studied set-description languages that have been used 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

For a detailed introduction to description logics we point to the second chapter of the DL handbook [1]. Briefly, in DL concepts are interpreted as subsets of a domain, and roles as binary relations. Throughout the paper, let $\mathcal{T} = \{ax_1, \ldots, ax_n\}$ 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. We will also use terminological axioms of the form $C = D$ and disjointness statements *disjoint(C,D)* between two concepts C and D, which are simple abbreviations of $C \sqsubseteq D \& D \sqsubseteq C$ and $C \sqsubseteq \neg D$, respectively. Most DL systems also allow for assertional axioms in a so-called ABox. In this paper, ABoxes will not be considered. Throughout the paper the term *ontologies* will refer to general knowledge bases that possibly include both terminological and assertional knowledge. The term *terminology* is used solely in the technical sense of a DL TBox.

2.1.1 Unsatisfiability and Incoherence

Let $\mathcal{U}$ be a set of objects, called the universe, and $(\cdot)^{\mathcal{I}}$ a mapping that interprets DL concepts as subsets of $\mathcal{U}$. An interpretation $\mathcal{I} = (U, (\cdot)^{\mathcal{I}})$ is then called 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 that is a model for all axioms in $\mathcal{T}$. Based on these semantics, a TBox can be checked for *incoherence*, namely, whether there are *unsatisfiable* concepts that are necessarily interpreted as the empty set in all models of the TBox. More formally:

1. A concept C is *unsatisfiable* w.r.t. a terminology $\mathcal{T}$ if and only if $C^{\mathcal{I}} = \emptyset$ for all models $\mathcal{I}$ of $\mathcal{T}$.
2. A TBox $\mathcal{T}$ is *incoherent* if and only if there is a concept name in $\mathcal{T}$ that is unsatisfiable.

Conceptually, these cases often point to modeling errors because we assume that a knowledge modeler would not specify something like an impossible concept in a complex way.

Table 1 demonstrates this principle. Consider the (incoherent) TBox $\mathcal{T}_1$, where A, B, and C, as well as $A_1, \ldots, A_7$ are concept names, and r and s roles. Satisfiability testing 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, for example, 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.

2.1.2 Unfoldable $\mathcal{ALC}$ TBoxes

In this paper we study ways of *debugging and diagnosing* of incoherence and unsatisfiability in DL terminologies. The general ideas can easily be extended to inconsistency of ontologies with assertions as suggested in [24]. Since the evaluation in this paper is about terminological debugging only, we restrict the technical definitions to the necessary notions.

Table 1 A small (incoherent) TBox $\mathcal{T}_1$

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

Whereas the definitions of debugging were independent of the choice of a particular description logic, we will later present algorithms for the description logic $\mathcal{ALC}$, and unfoldable TBoxes in particular.

$\mathcal{ALC}$ is a simple yet relatively expressive DL with conjunction ($C \sqcap D$), disjunction ($C \sqcup D$), negation ($\neg C$), and universal ($\forall r.C$) and existential quantification ($\exists r.C$), where the interpretation function is extended to the different language constructs as follows.

$$\begin{aligned}
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(C \sqcup D)^{\mathcal{I}} &= C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
(\neg C)^{\mathcal{I}} &= \mathcal{U} \setminus C^{\mathcal{I}} \\
(\exists R.C)^{\mathcal{I}} &= \{d \in \mathcal{U} \mid \exists e \in \mathcal{U} : (d, e) \in R^{\mathcal{I}} \text{ and } e \in C^{\mathcal{I}}\} \\
(\forall R.C)^{\mathcal{I}} &= \{d \in \mathcal{U} \mid \forall e \in \mathcal{U} : (d, e) \in R^{\mathcal{I}} \text{ implies } e \in C^{\mathcal{I}}\}
\end{aligned}$$

A TBox is called *unfoldable* if the left-hand sides of the axioms (the defined concepts) are atomic and unique and if the right-hand sides (the definitions) contain no direct or indirect reference to the defined concept [18]. In T_1, our example TBox, $A_1, \ldots, A_7$ are defined concepts.

2.2 Framework for Debugging and Diagnosis

We now introduce a theory of debugging and diagnosis and link it to description logic-based systems. In this 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.

In some situations, terminologies can contain a large number of unsatisfiable concepts. This can occur, for example, when terminologies are the result of a merging process of separately developed terminologies or when closure axioms (i.e., disjointness statements and universal restrictions) are added to terminologies. Unsatisfiability propagates; that is, one unsatisfiable concept may cause many other concepts to become unsatisfiable as well. Since a modeler often cannot determine 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 terminology.

2.2.1 Model-based Diagnosis

The literature on model-based diagnosis is manifold, but we focus on the seminal work of Reiter [20] and the subsequent work in [6], which corrects a small bug in Reiter's original algorithm. We refer the interested reader to a good overview in [3].

Reiter introduces a diagnosis 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 the observed behavior. In other words, assuming correctness of any one of the components in a diagnosis would cause inconsistency between the system description and the observed behavior.

To apply this definition to a description logic terminology, we regard the terminology as the *system* to be diagnosed and the axioms as the *components* of this system. If we look at the example terminology from Table 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. In Reiter's terminology, a minimal set of axioms that need to be removed (or better, fixed) is called a diagnosis. This adaptation of Reiter's method leads to the following definition of diagnosis.

Definition 1 Let $\mathcal{T}$ be an incoherent terminology. A *diagnosis for the incoherence problem of* $\mathcal{T}$ is a minimal set of axioms $\mathcal{T}' \subseteq \mathcal{T}$ such that $\mathcal{T} \setminus \mathcal{T}'$ is coherent. Similarly, a *diagnosis for unsatisfiability of a single concept* A in $\mathcal{T}$ is a minimal subset $\mathcal{T}' \subseteq \mathcal{T}$, such that A is satisfiable w.r.t. $\mathcal{T} \setminus \mathcal{T}'$.

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 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.

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 terminologies, these minimal conflict sets play a role of their own, as they are the prime tools for debugging, that is, 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, and the MIPS for incoherence of a terminology.

2.2.2.1 Minimal Unsatisfiability-Preserving Sub-TBoxes (MUPS) In [23] 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 – those containing only axioms that are necessary to preserve unsatisfiability.

Definition 2 A TBox $\mathcal{T}' \subseteq \mathcal{T}$ is a *minimal unsatisfiability-preserving sub-TBox (MUPS)* for A in $\mathcal{T}$ 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 as follows.

$$\begin{aligned}
&mups\ (\mathcal{T}_1, A_1) : \{\{ax_1, ax_2\}, \{ax_1, ax_3, ax_4, ax_5\}\}\\
&mups\ (\mathcal{T}_1, A_3) : \{\{ax_3, ax_4, ax_5\}\}\\
&mups\ (\mathcal{T}_1, A_6) : \{\{ax_1, ax_2, ax_4, ax_6\}, \{ax_1, ax_3, ax_4, ax_5, ax_6\}\}\\
&mups\ (\mathcal{T}_1, A_7) : \{\{ax_4, ax_7\}\}
\end{aligned}$$

In the terminology of Reiter's diagnosis each *mups*($\mathcal{T}$, A) is a collection of minimal conflict sets w.r.t. satisfiability of concept A in TBox $\mathcal{T}$.

Remember that a diagnosis is a minimal hitting set for a collection of conflict sets. Hence, from the MUPS, we can also calculate the diagnoses for unsatisfiability of concept A in TBox $\mathcal{T}$, which we denote $\Delta_{\mathcal{T},A}$.

$$\begin{aligned}
&\Delta_{\mathcal{T}_1,A1} : \{\{ax_1\}, \{ax_2, ax_3\}, \{ax_2, ax_4\}, \{ax_2, ax_5\}\}\\
&\Delta_{\mathcal{T}_1,A3} : \{\{ax_3\}, \{ax_4\}, \{ax_5\}\}\\
&\Delta_{\mathcal{T}_1,A6} : \{\{ax_1\}, \{ax_4\}, \{ax_6\}, \{ax_2, ax_3\}, \{ax_2, ax_5\}\}\\
&\Delta_{\mathcal{T}_1,A7} : \{\{ax_4\}, \{ax_7\}\}
\end{aligned}$$

2.2.2.2 Minimal Incoherence-Preserving Sub-TBoxes (MIPS) MUPS are useful for relating sets of axioms to the unsatisfiability of specific concepts, but they can also be used to calculate MIPS, which relate sets of axioms to the incoherence of a TBox in general (i.e., unsatisfiability of at least one concept in a TBox).

