# Debugging OWL Ontologies - A Reality Check

Heiner Stuckenschmidt

Universität Mannheim, Institut für Informatik,
A5, 6 68159 Mannheim, Germany
e-mail: heiner@informatik.uni-mannheim.de

**Abstract.** One of the arguments for choosing description logics as the basis for the Web Ontology Language is the ability to support the development of complex ontologies through logical reasoning. Recently, significant work has been done on developing advanced methods for debugging erroneous ontologies that go beyond the pure identification of inconsistencies. While the theory of these methods has been studied extensively little attention has been paid to the application of these Methods in practice. In this paper, we evaluate existing implementations of advanced methods for debugging description logic ontologies. We show that most existing systems suffer from serious problems with respect to scalability but surprisingly enough also with respect to the correctness of the results on certain test sets. We conclude that there is a need for further improvements of existing systems in particular with respect to bridging the existing gap between theory and practice of debugging ontologies.

## 1 Introduction

Description logics have been at the heart of semantic web technologies as a mechanism for formalizing ontologies. The ability to detect inconsistencies in ontology models is one of the most often quoted advantages of description logics and this argument has been extensively used to justify the use of description logics as a foundation for the web ontology language OWL. It has turned our very quickly, though, that being able to detect inconsistencies is only half the battle. Once an inconsistency has been found, providing an explanation of its origin and suggestions for resolving the inconsistency are equally important. This is the case because in complex or simply large ontologies it is often not feasible for the user to spot reason for an inconsistency. Providing support for this task is essential to the success of expressive ontology languages such as OWL because users will not be willing to use languages that are too complex for them to handle. A number of approaches have been proposed to deal with the problem of identifying the origins of problems and in description logic based ontologies and suggesting fixes for these problems. Early approaches contributing to the problem are methods for generating explanations of subsumption proofs in description logics that can be applied to the analysis of inconsistent concepts by explaining the subsumption relation of the respective concept and the empty concept. These approaches are normally based on the concepts of natural deduction, that is, they try to construct deductive consisting of a series of applications of deduction rules. These rules almost directly translate to an explanation in controlled language. More recently, a new line of research has evolved that applies principles of model-based diagnosis [9] to the problem. These approaches interpret inconsistencies as symptoms and use them to derive minimal sets of axioms

that cause these symptoms thereby constituting a diagnosis for the problem. Existing approaches in this area can be distinguished into black-box approaches that use an external reasoner as a black box for testing whether certain combinations of axioms still cause an inconsistency and white-box approaches that modify the proof procedure of existing reasoners in such a way that debugging information is recorded during the reasoning process [7]. While the theoretical aspects of debugging ontologies have been investigated in details, there is relatively little experience regarding the performance of debugging methods in practice. Most of the methods have been applied to example ontologies individually to show their feasibility but so far no attempts towards a systematic evaluation of existing debugging tools have been reported in the literature. In this paper, we take a first step towards a systematic evaluation of debugging tools. In particular, we apply existing debugging tools to a benchmark dataset that has been created for this purpose and discuss the results of the different approaches. In particular, we look at the functionality of the tools, completeness and correctness and minimality of the debugging results as well as run-time needed. Based on the results of our experiments, we draw rather negative conclusion about the practical merits of currently existing debugging tools. In particular, we show that most systems have serious performance problems and often even produce incorrect results. Only one system seems to be ready for practical applications. The paper is structured as follows. We first recall some basic definitions of description logics and the theory of ontology diagnosis. Afterwards we briefly describe a number of existing debugging systems that we have evaluated. We then describe the benchmark dataset and the experimental setting we used as well as the results of our debugging experiments. We close with a critical review of the systems evaluated and some general conclusions about the state of the art in ontology debugging.

## 2 Preliminaries

Before we take a closer look at existing tools for debugging ontologies, we first have to review some basic notions of description logics and the theory of debugging description logics.

### 2.1 Inconsistency, Unsatisfiability and Incoherence

Based on the model theoretic semantics of description logics (compare [1]), a number of formal properties have been defined that form the basis for any debugging effort. In particular, there are several properties of an ontology that indicate different kinds of conflicts in the formalization that have to be resolved by a debugging method.

