Are Query-Based Ontology Debuggers Really Helping Knowledge Engineers?

Patrick Rodler*, Dietmar Jannach, Konstantin Schekotihin, Philipp Fleiss

Department of Applied Informatics
AAU Klagenfurt, 9020 Klagenfurt, Austria
[firstname.lastname]@aau.at

Abstract

With the spread of Semantic Web technology, applications that are built upon explicitly encoded domain knowledge have regained popularity in recent years. Since the underlying knowledge bases can easily become large and complex, it is not uncommon that such knowledge bases contain faults. Correspondingly, a number of knowledge base debugging approaches, in particular for ontology-based systems, were proposed throughout recent years. Query-based debugging is an interactive approach that involves knowledge engineers answering a series of questions and then uses the provided answers to localize the true cause of an observed problem. Concrete implementations of this approach exist, such as the *OntoDebug* plug-in for the ontology editor Protégé. We conducted different user studies to assess the practical value and the limitations of such an interactive approach since typical simulation-based evaluations are not fully informative in this regard. One main insight from the studies is that query-based debugging is indeed more efficient than an alternative algorithmic debugging approach based on test cases. We also observed that users frequently made errors in the process, which highlights the importance of a careful design of the queries that are asked to the users.

Keywords: Knowledge Base Debugging, Interactive Debugging, User Study, Ontologies, Model-based Diagnosis, Protégé, Ontology Debugging Tool

*Corresponding author

1. Introduction

Systems that are built upon Artificial Intelligence (AI) techniques are often classified into two categories: *(i)* systems that are designed to automatically *learn from data*, and *(ii)* systems based on explicitly *encoded domain knowledge* and automated inference services. Knowledge-based software systems are typical representatives of the latter form of AI with a number of successful applications in various domains such as planning and scheduling, medical advice-giving systems, product configuration, or recommender systems [1, 2, 3, 4, 5].

The correctness of the decisions and suggestions made by a knowledge-based system depends directly on the ability of an expert to formulate and maintain a knowledge base (KB) that describes the application domain. Both knowledge formalization and maintenance can be challenging due to *(i)* the cognitive complexity of the task and *(ii)* the size and complexity of the resulting knowledge base—e.g., biomedical ontologies as found on BioPortal[1] sometimes contain thousands of axioms. The results reported, e.g., in [6, 7, 8, 9] suggest that people make systematic faults when writing or interpreting logical sentences. Furthermore, in some cases, knowledge bases are constructed in a collaborative manner by multiple contributors, which is another potential source of faults [10, 11, 12].

Overall, given that unintended or contradictory specifications are likely to occur in such knowledge bases, it is essential to provide experts with appropriate tools for fault detection, localization, and repair. Over the last decades researchers suggested different techniques and implemented a number of assisting tools for these tasks. Many of these techniques are based on the principles of model-based diagnosis (MBD) [13], which is a versatile fault localization method with a range of applications, e.g., in the context of electronic circuits, declarative programs, knowledge bases and ontologies, workflow specifications,

[1] http://bioportal.bioontology.org

as well as programs written in domain-specific and general-purpose languages [14, 15, 16, 17, 18, 19, 20, 21].

In the context of knowledge base debugging, MBD techniques are applied when a knowledge base does not fulfill some basic requirements, e.g., when it is inconsistent in itself, or when test cases indicate a failure, i.e., an unexpected output. In the usual MBD problem formulation, test cases are logical sentences that must (or must not) be entailed under the assumption of a correctly formulated knowledge base. The output of an MBD tool is a collection of diagnoses, where each *diagnosis* corresponds to a set of assumedly faulty parts of the knowledge base. Users of the debugger, such as experts or knowledge engineers, can then investigate one diagnosis after the other and inspect the involved components to see if they are faulty or not.