Definition 3 A TBox $\mathcal{T}' \subseteq \mathcal{T}$ is a *minimal incoherence-preserving sub-TBox (MIPS)* of $\mathcal{T}$ if and only if $\mathcal{T}'$ is incoherent and every sub-TBox $\mathcal{T}'' \subset \mathcal{T}'$ is coherent.

Hence, MIPS are minimal subsets of an incoherent 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 three MIPS: *mips*($\mathcal{T}_1$) = $\{\{ax_1, ax_2\}, \{ax_3, ax_4, ax_5\}$, and $\{ax_4, ax_7\}\}$.

Analogous to MUPS, each element of *mips*($\mathcal{T}$) is a minimal conflict set w.r.t. incoherence 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 $\mathcal{T}_1$.

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

The number of MUPS a MIPS is a subset of determines the number of unsatisfiable concepts of which it might be the cause. We call this number the *MIPS-weight*.

In the example terminology $\mathcal{T}_1$ 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, and the weight of $\{ax_4, ax_7\}$ is one. Intuitively, this suggests that the combination of the axioms $\{ax_3, ax_4, ax_5\}$ is more relevant than $\{ax_4, ax_7\}$.

Weights are easily calculated and play an important role in practice to determine relative importance within the set of MIPS, as we experienced in our case studies that are described in Section 4.

2.2.3 Pinpoints

Experiments described in [22] indicated that calculating diagnoses from MIPS and MUPS is simple but computationally expensive and often impractical for real-world terminologies. For this purpose, we introduced in [21] the notion of a *pinpoint* of an incoherent terminology $\mathcal{T}$, in order to approximate the set of diagnoses. The definition of the set of pinpoints is a procedural one, following a heuristic to ensure that *most* pinpoints will indeed be diagnoses. There is no guarantee of minimality, however, so not every pinpoint is necessarily a diagnosis.

To define pinpoints, we need the notion of a core: 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.

Definition 4 A nonempty subset of the 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, $\{ax_4\}$ of size 1. The other axioms in the MIPS are 1-ary cores. Pinpoints are defined in a structural way.

Definition 5 Let $mips(\mathcal{T})$ be the set of MIPS of $\mathcal{T}$, that is, a collection of sets of axioms. The set of possible outputs of the following procedure will be called the *set of pinpoints*.

Let $M := mips(\mathcal{T})$ be the collection of MIPS for $\mathcal{T}$, $P = \varnothing$:

(1) Choose in M an arbitrary core $\{ax\}$ of size 1 with maximal arity.
(2) Then, remove from M any MIPS containing $\{ax\}$.
(3) $P := P \cup \{ax\}$.

Repeat steps 1 to 3 until $M = \varnothing$. The set P is then called a *pinpoint* of the terminology.

Since step 1 contains a nondeterministic choice, there is no unique *pinpoint* but a set possible of possible outputs of the algorithm: the set of pinpoints.

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 the 2-ary core, $\{ax_4\}$. Removing the MIPS containing ax_4 leaves $\{ax_1, ax_2\}$. Hence, there is a nondeterministic choice: if we choose ax_1 to continue, $\{ax_4, ax_1\}$ is the calculated pinpoint; otherwise $\{ax_4, ax_2\}$. Both are diagnoses of $\mathcal{T}_1$.

3 Algorithms

In this section we present algorithms for all major reasoning tasks introduced in Section 2.

Technically, calculating MUPS is the biggest challenge, and we present two different algorithms: 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. The first is a so-called glass-box method, the second a black-box method.

To calculate MIPS, we first calculate MUPS for each unsatisfiable concept. From the MIPS, diagnoses can be calculated by using Reiter's original hitting-set tree method.

The algorithms for debugging were implemented in prototypical systems: the top-down algorithms in a system called MUPSTER, the bottom-up algorithms in a system called DION.

3.1 A Top-down Algorithm for Calculating MUPS

In the glass-box top-down approach we calculate the axioms required to maintain a logical contradiction by expanding a logical tableau with labels. This method requires a single tableau proof per unsatisfiable concept. On the other hand, it is based on a variation of a specialized logical algorithm and works only for description logics for which such a specialized algorithm exists and is implemented.

The algorithm described in this section has recently been extended to general terminologies in [17]. The prototypical implementation in the MUPSTER system, which we applied in two practical use-cases, is restricted to definitorical TBoxes. Our experience has shown that applying our methods to simple terminologies already yields valuable debugging information and that it is worth discussing this approach in more detail.[3] The algorithm we will describe calculates MUPS for the DL $\mathcal{ALC}$ and unfoldable TBoxes. It calculates MUPS based on Boolean minimization 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 Fig. 1) where all branches contain a contradiction (or " are closed," as we say). The information that axioms are relevant for the closure is contained in a simple label that is added to each formula in a branch. A *labeled formula* has the form $(a : C)^x$ where a is an individual name, C a concept and x a set of axioms, which we refer to as *label*. A labeled branch is a set of labeled formulas, and a tableau is a set of labeled branches. A formula can occur with different labels on the same branch. A branch is closed if it contains a clash, that is, if there is at least one pair of formulas with the same negated and non-negated atom on the same individual. The notions of open branch and closed and open tableaux are defined as usual and do not depend on the labels. We always assume that formulas are in *negation normal form* (nnf) and that newly created formulas are immediately transformed. We usually omit the prefix "labeled."

[3]Extending the actual implementation of MUPSTER to more expressive languages is not straightforward but is planned for future work.

($\sqcap$):	**if**	$(a : C_1 \sqcap C_2)^l \in B$, but not $\{(a : C_1)^l, (a : C_2)^l\} \subseteq B$
	then	$B' := B \cup \{(a : C_1)^l, (a : C_2)^l\}$.
($\sqcup$):	**if**	$(a : C_1 \sqcup C_2)^l \in B$, but neither $(a : C_1)^l \in B$ nor $(a : C_2)^l \in B$
	then	$B' := B \cup \{(a : C_1)^l\}$ and $B'' := B \cup \{(a : C_2)^l\}$.
(Ax)	**if**	$(a : A)^l \in B$ and $(A \sqsubseteq C) \in \mathcal{T}$
	then	$B' := B \cup \{(a : C)^{l \cup \{A \sqsubseteq C\}}\}$.
($\exists$):	**if**	$(a : \exists R_i.C)^l \in B$, and all other rules have been applied on all formulas over a, and if $\{(a : \forall R_i.C_1)^{l_1}, \ldots, (a : \forall R_i.C_n)^{l_n}\} \subseteq B$ is the set of universal formulas for a w.r.t. R_i in B,
	then	$B' := \{(b : C)^l, (b : C_1)^{l_1 \cup l}, \ldots, (b : C_n)^{l_n \cup l}\}$ where b is a new individual name not occurring in B.

Fig. 1 Tableau rules for $\mathcal{ALC}$-satisfiability w.r.t. a TBox $\mathcal{T}$ (with labels)

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 Fig. 1 as long as possible. The rules are standard $\mathcal{ALC}$-tableau rules with lazy unfolding and are 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, $T'' := \{B', B'', B_1, \ldots, B_n\}$ for the ($\sqcup$)-rule.

Once no more rules can be applied, we know which concept names are needed to close a saturated branch and can construct a minimization function for A and $\mathcal{T}$ according to the rules in Fig. 2. A propositional formula ϕ is called a *minimization function for A and $\mathcal{T}$* if A is unsatisfiable in every subset of $\mathcal{T}$ containing the axioms that are true in an assignment making ϕ true. In our case axioms are used as propositional variables in ϕ. Since 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 subbranches conjunctively. If an existential rule has been applied, the new branch

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

Fig. 2 *min_function*$(a, B, \mathcal{T})$: Minimization function for the MUPS-problem

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 minimization function $min\_function(a, \{(a : A)^{\varnothing}\}, \mathcal{T})$ (let us call it ϕ) that we calculated using the rules in Fig. 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[4] implying ϕ [19]. Since ϕ is a minimization function, every implicant of ϕ must be a minimization 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 $\{ax_1, \ldots, ax_n\}$. Since $ax_1 \wedge \ldots \wedge ax_n$ is the smallest implicant, we also know that $\{ax_1, \ldots, ax_n\}$ must be minimal, that is, a MUPS. Theorem 3.1 captures this result formally.