The most basic property indicating a conflict in an ontology is inconsistency of the overall model. Inconsistency is an unwanted property, because inconsistent theories logically imply any fact thus making the definitions mostly useless in the context of logical reasoning.

**Definition 1 (Inconsistency).** *A description logic terminology $\mathcal{T}$ is said to be inconsistent if there is no interpretation $\mathcal{I}$ that is a model for $\mathcal{T}$.*

While the notion of inconsistency exists for any kind of logic, there are weaker forms of conflicts that occur in the context of description logic. In particular, a terminology can only become inconsistent in the presence of an A-Box. In any case where

no A-Box exists, the interpretation mapping all concepts and relations to the empty-set is always a model of the terminology. A property that captures conflicts that solely exist in the terminology is the notion of concept unsatisfiability. A concept is said to be unsatisfiable if its definition does not allow a consistent instantiation.

**Definition 2 (Unsatisfiable Concept).** *Let $\mathcal{T}$ be a terminology and $C$ a concept in $\mathcal{T}$. $C$ is said to be unsatisfiable if for all models $\mathcal{I}$ of $\mathcal{T}$ we have $C^{\mathcal{I}} = \emptyset$.*

While unsatisfiable concepts to not make the ontology as such inconsistent still indicate potential problems, because adding instances to the ontology might result into an inconsistent model with all the unwanted consequences. As the role of ontologies in practical applications is mostly in to provide additional semantics for instance data, an ontology containing unsatisfiable concepts is often as bad as an inconsistent one. The existence of unsatisfiable concepts in an ontology is also called incoherence.

**Definition 3 (Incoherence).** *Let $\mathcal{T}$ be a terminology. $\mathcal{T}$ is called incoherent if there is at least one concept $C$ in $\mathcal{T}$ and $C$ is unsatisfiable.*

### 2.2 Debugging Description Logics

In [10] Schlobach and others provide some basic definitions that are commonly seen as the basis for ontology debugging. These definitions are partly taken from earlier work on general diagnosis and have been adopted to the need of description logic terminologies as a target for the diagnosis. The corresponding theory takes unsatisfiable concepts as a starting point and provides a number of definitions that correspond to subtasks in the debugging process. The first important concept is that of a minimal unsatisfiability preserving Subterminology (MUPS). Informally a MUPS is a minimal set of concept definitions from a terminology that together make a concept unsatisfiable.

**Definition 4 (MUPS).** *Let $\mathcal{T}$ be a terminology. A Terminology $\mathcal{T}' \subseteq \mathcal{T}$ is a MUPS for a concept $C \in \mathcal{T}$ if C is unsatisfiable wrt. $\mathcal{T}'$ and C is satisfiable wrt. all terminologies $\mathcal{T}'' \subseteq \mathcal{T}'$.*

MUPS roughly correspond to the notion of conflict set in classical diagnosis and computing MUPS for a given unsatisfiable concept is one of the most basis tasks for any debugging system as the MUPS can be assumed to contain at least one wrong definition that needs to be corrected or removed to fix the conflict in the ontology.

MUPS provide a basis for computing minimal incoherence preserving subterminologies (MIPS). MIPS are minimal sets of concept definitions that make any of the concepts unsatisfiable. MIPS are interesting, because they correspond to minimal conflict sets with respect to the problem of incoherence.

**Definition 5 (MIPS).** *Let $\mathcal{T}$ be a terminology. A Terminology $\mathcal{T}' \subseteq \mathcal{T}$ is a MIPS for $\mathcal{T}$ if $\mathcal{T}'$ is incoherent and all $\mathcal{T}'' \subseteq \mathcal{T}'$ are coherent.*

MUPS and MIPS provide a basis for computing minimal diagnosis for the problems (unsatisfiability or incoherence respectively). In particular, the hitting set algorithm originally proposed by Reiter [9] can be used to compute minimal diagnosis from both MIPS and MUPS. A minimal diagnosis, in this case is defined as a minimal set of concept definitions that if removed from the ontology solves the conflict at hand.