Unfortunately, the number of diagnoses can in some cases be large, e.g., because the information provided by the test cases is insufficient and does not allow the debugger to isolate the true cause of the observed failure. In such cases, already early works suggested asking an expert to provide additional information to narrow down the set of possible fault locations [21]. For example, in the traditional application domain of MBD techniques—electronic circuits—users of a diagnosis system were asked to make additional measurements that give some indication of the health state of certain components. In more recent years, different algorithms for *sequential* (or: interactive) diagnosis for knowledge-based systems were proposed [12, 22, 23, 24, 25, 26, 27]. Debuggers of this type interactively ask their users to provide feedback about the correctness of parts of the knowledge base or certain inferences. One concrete implementation of such a debugger is *OntoDebug* [28, 29], a plug-in for the Protégé ontology editor. Compared to approaches that solely rely on test cases, the main advantage of such *query-based* techniques is that they can interactively guide their users to the true cause of the observed problem. In addition, if the users always provide correct answers to the debugger's questions, then query-based diagnosis techniques can guarantee the identification of the true fault location.

The evaluation of sequential diagnosis techniques is usually based on simulations designed to measure, for instance, the time needed to derive the best next query to the expert or the total number of required queries to isolate a fault. Such measures however can have certain limitations when it comes to assess the true usefulness of a debugging approach. In the domain of software engineering the practical relevance of results obtained with the help of simulation-based evaluations of debugging tools was previously questioned by Parnin and Orso in [30]. In recent years, a number of user studies were therefore conducted by researchers that aim to assess the true usefulness of different academic approaches to tool-supported testing and debugging in the context of software engineering [31, 32, 33].

With the present paper we continue this line of recent research. Specifically, our goal is to determine the usefulness of query-based approaches for knowledge base debugging. We correspondingly conducted *laboratory* studies in the form of testing and debugging exercises, where participants either applied an approach based on test cases or were supported by a query-based debugger, concretely, by an earlier version of the OntoDebug tool mentioned above. Our research questions are related to *(i)* the efficiency and effectiveness of query-based debugging (i.e., do experts need less time, do they find more faults?), *(ii)* the ability of users to find out which of the returned diagnoses is the correct one, and *(iii)* the complexity of answering system-generated queries for experts.

Among other aspects, our results show that a query-based approach is indeed helpful to make the debugging process more efficient, without leading to a loss in effectiveness. Furthermore, as our experience and other studies show, experts sometimes provide wrong answers to the questions of a debugger ("oracle errors"). We therefore conducted additional pen-and-paper studies to develop and validate a prediction model that can be used to estimate the probability of such oracle errors based on the complexity of the query or test case.

The paper is organized as follows. After discussing previous works in Section 2, we provide the technical background on MBD-based knowledge base debugging in Section 3. Section 4 presents the detailed research questions of our work and Section 5 and Section 6 discuss the outcomes of our main studies. In Section

7, we finally present first results regarding our prediction model for oracle errors. The paper ends with a discussion of research limitations and a summary of our contributions.

2. Related Work

The process of creating and maintaining a KB is prone to error and—like in standard software development projects—experts can make mistakes when they encode the knowledge about a problem domain. Correspondingly, a number of techniques and tools for KB testing and debugging were proposed over the years. In the following, we first briefly review the main debugging strategies suggested in the literature and then specifically discuss previous works that aimed at evaluating the utility of the corresponding *tools* with the help of user studies.

2.1. General Knowledge Base Debugging Approaches

We can mainly distinguish between *model-based* and *heuristic* approaches for KB debugging. Among the *model-based* approaches, those based on the general MBD principles proposed in [13] are probably the most popular ones. They were, for example, used to debug ontologies [16, 34, 35], constraints [14, 36], or Answer Set Programming encodings [37, 38].

In case of ontology debugging, MBD methods are used to find sets of axioms, called diagnoses (or: candidates/repairs), that must be modified by a developer in order to formulate the intended ontology. From the technical perspective these methods can roughly be classified in glass-box and black-box ones [39, 40]. Glass-box approaches [41, 42, 43, 44, 45, 46, 47] modify the reasoner such that a single run outputs justifications or diagnoses directly. Black-box methods, in contrast, usually apply various search techniques [16, 22, 40, 48] with calls to highly-optimized reasoners for consistency checking and/or the computation of justifications (conflicts) [35, 36, 49, 50].