Theorem 3.1 *Let A be a concept name that is unsatisfiable w.r.t. an unfoldable $\mathcal{ALC}$-TBox $\mathcal{T}$. The set of prime implicants of the minimization function $min\_function(a, \{(a : A)^{\varnothing}\}, \mathcal{T})$ is the set $mups(\mathcal{T}, A)$ of minimal unsatisfiability-preserving sub-TBoxes of A and $\mathcal{T}$.*

Proof We first prove the claim that the propositional formula $\phi := min\_function(a, \{(a : A)^{\varnothing}\}, \mathcal{T})$ is indeed a minimization function for the MUPS problem w.r.t. an unsatisfiable concept A and a TBox $\mathcal{T}$. We show that a tableau starting on a single branch $B := \{(a : A)^{\varnothing}\}$ closes on all branches by unfolding axioms only that are evaluated as true in an assignment making ϕ true. This saturated tableau Tab^* is a particular subtableau of the original saturated tableau Tab that we used to calculate $min\_function(a, \{(a : A)^{\varnothing}\}, \mathcal{T})$, and it is this connection that we use to prove our first claim. Every branch in the new tableau is a subset of a branch occurring in the original one, and we define *visible formulas* as those labeled formulas occurring in both tableaux. By induction over the rules applied to saturate Tab we can then show that each branch in the original tableau closes on at least one pair of visible formulas. If A is unsatisfiable w.r.t. $\mathcal{T}$, the tableau starting with the branch $\{(a : A)^{\varnothing}\}$ closes w.r.t. $\mathcal{T}$. As we have shown that this tableau closes w.r.t. $\mathcal{T}$ on visible formulas, it follows that Tab^* is closed on all branches, which proves the first claim. By another induction over the application of the rules in Fig. 2, we can prove that ϕ is a *maximal* minimization function, which means that $\psi \rightarrow \phi$ for every minimization function ψ. This proves the first part of the proof; the first claim (and the argument from above) implies that every implicant of a minimization function identifies an unsatisfiability-preserving TBox, and maximality implies that prime implicants identify the minimal ones.

To show that the conjunction of every MUPS $\{ax_1, \ldots, ax_n\}$ is a prime implicant of $min\_function(a, \{(a : A)^{\varnothing}\}, \mathcal{T})$ is trivial, since $ax_1 \wedge \ldots \wedge ax_n$ is a minimization function by definition. But as we know that $min\_function(a, \{(a : A)^{\varnothing}\}, \mathcal{T})$ is maximal, we know that $ax_1 \wedge \ldots \wedge ax_n \rightarrow min\_function(a, \{(a : A)^{\varnothing}\}, \mathcal{T})$ which implies that $ax_1 \wedge \ldots \wedge ax_n$ must be prime, since otherwise $\{ax_1, \ldots, ax_n\}$ would not be minimal. □

[4]Note that in our case all literals are positive axioms.

The proof of Theorem 3.1 is almost identical to the proof of Proposition 6.3 and the subsequent lemmas of [2], in which the authors show correctness of their algorithm to calculate minimal inconsistent ABoxes in order to calculate minimal models for description logics with defaults.

Satisfiability in $\mathcal{ALC}$ is PSPACE-complete, and calculating MUPS does not increase the complexity because we can construct the minimization function in a depth-first way, allowing us to keep only one single branch in memory at a time. However, we calculate prime implicants of a minimization function the size of which can be exponential in the number of axioms in the TBox.

3.2 An Informed Bottom-up Algorithm for MUPS

In this section we describe an informed bottom-up algorithm to calculate MUPS with the support of an external DL reasoner. The main advantage of this approach is that it can deal with any DL-based ontology supported by an external reasoner. Currently there exist several well-known DL reasoners, such as RACER,[5] FaCT++,[6] and Pellet,[7] each of which has proved to be reliable and stable. They support various DL-based ontology languages, including OWL-DL.

Given an unsatisfiable concept A and a terminology $\mathcal{T}$, MUPS can be systematically calculated by checking whether A is unsatisfiable in subsets $\mathcal{T}'$ of $\mathcal{T}$ of increasing size. Such a procedure is complete and easy to implement, but infeasible in practice. Even the simplest real-world terminology in our tests in Section 6.1 has an average size of 5 axioms per MUPS and 417 axioms, which requires over 10^{11} calls to the external reasoner.

This observation implies that one has to control the subsets of $\mathcal{T}$ that are checked for satisfiability of A by means of a *selection function*. Such a selection function selects increasingly large subsets that are heuristically chosen to be relevant additions to the currently selected subset. Although this approach is not guaranteed to give us the complete solution set of MUPS, it provides an efficient approach for debugging inconsistent terminologies. We will now formally introduce the core notions of selection functions and relevance.

Given a terminology $\mathcal{T}$ and an axiom *ax*, a *selection function s* is a function that returns a linearly ordered collection of subsets of $\mathcal{T}$. More formally, for an ontology language $\mathbf{L}$, a selection function s is a mapping $s : \mathcal{P}(\mathbf{L}) \times \mathbf{L} \times \mathbb{N} \rightarrow \mathcal{P}(\mathbf{L})$ such that $s(\mathcal{T}, \phi, k) \subseteq \mathcal{T}$.

In [10] we defined two selection functions, the simplest one based on co-occurrence of concept names in axioms. Since in this paper we focus on unfoldable TBoxes,[8] we use a slightly more complex selection function here. The basic idea is that an axiom *ax* is relevant to a concept name A if and only if A occurs on the left-hand side of *ax*. In a way this variant of the bottom-up approach mimics the unfolding procedure in order to restrict the number of tests needed. This is also the one implemented in the DION system.

[5]http://www.sts.tu-harburg.de/~r.f.moeller/racer/.

[6]http://owl.man.ac.uk/factplusplus/.

[7]http://www.mindswap.org/2003/pellet/.

[8]Remember that the top-down is defined for unfoldable TBoxes only.

We use $\mathcal{V}_c(ax)$ ($\mathcal{V}_c(C)$) to denote the set of concept names that appear in an axiom ax (in a concept C, respectively). Concept-relevance is defined as follows.

Definition 6 An axiom ax is *concept-relevant to a concept or an axiom* ϕ iff

(1) $\mathcal{V}_c(C_1) \cap \mathcal{V}_c(\phi) \neq \varnothing$ if the axiom ax has the form $C_1 \sqsubseteq C_2$,
(2) $\mathcal{V}_c(C_1) \cap \mathcal{V}_c(\phi) \neq \varnothing$ or $\mathcal{V}_c(C_2) \cap \mathcal{V}_c(\phi) \neq \varnothing$ if the axiom ax has the form $C_1 = C_2$ or $disjoint(C_1, C_2)$.

Note that this approach is a syntactic one because, for example, the axiom $\neg D \sqsubseteq \neg C$ is treated differently from the axiom $C \sqsubseteq D$.

Based on this particular relevance function we can now, for a terminology $\mathcal{T}$ and a concept A, define a selection function s as follows.

Definition 7 The *concept-relevance-based selection function* for a TBox $\mathcal{T}$ and a concept A is defined as

(1) $s(\mathcal{T}, A, 0) = \varnothing$;
(2) $s(\mathcal{T}, A, 1) = \{ax \mid ax \in \mathcal{T}$ and ax is concept-relevant to $A\}$;
(3) $s(\mathcal{T}, A, k) = \{ax \mid ax \in \mathcal{T}$ and ax is concept-relevant to an element in $s(\mathcal{T}, A, k-1)\}$ for $k > 1$.

We use an informed bottom-up approach to obtain MUPS. In logics and computer science, an increment-reduction strategy is often used to find minimal inconsistent sets [4]. Under this approach, the algorithm first finds a collection of inconsistent subsets of an inconsistent set, before it removes redundant axioms from these subsets. Similarly, a heuristic procedure for finding MUPS of a TBox $\mathcal{T}$ and an unsatisfiable concept-name A consists of the following three stages:

– *Expansion* : Use a relevance-based selection function to find two subsets Σ and S of $\mathcal{T}$ such that a concept A is satisfiable in S and unsatisfiable in $S \cup \Sigma$.
– *Increment*: Enumerate subsets of Σ to obtain the sets S' such that the concept A is unsatisfiable in $S' \cup S$. We call those sets *A-unsatisfiable sets*.
– *Reduction*: Remove redundant axioms from those A-unsatisfiable sets to get MUPS.