**Definition 6 (Diagnosis).** *Let $\mathcal{T}$ be an incoherent terminology. A diagnosis for the incoherence problem of $\mathcal{T}$ is a minimal set of axioms $\mathcal{T}'$ such that $\mathcal{T} - \mathcal{T}'$ is coherent. Similarly, a diagnosis for unsatisfiability of a single concept $C$ in $\mathcal{T}$ is a minimal set of axioms $\mathcal{T}'$ , such that C is satisfiable with respect to $\mathcal{T} - \mathcal{T}'$.*

As computing diagnosis is very expensive computationally, Schlobach and others propose to use approximate notions of diagnoses called pinpoints. We will not go into details here but just think in terms of possibly approximate diagnoses.

## 3 Debugging Systems

In our evaluation we used ontology debugging tools that are freely available on the web. We can distinguish between pure debugging systems that read an ontology in a standardized format and computes certain features such as MUPS, MIPS and approximate diagnoses and ontology editors with an integrated debugging facility. In the following, we briefly describe the systems included in our experiments.

*MUPSter* The MUPSter System is an experimental implementation of the white-box debugging method described in [10], in particular for computing MUPS, MIPS and approximate diagnoses. It was rather meant as a proof of concept than a debugging tool for real use. The System has a number of limitations connected with the expressiveness of the ontologies it is able to handle. In particular, the MUPSter System is only guaranteed to work on unfoldable Terminologies in the logic $\mathcal{ALC}$. Further, the system does not directly work on OWL. It requires input ontologies to be encoded using the KRSS or the DIG format for description logics. We nevertheless included the system in our evaluation in order to test what the impact of these limitations are in practice. The version of the MUPSter System used is available online[1]

*DION* The DION System is an implementation of black-box debugging that has been developed in the context of the SEKT project []. The system is completely implemented in SWI-PROLOG and uses the DIG interface for communicating with an external reasoner. In our experiments, we used version 1.7.14 of the RACER system as an external reasoner for all black-box systems to guarantee fairness. Similar to the MUPSter System, DION works on ontologies in the DIG format and as MUPSter computes MUPS, MIPS and approximate diagnoses. The tool comes with a translation function for converting OWL ontologies to the DIG format that we used for generating input data for MUPSter and DION. The version of DION used in the evaluation is available online[2].

*RADON* The RADON system that implements a debugging method that is inspired by belief revision. The system uses a weakening based revision function that basically removes certain axioms from the model such that the conflict is resolved and the amount of information removed is minimal with respect to some measure of minimality [8]. The axioms to be deleted correspond to MUPS or MIPS respectively. This general approach allows the system to debug incoherent as well as inconsistent ontologies. The system uses the KAON 2 Reasoner for checking consistency and coherence of ontologies and

[1] http://www.few.vu.nl/ schlobac/software.html
[2] http://wasp.cs.vu.nl/sekt/dion/

can therefore be seen as a white-box approach. A special feature of the system, that is not used in this evaluation, however, is the ability to add knowledge to an existing model revise the existing model based on the new information. The additional information that the new information is always correct helps to restrict the search space and can therefore be assumed to improve the debugging result. The version of RADON used in this evaluation is available online[3]

*Protege* Being the most often used ontology editor around, some effort has been spent on extending Protege with a number of mechanisms for supporting the user in building correct models. One of these mechanisms is a debugging facility that has been developed in the context of the CO-ODE project [12]. In contrast to the systems mentioned so far, the method does not rely on a sound formal basis in terms of diagnostic reasoning or belief revision. Instead Protege uses a set of heuristic rules that are applied to the description of an unsatisfiable class and derive possible reasons for the unsatisfiability based on certain modeling patterns. The generated explanation does not always correspond to a MUPS but it can also contain natural language explanations such as 'The superclass is unsatisfiable'. As a results, the method is not complete. The authors argue, however, that this is not a drawback in practical situations as most common modeling mistakes are covered by the rules. The debugging system we used in the evaluation is part of Protege 4.0 which is available online[4].