In practical settings, given an inconsistent/incoherent ontology an MBD approach might return more than one diagnosis (sometimes referred to as *candidates*). In order to restrict the number of obtained diagnoses to only relevant ones Friedrich et al. [16] suggested the notion of test cases, which were later also used in, e.g., [51, 52, 53]. Each test case is defined as set of axioms that *must* or *must not* be entailed by the intended ontology. Given a set of test cases a debugger can use them to focus only on those diagnoses such that if all axioms of a diagnosis are changed the resulting ontology will satisfy all test cases. However, in many situations, it can be unclear to a developer which test cases should be formulated before the diagnosis session such that a debugger will be able to find the real cause of an unexpected output. In this case, query-based approaches [21, 25, 27, 54] help the user to automatically create test cases. Specifically, the task of the users is reduced to answering a sequence of queries on whether or not the intended ontology must entail a given set of axioms. Given the answers of the developer a sequential debugger can determine the *true diagnosis* within the candidates, i.e., the one diagnosis that pinpoints the actually faulty parts of the knowledge base.

Depending on the complexity of the underlying problem, model-based methods can be comparably costly in terms of computation time and space. However, one main advantage of MBD approaches is that any diagnosis that is returned is a precise and succinct explanation of all identified problems.

In contrast, *heuristic approaches* to KB debugging, such as [55, 56], are usually based on handcrafted syntactic pattern matching procedures, see, e.g., [8, 57]. Their main advantage is that they allow for the fast fault localization in case model-based approaches are too slow. Typically, the debugging procedures are specifically designed to find (combinations of) syntax constructs in a KB that are highly likely to be faulty. Examples of such constructs are, among others, the application of universal role restrictions or disjointness constraints in related ontology axioms [9]. However, the computational efficiency of these methods comes at a price. On the one hand, they can only identify bugs for which appropriate heuristics were defined. On the other hand, they might point to alleged bugs that turn out to be correct descriptions. Consequently, the reliability of the returned results in terms of precision and recall can be low.

In this paper, we focus on the MBD approach presented in [20, 27], since it *(i)* provides guarantees about the completeness and soundness of the debugging algorithms, and *(ii)* allows for a precise fault localization by querying its users for additional information.

2.2. Usability Analysis of Tools

Since KBs in practice can be large and complex, the research community developed a number of Integrated Development Environments (IDEs) for KB creation and maintenance. Examples of such environments are the MiniZinc IDE for constraint modeling [58], Protégé, which supports the creation of ontologies [59], ASPIDE as a tool for the development of Answer Set Programs [60], as well as various Prolog IDEs like SWI-Prolog [61]. Several of these IDEs come with some embedded debugging support or can be extended with external tools like the OntoDebug plug-in used in this paper [62, 63, 28].

Two main approaches exist in the literature to evaluate the usability of KB debugging tools. One way is to do *computational analyses* providing us insights about the usability of the tools indirectly. The second form is based on *user studies*, where the performance and behavior of experts while using the debugger is observed and analyzed. The majority of research works in the field are based on the first form of experiments. In comparison to user studies, conducting computational analyses is usually easier, since the only requirement for such evaluations is the existence of a representative collection of faulty KBs that contain real-world or artificially injected faults. Given such KBs, the performance of different debugging algorithms can be compared, for example, in terms of their time and space requirements, the number of calls to the reasoner, the theoretical number of required user interactions, or the precision of the fault localization process. The obtained results can then be used to *indirectly* assess potential usability enhancements of considered debugging approaches. For instance, we can assume that the reduction of the required computation time improves the system's usability, e.g., because the developer gets faster feedback and can find more bugs in shorter time.