Figure 3 describes an algorithm **MUPS_bottomup(**$\mathcal{T}$, A) based on this strategy to calculate MUPS. The algorithm first finds two subsets Σ and S of $\mathcal{T}$ by increasing the relevance degree k on the selection function until A is unsatisfiable in $S \cup \Sigma$ but satisfiable in Σ. Compared with $\mathcal{T}$, the set Σ can be expected to be relatively small. The algorithm then builds the power-set of Σ to get A-unsatisfiable sets by adding an axiom $ax \in \Sigma$ in each iteration of the loop to the sets S' in the working set W. If A is satisfiable in $S' \cup \{ax\}$, then the set $S' \cup \{ax\}$ is added to the working set to build up the union of each element of the power-set of Σ with the set S.[9] If A is unsatisfiable in $S' \cup \{ax\}$, then the set $S' \cup \{ax\}$ is added into the resulting set $M(\mathcal{T}, A)$ instead of the working set W. This approach avoids the calculation of the full power-set of

[9]Namely, $\{S'/S | S' \in W\} \subseteq \mathcal{P}(\Sigma)$.

```
$k := 0$
$M(\mathcal{T}, A) := \varnothing$
**repeat**
  $k := k + 1$
**until** $A$ unsatisfiable in $s(\mathcal{T}, A, k)$                                   (*)
$\Sigma := s(\mathcal{T}, A, k) - s(\mathcal{T}, A, k-1)$
$S := s(\mathcal{T}, A, k-1)$
$W := \{S\}$
**for all** $ax \in \Sigma$ **do**
  **for all** $S' \in W$ **do**
    **if** $A$ satisfiable in $S' \cup \{ax\}$ and $S' \cup \{ax\} \notin W$ **then**
      $W := W \cup \{S' \cup \{ax\}\}$
    **end if**
    **if** $A$ unsatisfiable in $S' \cup \{ax\}$ and $S' \cup \{ax\} \notin M(\mathcal{T}, A)$ **then**
      $M(\mathcal{T}, A) := M(\mathcal{T}, A) \cup \{S' \cup \{ax\}\}$
    **end if**
  **end for**
**end for**
$M(\mathcal{T}, A) :=$ **MinimalityChecking**$(M(\mathcal{T}, A))$
**return** $M(\mathcal{T}, A)$
```

Fig. 3 **MUPS_bottomup**

Σ because any superset of $S' \cup \{ax\}$ in which A is unsatisfiable is pruned. Finally, by checking minimality we obtain MUPS. The procedure to check minimality of the calculated subsets of $\mathcal{T}$ is described in Fig. 4.

Proposition 3.1 *The algorithm* **MUPS_bottomup**$(\mathcal{T}, A)$ *of* Fig. 3 *is sound. This means that it always returns MUPS, in other words, that* $M(\mathcal{T}, A) \subseteq mups(\mathcal{T}, A)$*, for any output* $M(\mathcal{T}, A)$.

```
**for all** $M \in M(\mathcal{T}, A)$ **do**
  $M' := M$
  **for all** $ax \in M'$ **do**
    **if** $A$ unsatisfiable in $M' - \{ax\}$ **then**
      $M' := M' - \{ax\}$
    **end if**
  **end for**
  $M(\mathcal{T}, A) := M(\mathcal{T}, A) - \{M\} \cup \{M'\}$
**end for**
**return** $M(\mathcal{T}, A)$
```

Fig. 4 **MinimalityChecking**(M$(\mathcal{T}, A)$)

Proof It follows from the construction of the sets in the collection $M(\mathcal{T}, A)$ in **MUPS_bottomup**$(\mathcal{T}, A)$ that the concept A is always unsatisfiable for any element S in $M(\mathcal{T}, A)$. Otherwise it would have never been added in the first place. Minimality is enforced by the procedure **MinimalityChecking**(M$(\mathcal{T}, A)$). □

Take our running example. To calculate $M(\mathcal{T}_1, A_1)$, the algorithm first gets the set $\Sigma = \{ax_2, ax_3\} = \{ax_1, ax_2, ax_3\} - \{ax_1\}$. Thus, $M(\mathcal{T}_1, A_1) = \{\{ax_1, ax_2\}\}$. We note that the algorithm cannot find that $S_1 = \{ax_1, ax_3, ax_4, ax_5\}$ is a MUPS for $\mathcal{T}_1$ and A_1. This fact points to the incompleteness of our algorithm. The problem is the stopping condition of the expansion phase (denoted by $(*)$ in the algorithm). This condition means that only the MUPS with axioms with maximal relevance with regard to the unsatisfiable concept will be found. In principle, the rigid stopping condition $(*)$ in **MUPS_bottomup**$(\mathcal{T}, A)$ could easily be replaced by a full expansion, that is, by a condition that requires saturation of the expansion phase. However, since the primary goal of our implementation was practical applicability, the **MUPS_bottomup**$(\mathcal{T}, A)$ algorithm is implemented in DION as described above.

3.3 Calculating MIPS

Given all MUPS, we can easily calculate the 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 $\mathcal{T}' \subseteq \mathcal{T}$. A simple algorithm for the calculation of MIPS for $\mathcal{T}$ now simply follows from Theorem 3.2, which is a direct consequence of the definitions of MIPS and MUPS.

Theorem 3.2 *Let $\mathcal{T}$ be an incoherent TBox with unsatisfiable concepts $unsat(\mathcal{T})$. Then, $mips(\mathcal{T}) = sr(\bigcup_{A \in unsat(\mathcal{T})} mups(\mathcal{T}, A))$.*

We remark that this algorithm might produce nonminimal incoherence-preserving subterminologies when not all MUPS are available – as can happen, for example, in the incomplete version of our bottom-up algorithm.

3.4 Calculating Diagnoses and Pinpoints

We mentioned in Section 2 that MIPS (MUPS) correspond to minimal conflict sets for the diagnosis problem of incoherent terminologies (an unsatisfiable concepts). From Reiter's seminal paper [20] a relatively straightforward algorithm for calculating (both types of) diagnoses using hitting sets (HS) follows immediately. The general idea is that diagnoses are paths in minimal trees, where each of the nodes is labeled with a set of contradicting axioms (the conflict sets), and where each edge on the path "hits" precisely one node label on its way to the leaves. More formally, 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 ✓ 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 $S \cap H(n) = \emptyset$, 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

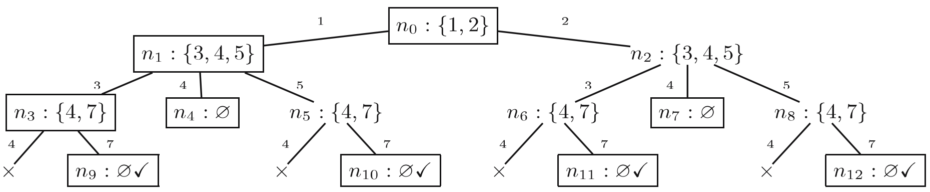

Fig. 5 HS-tree with MIPS

n by an edge labeled by σ. For any node labeled by ✓, $H(n)$, that is, the labels of its path from the root, is a hitting set for C.

Figure 5 shows a HS-tree T for the MIPS of our example TBox $\mathcal{T}_1$ of Section 2, where an axiom ax_i is represented by the number i. Each path in the tree to a leaf marked by ✓ is a diagnosis.

Unfortunately, the problem of finding minimal hitting sets is known to be NP-complete. In [22] we showed that a nonoptimized implementation of this algorithm often comes to a satisfactory solution, but also failed to calculate diagnoses even for relatively small terminologies.

Based on these findings, we proposed in [21] to calculate pinpoints as approximations for diagnoses. A detailed description of pinpoints can be found in Section 2.2.3.

4 Two Case Studies

Before providing an evaluation of computational behaviour of our algorithms, we briefly discuss two case studies from the medical domain.

4.1 DICE

The DICE knowledge base,[10] which is under development at the Academic Medical Center in Amsterdam, contains about 2,500 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 1,500 reasons for admission, DICE contains concepts regarding anatomy, etiology, and morphology.

DICE originally had a frame-based representation, which was migrated to DL in order to be able to perform auditing w.r.t. incorrect definitions and missed classification. The migration resulted in a TBox using the language $\mathcal{ALCQ}$. Since qualified number restrictions that are used in DICE are not yet supported by the algorithms, they were replaced by existential restrictions, resulting in a TBox using $\mathcal{ALC}$. In the DL-based representation many closure axioms are used in order to be able to find incorrect definitions. These closure axioms include universal restrictions and disjointness of sibling concepts. As a result of the migration, various concepts became unsatisfiable.

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

A recent version of DICE was classified, resulting in 65 unsatisfiable concepts. Using the bottom-up method described earlier (and the MUPSTER[11] system), we calculated 175 MUPS. Applying the algorithm described in the previous section, we found 142 MIPS, with 121 MIPS of weight 1, 10 MIPS of weight 2, and 11 MIPS of weight 4. This distribution indicates that a relatively small number of conflicts are a cause of more than one unsatisfiable concept, whereas the majority of conflicts result in only one unsatisfiable concept.