*SWOOP* The SWOOP editor developed at the University of Maryland also has an integrated debugging mechanism. It benefits from the tight integration with the Pellet reasoner that has been extended with white-box methods for debugging unsatisfiable concepts [5]. The mechanism used in the evaluation is the general explanation function that can be used to generate explanations for any derived subsumption statement including subsumption by the empty concept. The explanation is generated on the basis of a trace of the tableaux proof and consists of a number of axioms that correspond to a MUPS for the respective concept. The version of the debugging method used in this evaluation is contained in SWOOP version 2.3 and can be obtained online[5].

## 4 Experiments

The goal of our experiments was to find out how well the different systems cope with realistic debugging tasks at different levels of difficulty. For this purpose, we created a benchmark dataset consisting of ontologies about the same domain that have been designed by different people. We introduced inconsistencies in these ontologies using automatic matching systems for conducting a pairwise integration of these ontologies. The resulting merged ontologies contain unsatisfiable concepts which are a result of known errors in the matching process that have been documented before. This has the advantage that despite not dealing with toy examples we have a pretty good idea of which axioms are the source of the problem and should be detected by the debugging systems. In the following we describe the data used as well as the experimental setting and the results in more details.

[3] http://radon.ontoware.org/
[4] http://protege.stanford.edu/
[5] http://code.google.com/p/swoop/

### 4.1 Datasets

We evaluated the debugging systems using the OntoFarm Dataset. It consists of a set of ontologies in the domain of conference organization that have been collected by researchers of the Knowledge Engineering Group at the University of Economics Prague [11][6]. The ontologies have been built manually by different persons some based on experiences with organizing conferences. Some are based on existing conference management tools. In general they cover the structure of a conference, involved actors, as well as issues connected with the submission and review process.

We used the automatic matching systems falcon [4] and CTXmatch [2] to compute hypotheses for equivalence mappings between pairs of these ontologies. We translated the mapping hypotheses into equivalence statements and created new ontologies consisting of the union of the mapped ontologies and the equivalence statements. Due to mistakes in the matching process, these union ontologies often contain unsatisfiable concepts. Based on the evaluation of the mappings carried out in the context of the Ontology Alignment Evaluation Initiative [3] we know exactly which axioms are erroneous and should ideally be detected by the debugging systems.

### 4.2 Concrete Test Cases

From the data described above, we chose a number of test cases. These test cases correspond to those combinations of ontologies using mappings of the two systems mentioned above that contain unsatisfiable concepts. It turned out that these contain test cases of different levels of complexity ranging from very simple tests that almost all systems in could manage to very complex ones only some systems solved correctly. As indicators for the complexity of a debugging problem we used the following criteria

1. The size of the ontology in terms of number of classes
2. the number of unsatisfiable concepts in the ontology
3. the average number of MUPS per unsatisfiable concept
4. the kinds of axioms that occur in a MUPS

All of these criteria have an impact on the difficulty of the problem as they contribute to the size of the combinatorial problems that has to be solved at the different levels of the debugging process. According to these criteria we distinguish between simple, medium and hard problems. In total, we used the following test cases three of which we classified as easy, three as medium and two as hard tasks.

### 4.3 Experimental Setting

We first used each of the debugging tools in the evaluation to check the test data sets for unsatisfiable concepts. This may seem to be a trivial, but it turned out that some of the tools already had problems finding all unsatisfiable concepts due to limitations on subset of OWL constructs the tool handles correctly. We then used each tool to compute MUPS for all the unsatisfiable concepts found and compared the results to each other. In contrast to the first step, we were not able to compare the results with a provably correct result. We rather compared the results of the different tools and did a manual

[6] The ontologies are available from `http://nb.vse.cz/~svabo/oaei2006/`.

inspection of the results proposed. In some cases, we tested correctness and minimality of results by systematically removing axioms from the ontology an rerunning the test. In this way we could show that some of the results are not minimal (they still lead to unsatisfiability after axioms had been removed from the MUPS) and some are not even correct (after fixing the known problem, the tool still computed a MUPS for a conflict that did not exist any more). We also explored more advanced features of the tools in a more ad hoc fashion as a systematic evaluation of MIPS and diagnosis was impossible due to the extreme run-times that often forced us to interrupt the computation after a couple of hours.