However, computational analyses cannot be used to determine if the assumptions of the considered debugging methods actually hold. For instance, the interactive ontology debugging method suggested in [27] assumes that a user can decide with certainty if an arbitrary axiom must be entailed by the intended ontology or not. If this assumption does not hold, i.e., the user cannot (correctly) answer all queries of the debugger, the fault localization process might not lead to a unique (correct) result.

User studies can help us to verify such assumptions and can give us additional insights regarding the acceptance and usefulness of a debugging tool. In the literature, only a very few examples of such user studies exist.

For example, the model-based ontology debugging approach proposed in [34] and implemented in the Swoop editor [64] was evaluated by twelve undergraduate and graduate students [42]. The authors' goal was to investigate if providing *justifications of bugs*, i.e., irreducible faulty subsets of an ontology, can help users find and repair bugs more efficiently. Every subject that participated in the study had at least nine months of experience in ontology engineering and went through an additional 30-minute training session on ontology debugging. The results of the study indicate that tool support in the form of justifications during the debugging process is essential for successful fault localization. However, given the small number of participants, the authors were not able to validate that their results are statistically significant.

Another user study reported in [65] questioned if the justifications generated by model-based ontology debuggers can actually be understood by the users. Experiments were conducted with 14 undergraduate students and showed that justifications can be separated into easy and hard ones. Unfortunately, also in this case the small number of participants did not allow the authors to obtain sufficient statistical evidence to understand why the users find some explanations hard or easy.

Finally, a collection of heuristic approaches [8, 9, 66] was studied in [67] and compared with an MBD approach [35]. All 14 subjects participating in the study were educated software engineers, had some experience with ontologies, but no knowledge about hydrology, which was the domain of the study. The task for the participants was to debug and repair an ontology without understanding exactly what it is about. One group of six participants was supported by the MBD approach; the remaining subjects used a heuristic strategy. The obtained results were not fully conclusive. Both participant groups needed about the same amount of time and no clear preference for either of the approaches was observed. Only for the problem of repairing the ontology, the heuristic patterns helped the subjects to identify bugs more accurately. However, this result must be interpreted with care because the model-based tool did not provide any repair support at that time.

In our work, we continue this line of research which aims to assess the usability of debugging approaches based on user studies. Similarly to previous works, we base the user study on different KBs (ontologies) into which artificial faults were injected. In addition, like in previous research, we involve students in the study, who have a certain level of education in the development and debugging of ontologies and who received some initial training with the tool. In contrast with previous studies, we were able to recruit a larger number of participants, which allows us to apply certain statistical analyses. Moreover, we are focusing not on justifications, which are alternative explanations of *one* fault, but on diagnoses, where each diagnosis provides a potential characterization of *all* faults in an ontology.

3. Background: Knowledge Base Debugging with MBD

In this section, we outline the main principles of applying model-based diagnosis techniques for knowledge base debugging. We use the particular problem of ontology debugging to illustrate the problem. Ontology debugging was also the task in the user studies reported in this paper, where the participants used the *OntoDebug*[2] debugging plug-in [28] of the popular ontology editing tool *Protégé* [68].[3] The underlying principles and algorithms of the debugging approach are, however, not limited to ontologies and can be applied for various forms of knowledge representation and reasoning, see [20, 24, 62, 69].

3.1. Model-based Diagnosis for Ontology Debugging

In the field of computer science, ontologies are the core of semantic systems. Using a language like OWL [70], they formally describe the relevant concepts in a domain as well as their properties and interrelations. Usually the main goal of semantic applications is to use some form of logic-based reasoning to derive additional facts (*entailments*) from the given knowledge base.

The starting point for a debugging session normally is when we observe a discrepancy between what we call the *intended ontology* (denoted as $\mathcal{O}^*$) and what we observe for a current version of an ontology $\mathcal{O}$. In the biology domain, the ontology engineer might, for example, expect that the ontology-based system is able to deduce from the given axioms that men are animals.[4] If, however, it is inferred, e.g., that men and animals are disjoint, the underlying KB is incorrect and the problem is to find one or more faults in the ontological axioms. Generally, the possible discrepancies include the inconsistency of $\mathcal{O}$, the unsatisfiability of its classes or the presence or absence of certain entailments [28].