One of the pinpoints of this terminology consists of the following five axioms, which form the largest cores, according to the procedure defined in Section 2.2.3.

Disjointness of children of *Act*	with arity 60
Disjointness of children of *Dysfunction/Abnormality*	with arity 56
Disjointness of children of *System*	with arity 15
Axiom for *Heart valve operations*	with arity 7
Disjointness of children of *Toxical substance*	with arity 4

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." Given this method, we fully debugged DICE, which is now being applied to register patients in the intensive care unit of the AMC.

4.2 FMA

To test how our approach can be applied to other terminologies, we have used the Foundational Model of Anatomy (FMA).[12] FMA, developed by the University of Washington, provides about 69,000 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,[13] has been migrated to DL, using the language $\mathcal{ALC}$. In order to restrict the language to $\mathcal{ALC}$, inverse relations, specified in the frame-based representation, were ignored in the migration process. Because of its large size, we were not able to classify the full FMA terminology with RACER. We hence limited the case study to "Organs," which comprises a convenient subset that is representative for the FMA. Of the 3,826 concept definitions, 181 were found to be unsatisfiable. Interestingly, this resulted in a pinpoint with a single definition, that of "Organ." This could be explained by the definition of Organ.

[11]For the use case, a variant of MUPSTER was used with limited support for nonunfoldable TBoxes. This was necessary because of the disjointness axioms in DICE. In fact, we adapted the unfolding rule of calculus in Fig. 1 on page 11 so that a formula $(a : \neg A_1)^{I \cup \{disjoint(A_1, A_2)\}}$ is added for every disjointness statement $disjoint(A_1, A_2)$ whenever $(a : A_1)^I$ is in a branch. This approach can potentially lead to nontermination, which needs to be treated carefully. A consequence is that the procedure is not guaranteed to calculate all MUPS. Nevertheless, we will use the terms MUPS and MIPS throughout this paper to simplify the presentation.

[12]http://sig.biostr.washington.edu/projects/fm/.

[13]http://protege.stanford.edu/.

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

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

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

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 role value for the (Regional)PartOf role, that is, defining Organ as follows.

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

Changing the definition in this way results in a coherent TBox, which shows how one axiom can lead to unsatisfiability of a large number of concepts. Pinpointing properly detects this single axiom, which was indeed incorrectly specified from a medical perspective.

5 Experimental Setup

The results from the previous section are encouraging: for the practical cases they were developed for, the MUPS, MIPS, diagnoses, and the like indeed helped in some cases. What remains to be investigated is whether we have effective algorithms to calculate those debugging notions in general, whether a particular method will be more appropriate than another, and how well our methods will scale. In the next section we answer these questions by performing some controlled experiments, in which we run our tools MUPSTER and DION against a number of benchmarks.

5.1 Questions to Be Investigated

In some more detail, the questions we study are as follows:

– *Can debugging be performed efficiently?* More concretely, given an incoherent terminology, can our tools support a practitioner effectively? We will see that the answer to this question is not necessarily positive. Even worse, we know that for most description logics our problem is exponentially hard,[14] which means that we will never be able to guarantee termination in realistic time for arbitrary input. So, the following question will be very important to answer.

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

- *What makes an incoherent terminology difficult to debug?* Are there particular classes of terminologies that are more difficult to debug than others? If we can identify such classes, there is a logical follow-up question.
- *Which are the most appropriate methods for debugging incoherent terminologies?* Not only will there be classes that are more difficult than others, there might also be classes of TBoxes for which one method is more appropriate than the other. To answer this question could allow users to choose the appropriate implementation for their needs.

5.2 The Benchmarks

A good test set must meet certain criteria. Most important, a test set should be prototypical, so that it represents a larger class of realistic problems. Also, it has to be systematic, so that the influence of particular properties of classes of TBoxes can be evaluated. Moreover, a test set should be statistically viable; that is, 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.

To address the above-mentioned research questions under these criteria, we conducted three different types of benchmark experiments: first, an evaluation of the methods with real-world terminologies, second, an evaluation using an adapted benchmark set from the DL literature, and third, some experiments with our own purpose-built data set.

5.2.1 Evaluation with Real-world Terminologies

The first approach to evaluation of debugging methods is to consider a number of publicly available description logic terminologies. These are summarized in Table 2.

We split our test terminologies into three groups, according to the way they were built. As examples of the first group, terminologies created through migration, we consider an older version of the anatomy fragment of DICE (we abbreviate DICE-A) and a previous full version of full DICE (abbreviated DICE). The incoherence of DICE-A has two distinct causes. First, this is a snapshot from the terminology in its creation process; that is, it contains real modeling errors. Second, the high number of contradictions is specific for migration as a result of stringent semantic assumptions, which were made in order to uncover as many migration errors as possible.

In the second group, MGED and Geo are variants of terminologies that are incoherent because they have disjointness statements artificially added for semantic enrichment (as suggested in [21]). MGED[15] provides standard terms for the annotation of micro-array experiments to enable structured queries on those experiments. Geo[16] is a terminology of geography made available by the Teknowledge Corporation.

[15] http://www.mged.org/.

[16] http://ontology.teknowledge.com/.

Table 2 Real-world terminologies

	#ax	#unsat	#mips	|mips|	Length of mD
DICE-A	534	76	16	3	3
DICE	4,995	27	55	4	–
MGED	406	72	38	4	3
Geo	417	11	22	2.6	8
S&C	6,382	923	–	–	–
MadC	69	1	–	–	1

The third group contains the merged terminologies of SUMO[17] and CYC,[18] two well-known upper ontologies. Since they cover similar topics, there are a high number of unsatisfiable concepts.

We constructed $\mathcal{ALC}$ versions for all five terminologies. Without loss of unsatisfiability,[19] we removed, for example, numerical constraints, role hierarchies, and instance information. All terminologies, however, were noncyclic and could be transformed to an unfoldable format with the exception of the disjointness statements that were treated as described in footnote 11 on page 18.

For the last example shown in Table 2, the MadC[20] ontology, this is not the case. This ontology was constructed to illustrate language features of description logics. We use it to illustrate that DION's generic method works for expressive formalisms, where language-specific methods fail. MadC is incoherent with an unsatisfiable concept *MadCow*.

Benchmarking with real-life terminologies is a most natural way of evaluating the quality of debugging algorithms. On the other hand, only a limited number of realistic terminologies are available that are incoherent. There are two reasons for this. First, published terminologies usually have undergone a careful modeling process, and one should expect that logical modeling errors have already been eliminated. Second, current terminologies often still use quite inexpressive languages and avoid incoherence 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.

Alternatively, one commonly uses systematically created test sets for benchmarking, and such sets also exist for evaluating DL reasoning.

5.2.2 Benchmarking with (Adapted) Existing Test Sets

The issue of benchmarking description logic systems has been addressed several times in the literature over the past 10 years, mostly by using adaptations of modal-

[17] http://www.ontologyportal.org/.

[18] http://www.opencyc.org/.

[19] This means that all unsatisfiable concepts were still unsatisfiable after our transformations.

[20] http://www.cs.man.ac.uk/~horrocks/OWL/Ontologies/mad_cows.owl.

logic test sets [15]. In most of these studies the purpose was to evaluate the runtime of DL reasoners with respect to the complexity of the used language (mostly the modal depths). Nevertheless, systematic benchmarking remains an open problem. Unfortunately, our tools restrict the experiments to test sets with unfoldable $\mathcal{ALC}$ terminologies. Finding a good benchmark for unfoldable terminologies is difficult, though, since existing extensions to terminologies 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. We used the satisfiability tests from the well-known DL benchmark from the 1998 system comparison [9]. We took nine 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 for the speed of the DL system is the highest concept in the list that could still be solved in 100 s. These test sets were built in the early days of the latest generation of description logic tools, which did not have as many optimizations 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, since our implementation of the top-down algorithms works (in the current version) without optimizations.[21]

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 subconcept S of C that is in the scope of an odd number of negations is (recursively) replaced by an atom A_S, and an axiom $A_S \sqsubseteq S$ is added to the terminology, where A_S and A_C are new names not occurring in the concept. 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 $\{A_C \sqsubseteq A_1 \sqcap A_2,\ A_1 \sqsubseteq \exists r.(A \sqcup B),\ A_2 \sqsubseteq \forall r.(\neg A \sqcap \neg B)\}$. Next we transform the definitions of A_1 and A_2, which leads to the TBox $\{A_C \sqsubseteq A_1 \sqcap A_2,\ A_1 \sqsubseteq \exists r.A_3,\ A_2 \sqsubseteq \forall r.A_4,\ A_3 \sqsubseteq A \sqcup B,\ A_4 \sqsubseteq \neg A \sqcap \neg B\}$, and so on. The resulting terminology for $C = \exists r.(A \sqcup B) \sqcap \forall r.(\neg A \sqcap \neg B)$ is as follows.