## 5 Results

The results of our experiments were not very encouraging and revealed a number of problems some of the tools face when being applied to unforseen tasks.

### 5.1 Performance

Our experiments confirmed the observation that black-box approaches for debugging suffer from serious performance problems due to the combinatorial explosion of reasoner calls needed for computing the diagnosis. The Black-Box Systems in the test, namely RADON and DION suffered from very long run times. In the case of DION we faced serious performance problems with the result that only the smallest test case could be solved by the system. In the second case the computer ran out of memory. All other test where interrupted after 12 hours(!). This negative result can probably be explained by the use of PROLOG as an implementation language that is known to be rather resource consuming and by the inherent complexity of the black-box approach the system implements. Acceptable performance was observed by Protege and SWOOP which is not surprising as these Systems aim at supporting user interaction which requires real-time behavior.

### 5.2 Detecting Unsatisfiable Concepts

The most surprising insight for us was that some of the tools already have problems with the very basic task of finding unsatisfiable concepts which we assumed to be a solved problem. As table 1 shows the only tool that correctly identified all unsatisfiable concepts was SWOOP while Protege and MUPSter missed some of the problems. In the case of MUPSter this was expected as the tool only covers a sublanguage of OWL. Looking at the impact of this restriction it turns out that the system still works for most simple cases and still finds most of the unsatisfiable concepts in the complex ones. Protege seems to loose information in the communication between the editor and the external reasoner. So far we did not find out what exactly is the problem. Possible problems are with the use of datatype properties in definitions and nested restrictions that sometimes seem to get lost. Even more surprising is the fact that the RADON system overshoots and marks concepts as unsatisfiable that are not. A closer look revealed that the System does not correctly interpret datatype properties. As a consequence, all concepts whose definitions involved datatype properties were named unsatisfiable which lead to the surprising result.

| | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** | **Test 6** | **Test 7** | **Test 8** |
|---|---|---|---|---|---|---|---|---|
| Protege | 2/2 | 2/2 | 2/2 | **0/1** | 4/4 | 7/7 | **13/18** | 39/39 |
| SWOOP | 2/2 | 2/2 | 2/2 | 1/1 | 4/4 | 7/7 | 18/18 | 39/39 |
| RADON | **38/2** | 2/2 | 2/2 | 1/1 | 4/4 | 7/7 | 18/18 | 39/39 |
| MUPSter | 2/2 | 2/2 | 2/2 | 1/1 | 4/4 | 7/7 | **13/18** | **35/39** |
| DION | 2/2 | *ooM* | - | - | - | - | - | - |

**Table 1.** Unsatisfiable concepts determined

### 5.3 Coverage of the Debugging Task

The functionality of the debugging tools used differs, in order to be able to compare them, we use the formal definitions of MUPS, MIPS and diagnosis as a basis of our comparison. It turns out that all systems compute some kind of MUPS - in the case of Protege the result of the debugging process can be seen as an approximation of a single MUPS. All other systems compute all MUPS. MIPS are only computed by the pure debugging systems in the evaluation and (approximate) diagnoses are only provided by MUPSter and DION. We therefore use the MUPS as the main basis for our comparison.

### 5.4 Computing MUPS

The actual debugging functionality of the tools was hard to compare as we do not have provably correct results for the example ontologies. A manual inspection of the debugging output, however, revealed some fallacies of the tools. While the restriction of the MUPSter tool to a subset of OWL only led to a small degradation of the results with respect to finding unsatisfiable concepts, the tool only produced trivial MUPS. In particular, the MUPS only contained the bottom concept. This means that it is not applicable to more expressive ontologies. The DION tool computed a correct MUPS and even MIPS and an approximate diagnosis for Test 1 but as mentioned above failed on the other tests due to memory and runtime problems. The PROTON Tool only worked correctly on some of the tests, on others, in particular Test 1 the MUPS computed by PROTON were neither minimal nor unsatisfiability preserving. For simple problems that only had a single MUPS the tool came up with more than ten different MUPS most of which contained almost the entire ontology. This can probably be explained by the problems with determining unsatisfiability which for the case of a black-box approach completely messes up the revision. With respect to the heuristic debugging approach implemented in Protege we detected two major problems. The first is the creation of cyclic explanations for equivalent concepts. Quite often, the tool explains the unsatisfiability of a concept with the unsatisfiability of another equivalent concept and vice versa without pointing to the actual problem which resides in the definitions of these concepts. Further, we found errors in the definition of the heuristic rules. In particular, there were cases where domain and range definitions were mixed up leading to wrong explanations. We did not check all the debugging rules for correctness but there could be more problems like this. The only tool that seems to correctly compute MUPS for all of the test cases was the SWOOP system, although, as mentioned above we did not formally check the correctness and minimality of all the results.