3.1.1. Example

We use the following example to illustrate how MBD techniques can be applied to ontology debugging. Let our ontology consist of the following *terminological axioms* $\mathcal{T}$: $\{ax_1 : A \sqsubseteq B,\ ax_2 : B \sqsubseteq C,\ ax_3 : C \sqsubseteq D,\ ax_4 : D \sqsubseteq R\}$ They define that A is a subclass of B, B a subclass of C etc. In a specific domain, this could for example mean that a *MathStudent* is a subclass of *Student*, which

[2] http://isbi.aau.at/ontodebug/

[3] https://protege.stanford.edu/

[4] See, e.g., http://owl.man.ac.uk/2003/why/latest/.

is a subclass of *UniversityMember* etc.

Furthermore, the ontology contains two *assertional axioms* $\mathcal{A} = \{A(v), A(w)\}$, which specify that v and w are instances of class A. In the practical application, we could have an assertion like *MathStudent(john)*. Let us assume that the two assertions are known to be correct. In this case, they should not be considered as fault candidates in the debugging process. To this end, the knowledge engineer would add these axioms to the *background theory* $\mathcal{B}$ [20]. That is, the ontology would be partitioned into a possibly fault part $\mathcal{O}$ and a correct part $\mathcal{B}$. In our case, $\mathcal{O} := \mathcal{T}$ and $\mathcal{B} := \mathcal{A}$.

To make sure that the ontology is correct, the user can specify a set of *positive test cases* P and a set of *negative test cases* N, where each test case is an axiom. In our case, let $P = \{B(v)\}$ and $N = \{R(w)\}$, which means that v should be inferred to be of class B and w must *not* be inferred to be of class R. So, P comprises required entailments whereas N contains required non-entailments.

Unfortunately, our ontology $\mathcal{O}$, together with the correct axioms $\mathcal{B}$, entails $R(w)$, since $A(w)$ holds and A transitively is a subclass of R. Formally, $\mathcal{O} \cup \mathcal{B} \models R(w)$. Given the positive and negative test cases, the ontology $\mathcal{O}$ as well as the background theory $\mathcal{B}$, one can then derive that it is not possible that all axioms of $\mathcal{O}$ are correct at the same time, i.e., at least one of them must be faulty. In technical terms, we have determined a minimal *conflict set* [13] $CS = \{ax_1, ax_2, ax_3, ax_4\}$ (which is also the only conflict set in this example). These computations can be accomplished, e.g., by means of a *conflict detection* algorithm [36].

As a consequence, the engineer has to modify the ontology in a way that the negative test cases are not entailed anymore, while at the same time the positive test cases can still be satisfied. Given the conflict set CS, according to [13], the possible minimal repair strategies (called *diagnoses*) in our case are

$\mathcal{D}_1 : [ax_1] \quad \mathcal{D}_2 : [ax_2] \quad \mathcal{D}_3 : [ax_3] \quad \mathcal{D}_4 : [ax_4]$

Intuitively, in the example, the removal of any individual axiom in $\mathcal{O}$ would break the subclass relationship chain, and the undesired entailment would not be present any more. However, based on the positive and negative test cases alone, an MBD algorithm cannot discriminate between the four diagnoses and we cannot derive the *true* cause of the problem. The user can therefore either inspect all axioms manually, or provide more information, e.g., in terms of additional test cases.

In our case, let us assume that the user specifies an additional negative test case $B(w)$, i.e., it must not be entailed that w is of class B. With $N = \{R(w), B(w)\}$ and $P = \{B(v)\}$, a model-based debugging algorithm will return $\mathcal{D}_1$ as the only minimal diagnosis. Specifically, removing ax_1 from $\mathcal{O}$ will lead to the effect that none of the unwanted entailments in N will be observed anymore.