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

For each set of unsatisfiable concepts in DL98 we created an incoherent terminology given the above-mentioned method for the first three concepts. The results are called k_branch_p_tbox1, k_branch_p_tbox2, and 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 nonoptimized reasoning*. We will

[21]There is an experimental implementation that trades completeness of the method with a number of optimizations. In this paper we restrict our attention to the more robust standard implementation of MUPSTER, which is complete for unfoldable TBoxes.

see that this has drastic effects on the runtimes of the different algorithms. On the other hand, this reasoning-centered view does not really coincide with the average structure of the realistic terminologies. Mostly these terminologies are relatively flat in structure, but large; and often the definitions are strongly dependent on other definitions. To account for this situation, 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 for debugging, 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, 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. For example, 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 choice. In 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 a plethora of parameters 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. See [24] for the details.

In particular, we decided to vary the size of the TBox (#t), the concept size (#s), the ratio of disjunction versus conjunction, and the ratio of negated versus non-negated atoms.

To create the *i*th axiom (i.e., to define the *i*th concept name), we create a concept of size #s drawn from all the concept-names combined with the remaining concept names that were not defined in the first $i-1$ axioms. The construction of the concept is then simply a construction based on random choice with the given probability distributions for disjunction/conjunction and negated/non-negated atoms.

Given this method, we constructed 1,611 unsatisfiable terminologies. Note that this method does not automatically deliver incoherent TBoxes. On the contrary, the overall rate of incoherence given this method is less than 30%. To consider only incoherent TBoxes, we therefore simply run an optimized DL reasoner to delete all coherent TBoxes from the test set.

We note that the choice of parameters greatly influences the ratio of satisfiable versus unsatisfiable terminologies, particularly the *disjunction/conjunction* ratio: with disjunction-likelihood of 50%, only 5% of the resulting TBoxes were incoherent, as opposed to almost 50% of the TBoxes when the likelihood was just 10%. The consequence is easy to see: if 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 a high disjunction ratio. The simplest 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 we have created a test set of TBoxes where all TBoxes corresponding to particular parameter choices are equally likely to occur. Both the test set and the generator will be made available on our website.[22]

6 Experimental Results

In the previous section, we described three different test sets we use for evaluating the runtimes: first, a set of six real-life terminologies we collected from the Internet and our own use-cases; second, an extension of an existing benchmark for evaluation of description logic systems; and third, a purpose-built benchmark consisting of 1,611 systematically constructed incoherent terminologies. 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.

For the evaluation we have to keep two aspects in mind: the difference in expressiveness of the two tools, and the fact that one is complete and the other not. Remember that, because DION uses an external DL systems for the reasoning part via a `DIG` interface, DION's expressiveness depends only on the external system being used. Thus, DION can debug arbitrary ontologies in very expressive description logics such as $\mathcal{SHIQ}$, whereas MUPSTERis restricted to unfoldable $\mathcal{ALC}$ TBoxes and is therefore less expressive than DION. On the other hand, it implements a method to guarantee that it calculates all MIPS for unfoldable TBoxes. However, our experience and experiments show that DION and MUPSTER produce the same results for all the terminologies we considered in our experiments. Although we cannot ensure that there are no pathological examples where DION fails to find a particular MIPS, there was no such example in our experiments, neither in the real-world examples we studied in Section 6.1 nor in the constructed terminologies of Section 6.2 or 6.3.

6.1 Experiments with Existing Terminologies

All experiments were 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 1 h.

Results Table 3 summarizes the runtimes of the two systems on the six terminologies. As a reference we also give the runtime RacerPro 1.8.1 needs to find the unsatisfiable concepts. For the MadCow terminology, MUPSTER could not be applied because this terminology contains GCIs.

While MUPSTER manages to calculate the MIPS within the time limit for all but the Sumo&Cyc terminology, DION fails in two additional cases: the anatomy part of DICE and the MGED TBox. The results of the experiments show a scalability problem for both algorithms with really big terminologies, as both methods failed to calculate MIPS in the case of Sumo&Cyc, and DION also needs more than 1 h for two other terminologies. On the other hand, the runtimes of DION are faster for the two examples where it finds a solution.

[22]http://www.few.vu.nl/~schlobac.

Table 3 Comparing top-down and bottom-up methods

	Top-down: MUPSTER	Bottom-up: DION	RacerPro
	time for all MIPS	time for all MIPS	time for coherence
DICE-A	12 s	timed out	4.3 s
DICE	54 s	32 s	16.62 s
MGED	5 s	timed out	3.38 s
Geo	20 s	11 s	1.82 s
Sumo&Cyc	timed out	timed out	4.7 s
MadCow	not applicable	0.66 s	0.22 s

We note that, as a proof of concept more than anything else, DION is indeed capable of calculating MIPS for the highly expressive, but admittedly small, MadCow terminology.

Analysis Debugging is computationally difficult, and calculating MUPS and MIPS will in some cases fail, no matter what algorithms is used. On a more optimistic note, however, solutions can be found in fairly complex terminologies, such as DICE or MGED.

This remark is in line with reasoning in description logics in general, where the worst-case complexity tells us that reasoning is in principle intractable. But despite this very high theoretical complexity, there are very promising results in practical complexity (both for classical reasoning and for debugging). Of the four cases where our tools find solutions, two interesting pieces of information emerge: the shorter runtime of DION as compared to MUPSTER in the two cases where DION finds a solution, and the independence of the runtimes of DION and RacerPro. Since 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 runtime of RacerPro on Sumo&Cyc would suggest that DION should be able to find MIPS, but it does not. This result clearly shows that it is not the complexity of the reasoning but the number of reasoning tasks that makes bottom-up debugging difficult.

One explanation of this behavior could be in the number of unsatisfiable concepts in the six terminologies. Interestingly, the terminologies with the highest number of unsatisfiable concepts are the most difficult to debug for DION. As both DION and MUPSTER implement the same algorithm to calculate MIPS from MUPS, this algorithm cannot be the explanation for DION's problems. The real problem must be to find the MUPS.

Table 4 summarizes other properties of the four terminologies for which MUPSTER found MIPS. It shows that the number of MUPS (#mups) roughly corresponds to the number of unsatisfiable concepts, as the average number of MUPS (avrg # mups) per unsatisfiable concept does not vary significantly.

Another explanation of this artifact could be the number and size of the MUPS, as there is a direct correlation between the complexity of algorithm implemented in DICE and the size of the MUPS. The mean of the size of the MUPS for MGED would confirm this theory, but for DICE and DICE-A no significant difference can be seen.

Table 4 Properties of MUPS in the real-world terminologies

	#mups	avrg #mups	mean |mups|	std deviation |mups|
DICE-A	152	2	6.46	0.65
DICE	70	2.59	6.45	1.2
MGED	244	3.4	8.13	1.05
Geo	22	2	5.14	0.69

The lower standard deviation of the size of the MUPS for DICE-A could suggest that a higher percentage of MUPS calculations is done in lower branches of the search tree for relevant concepts (with exponential blowup of complexity).

Unfortunately, for such a study one needs the debugging output as input to the experiments, something that was outside the scope of this paper. What we investigated in more detail was whether other criteria could be responsible for particular runtime behavior. For this purpose we conducted experiments on two sets of systematically built benchmarks.

6.2 Experiments with Existing Benchmarks

The generalizability of the results of the previous set of experiments is limited because they could be severely influenced by some peculiarity of the chosen terminologies. Thus we conducted a second set of experiments based on the benchmark set from the DL98 systems competition. To get a better understanding of the influence of specific properties of terminologies on the computational properties of the respective algorithms, we ran both MUPSTER and DION against the set of 27 TBoxes translated from the DL98 test bench as described in Section 5.2.2. The results for MUPSTER and DION are summarized in Fig. 6. The graphics show the runtime per problem up to the timeout of 100 *s*.