## 6 Conclusions

The conclusion we have to draw from our experiments is that most existing approaches for debugging OWL ontologies are mostly of theoretical interest and that more work is needed to make these methods applicable in practice. The lessons learned from the evaluation concern the following three points

*Well-foundedness* The results of the heuristic debugging approach implemented in Protege show the need for the use of theoretically well founded methods as a basis for ontology debugging. While the authors of the method argue that it covers most of the relevant cases, we found out that if applied outside there cases, which in the case of Protege are modeling errors, the systems performance is rather poor. In our test cases the errors are rather matching than modeling errors. As a consequence, unsatisfiable concepts almost always had an equivalent concept, a situation Protege cannot deal with correctly. Our experiments also clearly shows the problem of heuristic approaches in terms of correctness. The error we spotted only turned up when the tool was used in an unforseen situation. If heuristic methods are used, more systematic testing is required to exclude these kinds of problems.

*Robustness* To our surprise also well-founded methods produced wrong results. In our cases this was a result of the inability of the underlying reasoning services to correctly interpret the complete OWL standard. What we missed in this context was a graceful degradation of the systems performance. In most cases, the inability to interpret a certain definition led to a completely wrong or trivial result. An interesting topic for future research is in methods with better robustness properties, i.e. methods that do not completely fail on input they cannot handle completely but still compute diagnoses that are correct and minimal with respect to the subset understood by the system. On a more technical level, existing systems should pay more attention to non-logical features of OWL, in particular to the use of datatype properties. These seem to be the main source of practical problems encountered in our experiments.

*Performance* Run-time performance is a major issue in the context of a practical application of debugging methods, especially because debugging will normally be an interactive process. Our experiments confirmed the common knowledge that black-box approaches suffer from their computational complexity as they normally require an exponential number of calls to an external description logic reasoner. None of the black-box systems in the test showed a performance that would allow for an interactive debugging strategy. Another interesting observation is that many black-box methods discussed in the literature are not available as tool, probably for this reason. The experiments also showed that besides the theoretical complexity, the implementation strategy is also an issue. The poor performance of the DION System clearly shows that the PROLOG-based solution is only useful as a proof of concept, but not ready for real applications. Not surprisingly, the two debugging tools integrated in ontology editors did not use black-box approaches.

In summary, the way to move forward is to further develop white-box approaches for ontology debugging focussing on the robustness of the methods and the coverage of the OWL Standard. Further, there is a case for specialized debugging tools such

as the one reported in [6] that addresses the specific problem of debugging mappings between ontologies and therefore shows a much better performance on the test datasets, mainly because it restricts the search space to axioms representing mappings while the ontologies themselves are assumed to be correct. Such a mechanism allowing the user to focus the debugging process to a subpart of the ontology would clearly enhance existing approaches.