We kept this relatively short timeout from the original DL experiments. The reason rests with the nature of the experiments. Remember that the DL benchmark was created so that it shows the weakness of an algorithm in an exponential way;

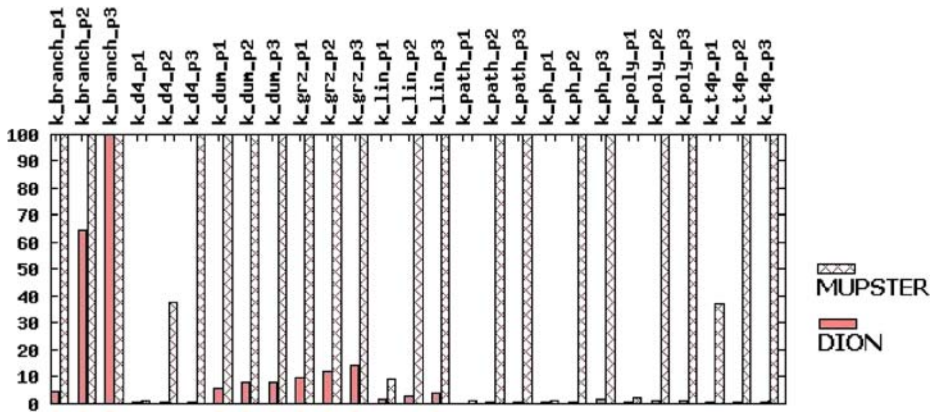

Fig. 6 Runtime (in seconds) of DION and MUPSTER on DL98 test set

hence, 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 differ from 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 behavior of DION. More concretely, we note that MUPSTER fails in all of the nine different classes to calculate even the third problem, whereas DION fails only in a single case (k_branch_p) to terminate in time.

Analysis The MUPSTER tool 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-built to point to computational difficulties in satisfiability checking; that is, they explicitly exploit structural properties of formulas and provers. More concretely, experience showed that the DL systems of the 1990s 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."

We conclude that basic optimizations can significantly improve the performance of MUPSTERbecause techniques such as lemmatizing 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 practice, however, since minimality is a nice but not strictly required feature of the MIPS.

Since DION uses optimized reasoners that can nowadays solve problems like those of the DL98 test set within milliseconds, it solved the problems easily. This success also has to do with the structure of the test set: since 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 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 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.[23] Figure 7 summarizes the runtimes of this experiment. Although the problem is in principle exponential in the size of the TBox, DION shows linear behavior in this experiment. This result also corresponds to the increasing runtime RacerPro needs for checking coherence. The explanation is relatively simple: the strategy underlying DION controls the inherent complexity of the search process in a nice way.

6.3 Experiments with Purpose-Built Benchmark

The final set of experiments to evaluate algorithms of debuggers was conducted on our own purpose-built benchmark described in Section 5.2.3. Here the focus was not

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

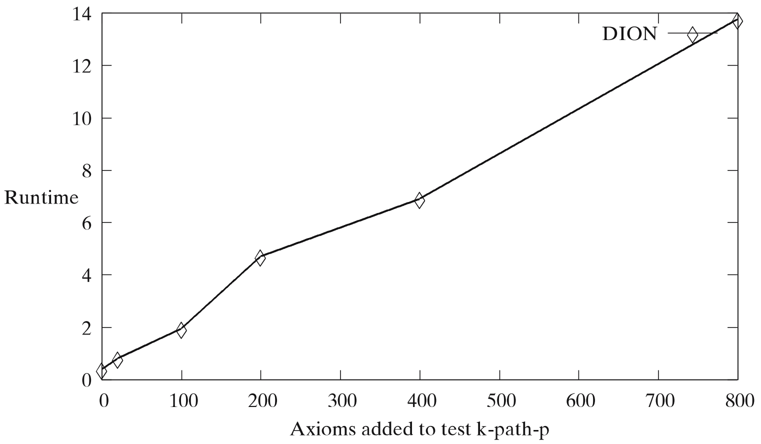

Fig. 7 Runtime (in seconds) of DION depending on the number of (unrelated) axioms to k_path_p1

on creating difficult reasoning problems but on identifying features of terminologies that would make debugging hard or easy.

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 approach allows us to get some more insights into the influence of structural peculiarities of TBoxes on the time needed for debugging.

Results For the 1,611 TBoxes tested in these experiments we checked the runtimes of the tools DION and MUPSTER for debugging and RacerPro for consistency checking. The overall average runtimes in seconds are summarized in the following table:

	MUPSTER	DION	RacerPro
average in s	2.05	1.68	1.35

These numbers show DION as the clear overall winner over MUPSTER, a result that is in line with the results of the previous experiments. We note that the overall runtimes of MUPSTER contain a call to RacerPro for a list of unsatisfiable concepts. Since this external runtime 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 runtime behavior of these two tools therefore 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, a correlation between the times of these tools also can be expected.

The results of our experiments to identify computational properties of particular classes of TBoxes are summarized in Figs. 8, 9 and 10, in which we show the average runtime (in seconds) for each tool given a particular instantiation of values for the varying features.

In Fig. 8 we compare the runtimes of our three tools on classes of TBoxes with concept lengths of 3–7. For varying concept sizes, the result shows a clear correlation

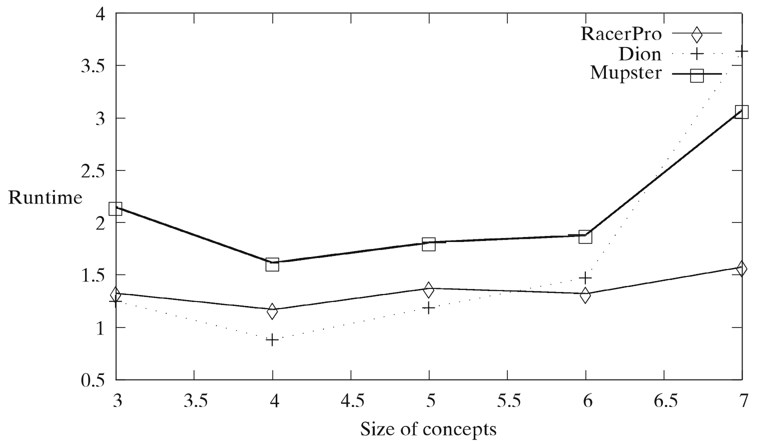

Fig. 8 Runtime (in seconds) depending on the size of the concept definitions

among the three methods. There is a slight dip in computational difficulty of the concepts of size 4, followed by an increasing runtime for all tools. 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 where the size of the TBox is varied. Here, we take TBoxes of different length into account, varying from 10 to 90 in steps of 10. The general line is an almost linear increase in runtime for all three tools up to 90 concepts.

We also conducted experiments varying the ratio between disjunctions and conjunctions and varying the ratio of negated versus non-negated atoms in the axioms of the TBox. However, these experiments did not reveal any interesting structural dependencies of the behavior of our implementations on these parameters. Hence, we omit the detailed reports of these experiments.

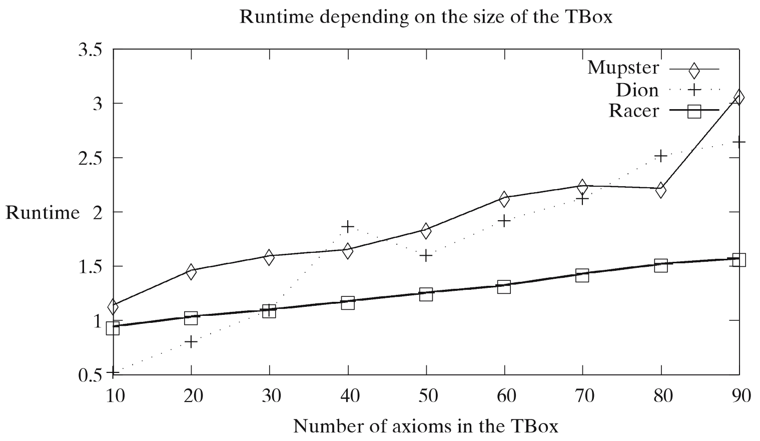

Fig. 9 Runtime (in seconds) depending on the size of the TBox

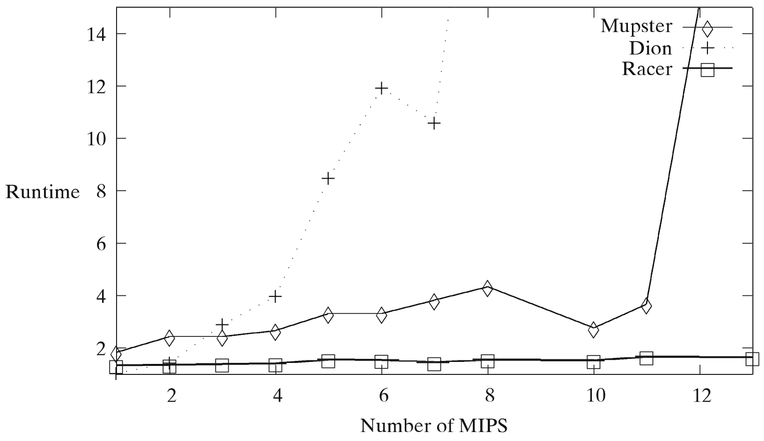

Fig. 10 Runtime (in seconds) depending on the number of MIPS

Analysis Unfortunately, none of the initial experiments seem to indicate a clear explanation for the runtime behavior of particular classes of TBoxes. A tendency can be seen that debugging becomes more difficult with increasing TBox and concept size. But another conclusion might be drawn. Figure 8 shows an increase in the runtime of DION that is much steeper than the runtimes of the other two tools. The reason rests with 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 Fig. 9 suggests that the increase in complexity of the TBox is controlled by the heuristics in a better way.

Both experiments depend on the features of the TBoxes known before debugging. This information could be useful if we want to decide automatically which tool to use for which terminology. A second class of features is those that are inherent to debugging, in other words, that can be determined only by running a debugger on the data sets. One such feature is the number of MIPS for an incoherent terminology.

Figure 10 shows the runtimes with respect to the number of MIPS of the incoherent terminologies. The results differ from the previous ones in that they show a distinct behavior for each of the three tools. RacerPro's runtime remains almost constant for increasing MIPS-size; MUPSTER shows an almost linear increase (up to the number 11); and an exponential behavior is visible for DION.

For the values above 10, the numbers have to be read with care, as there are only six examples, but the average is clear: to debug six TBoxes with more than 10 MUPS, DION needs 232 s as compared to the 50 s MUPSTER needs.

How can we explain the increasing runtime 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. Since the increasing number of MIPS is directly related to an increasing number of MUPS, the difference is that MUPSTER calculates all MUPS for an unsatisfiable concept 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 depth of the search tree of DION, which adds one exponential factor for each additional level in the tree.

As the latest experiments show, studying the properties of the results of the debug seems to be a fruitful line to explain of the runtime behavior of DION on the realistic terminologies studied in Section 6.1.

A number of open questions remain for future research:

– What is the influence of other properties of TBoxes, that is, 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 paper we focused on a priori properties of TBoxes, that is, 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.

6.4 Answering the Research Questions

Let us end this section by answering the research questions we presented in Section 5. 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 terminologies can be debugged in some, but not all, cases. Overall, MUPSTER shows better performance on our real-world examples than does DION, probably because of the large number of MIPS. On the other hand, the other two benchmarks 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. Moreover, the DL benchmark where complex reasoning is required shows that optimized reasoners can be used for efficient (if incomplete) debugging, but also that nonoptimized 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 any complete approaches might scale up in this case.

What makes an incoherent terminology difficult to debug? The larger an incoherent terminology, the more difficult it becomes. It seems that the increase in runtime is linear with increasing TBox size, but exponential in the average size of the concepts. 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 modeling errors.

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

7 Concluding Remarks

In this paper we have presented a formal characterization for debugging and diagnosis, and we defined two algorithms for computing MIPS, which are directly useful and provide the basis for the calculation of diagnoses. We also provided an evaluation of these algorithms, done in two ways. First, we studied the effectiveness of our proposal in a realistic setting on two real-world terminologies. Second, we conducted a controlled experimental analysis to get a better understanding of the computational properties of the debugging problem and our algorithms for solving it.

The results of these evaluations are mixed. In general, we can conclude that our proposed notions for debugging are useful in practice. Our experience shows that a number of heuristics and additional measures, such as the weight of MIPS or the pinpoints, are crucial in practical applications.

In terms of computational behavior our results are 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 in our controlled experiments, fails to compute MIPS for a number of real-world terminologies, where MUPSTER comes up with a solution within minutes. On the other hand, the more complex reasoning becomes, the worse MUPSTER's performance is.

Our experiments yielded a 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. Several open questions remain, mostly related to the combination of properties. This means that an estimation of the runtime of the debuggers on the basis of the structure of the incoherent terminology is still very difficult.

An interesting result is the comparison of the two methods: top-down and bottom-up, which both have their merits in different cases. This might be an argument to extend the MUPSTER approach to more expressive languages and nonrestricted ontologies.

We note as a final remark that since this article was written, extensions of our methods and alternative implementations have been published (most notably [17] and [12]). Availability of these tools will offer the opportunity for more extensive benchmarking, on more expressive terminologies and ontologies.

Acknowledgements We thank the anonymous referees for their detailed, knowledgeable, and thus very valuable comments. This research was partially funded by the SEKT project under EU-IST grant nr. 506826, and by the Netherlands Organisation for Scientific Research under grant nr. 634.000.020 and 220.80.001.

References

1. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, Cambridge, MA (2003)
2. Baader, F., Hollunder, B.: Embedding defaults into terminological representation systems. J. Autom. Reason. **14**, 149–180 (1995)
3. Console, L., Dressler, O.: Model-based diagnosis in the real world: lessons learned and challenges remaining. In: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI'99, pp. 1393–1400 (1999)

4. de la Banda, M.G., Stuckey, P.J., Wazny, J.: Finding all minimal unsatisfiable subsets. In: Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming, ACM-SGPLAN'03, pp. 32–43 (2003)
5. Friedrich, G., Shchekotykhin, K.M.: A general diagnosis method for ontologies. In: Proceedings of the 4th International Semantic Web Conference, ISWC'05. LNCS, vol. 3729, pp. 232–246 (2005)
6. Greiner, R., Smith, B.A., Wilkerson, R.W.: A correction to the algorithm in Reiters theory of diagnosis. Artif. Intell. **41**(1), 79–88 (1989)
7. Hansson, S.: A Textbook of Belief Dynamics. Kluwer, Dordrecht (1999)
8. Hansson, S.O., Wassermann, R.: Local change. Stud. Log. **70**(1), 49–76 (2002)
9. Horrocks, I., Patel-Schneider, P.F.: DL systems comparison. In: Proceedings of the 1998 Description Logic Workshop (DL'98), pp. 55–57 (1998)
10. Huang, Z., van Harmelen, F.: Reasoning with inconsistent ontologies: evaluation. Project Report D3.4.2, SEKT (2006)
11. Huang, Z., van Harmelen, F., ten Teije, A.: Reasoning with inconsistent ontologies. In: Proceedings of the International Joint Conference on Artificial Intelligence – IJCAI'05 (2005)
12. Kalyanpur, A.: Debugging and repair of OWL ontologies. Ph.D. thesis, University of Maryland (2006)
13. Kalyanpur, A., Parsia, B., Cuenca-Grau, B., Sirin, E.: Beyond axioms: fine-grained justifications for arbitrary entailments in OWL-DL. In: Description Logic Workshop (DL'06) (2006)
14. Kalyanpur, A., Parsia, B., Sirin, E., Hendler, J.: Debugging unsatisfiable concepts in OWL ontologies. J. Web Sem. **3**(4) (2005)
15. Massacci, F., Donini, F.M.: Design and results of TANCS-2000 non-classical (Modal) systems comparison. In: International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Tableau'00, pp. 52–56 (2000)
16. Meyer, T., Lee, K., Booth, R.: Knowledge integration for description logics. In: Proceedings of the Twentieth National Conference on Artificial Intelligence, AAAI'05, pp. 645–650 (2005)
17. Meyer, T., Lee, K., Booth, R., Pan, J.Z.: Finding maximally satisfiable terminologies for the description logic ALC. In: Proceedings of the 21st National Conference on Artificial Intelligence, AAAI'06 (2006)
18. Nebel, B.: Terminological reasoning is inherently intractable. Artif. Intell. **43**, 235–249 (1990)
19. Quine, W.: The problem of simplifying truth functions. Am. Math. Mon. **59**, 521–531 (1952)
20. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. **32**(1), 57–95 (1987)
21. Schlobach, S.: Debugging and semantic clarification by pinpointing. In: Proceedings of the 2nd European Semantic Web Conference – ESWC'05. LNCS, vol. 3532, pp. 226–240 (2005)
22. Schlobach, S.: Diagnosing terminologies. In: Proceedings of the Twentieth National Conference on Artificial Intelligence, AAAI'05, pp. 670–675 (2005)
23. Schlobach, S., Cornet, R.: Non-standard reasoning services for the debugging of description logic terminologies. In: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, IJCAI'03 (2003)
24. Schlobach, S., Cornet, R., Huang, Z.: Inconsistent ontology diagnosis. Project Report D3.6.2, SEKT (2006)