## References


1. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2002.
2. Paolo Bouquet, Luciano Serafini, and Stefano Zanobini. Peer-to-peer semantic coordination. *Journal of Web Semantics*, 2(1), 2005.
3. Jerome Euzenat, Malgorzata Mochol, Pavel Shvaiko, Heiner Stuckenschmidt, Ondrej Svab, Vojtech Svatek, Willem Robert van Hage, and Mikalai Yatskevich. First results of the ontology alignment evaluation initiative 2006. In Richard Benjamins, Jerome Euzenat, Natasha Noy, Pavel Shvaiko, Heiner Stuckenschmidt, and Michael Uschold, editors, *Proceedings of the ISWC 2006 Workshop on Ontology Matching*, Athens, GA, USA, November 2006.
4. Wei Hu, Gong Cheng, Dongdong Zheng, Xinyu Zhong, and Yuzhong Qu. The results of falcon-ao in the oaei 2006 campaign. In *Proceedings of the ISWC 2006 Workshop on Ontology Matching*, Athens, GA, USA, November 2006.
5. Aditya Kalyanpur, Bijan Parsia, Evren Sirin, and James Hendler. Debugging unsatisfiable classes in owl ontologies. *Journal of Web Semantics*, 3(4), 2005.
6. Christian Meilicke, Heiner Stuckenschmidt, and Andrei Tamilin. Repairing ontology mappings. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*, Vancouver, Canada, 2007.
7. Bijan Parsia, Evren Sirin, and Aditya Kalyanpur. Debugging owl ontologies. In *Proceedings of the 14th international World Wide Web Conference*, page 633640, Chiba, Japan, 2005.
8. Guilin Qi and Peter Haase. Consistency model for networked ontologies. NEON Deliverable D1.2.1, The NEON Project, 2007.
9. Ray Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.
10. Stefan Schlobach and Ronald Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, page 355362, Acapulco, Mexico, 2003.
11. Ondrej Svab, Svatek Vojtech, Petr Berka, Dusan Rak, and Petr Tomasek. Ontofarm: Towards an experimental collection of parallel ontologies. In *Poster Proceedings of the International Semantic Web Conference 2005*, 2005.
12. Hai Wang, Matthew Horridge, Alan L. Rector, Nick Drummond, and Julian Seidenberg. Debugging owl-dl ontologies: A heuristic approach. In *Proceedings of the International Semantic Web Conference 2005*, pages 745–757, 2005.

## Appendix A: Test Cases

*Test 1 (easy)*

**Ontologies** CRS and PCS, matched with CTXmatch
**Size** Small (39 Concepts)
**Problems** A few (2 Unsatisfiable Concepts)
**Complexity** low (one MUPS per Concept, only subsumption and negation involved)

*Test 2 (easy)*

**Ontologies** CRS and SIGKDD, matched with falcon
**Size** Small (48 Concepts)
**Problems** A few (2 Unsatisfiable Concepts)
**Complexity** low (one MUPS per Concept, only subsumption and negation involved)

*Test 3 (easy)*

**Ontologies** CRS and CONFTOOL, matched with falcon
**Size** Medium (64 Concepts)
**Problems** A few (2 Unsatisfiable Concepts)
**Complexity** low (one MUPS per Concept, only subsumption and negation involved)

*Test 4 (medium)*

**Ontologies** CRS and CMT, matched with CTXmatch
**Size** Small (44 Concepts)
**Problems** One (1 Unsatisfiable Concepts)
**Complexity** medium (one MUPS per Concept, subsumption, equivalence, union and negation involved)

*Test 5 (medium)*

**Ontologies** CONFTOOL and EKAW, matchesd by falcon
**Size** larger (112 Concepts)
**Problems** some (7 Unsatisfiable Concepts)
**Complexity** medium (1-5 MUPS per concept, only subsumption and negation involved)

*Test 6 (medium)*

**Ontologies** SIGKDD and EKAW, matched by falcon
**Size** larger (123 Concepts)
**Problems** some (4 Unsatisfiable Concepts)
**Complexity** medium (2 MUPS per Concept, subsumption negation and domain restrictions involved)

*Test 7 (hard)*

**Ontologies** CMT and CONFTOOL, matched by CTXmatch
**Size** medium (68 Concepts)
**Problems** many (18 Unsatisfiable Concepts)
**Complexity** high (2 to 5 MUPS per Concept, subsumption, negation, domain restrictions, inverse and number restrictions involved)

*Test 8 (hard)*

**Ontologies** CONFTOOL and EKAW, matched by CTXmatch
**Size** larger (112 Concepts)
**Problems** a lot (39 Unsatisfiable Concepts)
**Complexity** high (2 to 6 MUPS per Concept, subsumption, negation, number restrictions, domain restrictions, inverse and existential quantifiers involved)