Unfortunately, the modified ontology $\mathcal{O}_1 := \mathcal{O} \setminus \mathcal{D}_1$ does not entail the positive test case anymore. Therefore, $\mathcal{O}_1$ must be somehow extended to lead to the required entailments, in our case to the positive test case $B(v)$. Since the debugger cannot know how to correctly extend the knowledge base, one strategy is to use the required entailments P explicitly as an extension [20]. Hence, in our example one would simply add $B(v)$ to $\mathcal{O}_1$.

3.1.2. Formal Characterization: Diagnosis Problem

More formally, an instance of a diagnosis problem can be characterized as follows [14, 28].

Definition 1 (Diagnosis Problem Instance (DPI))**.** *Let $\mathcal{O}$ be an ontology (including possibly faulty axioms) and $\mathcal{B}$ be a background theory (including correct axioms) where $\mathcal{O} \cap \mathcal{B} = \emptyset$, and let $\mathcal{O}^*$ denote the (unknown) intended ontology. Moreover, let P and N be sets of axioms where each $p \in P$* must *and each $n \in N$* must not *be entailed by $\mathcal{O}^* \cup \mathcal{B}$, respectively. Then, the tuple $\langle \mathcal{O}, \mathcal{B}, P, N \rangle$ is called a* diagnosis problem instance (DPI)*.*

A diagnosis is then a set of axioms that are removed from the ontology, with the particular requirement that the resulting ontology, together with the background knowledge and the positive test cases, is *(a)* consistent and *(b)* does not entail the negative test cases.

Definition 2 (Diagnosis)**.** *Let* $\langle \mathcal{O}, \mathcal{B}, P, N \rangle$ *be a DPI. Then, a set of axioms* $\mathcal{D} \subseteq \mathcal{O}$ *is a* diagnosis *iff both of the following conditions hold:*

1. $(\mathcal{O} \setminus \mathcal{D}) \cup P \cup \mathcal{B}$ *is consistent (coherent, if required)*[5]
2. $(\mathcal{O} \setminus \mathcal{D}) \cup P \cup \mathcal{B} \not\models n$ *for all* $n \in N$

A diagnosis $\mathcal{D}$ *is* minimal *iff there is no* $\mathcal{D}' \subset \mathcal{D}$ *such that* $\mathcal{D}'$ *is a diagnosis.*

Different diagnosis computation algorithms exist; they can be distinguished based on whether they generate diagnoses *indirectly*, i.e., via the computation of conflict sets, or *directly*, e.g., via divide-and-conquer techniques or through the prior compilation of the problem to an alternative target representation like SAT [13, 20, 22, 40, 72, 73, 74, 75, 76].

3.2. Sequential Diagnosis

As the example shows, additional knowledge (in our case, test cases) can help to further focus the debugging process and rule out possible fault candidates. Not all test cases are, however, equally helpful. One of the goals of *sequential diagnosis* is therefore to automatically identify "good" or optimal test cases, and to interactively ask the user to specify for each such test case, whether the comprised axioms are entailments or non-entailments of the intended ontology. Based on the answer, the debugger can then update its knowledge and repeat the process until only one single diagnosis remains.

3.2.1. Example

One way to assess the utility of different possible test cases—which at the end correspond to queries to the user—is to analyze the entailments of the ontologies $\mathcal{O}_i^* := (\mathcal{O} \setminus \mathcal{D}_i) \cup P$ after the application of the different diagnoses $\mathcal{D}_i$.

[5] An ontology $\mathcal{O}$ is *coherent* iff there do not exist any unsatisfiable classes in $\mathcal{O}$. A class X is *unsatisfiable* in an ontology $\mathcal{O}$ iff, for each interpretation $\mathcal{I}$ of $\mathcal{O}$ where $\mathcal{I} \models \mathcal{O}$, it holds that $X^{\mathcal{I}} = \emptyset$. See also [71, Def. 1 and 2]

In our example from above, the four ontologies $\mathcal{O}_1^*, \ldots, \mathcal{O}_4^*$ have, among others, the following entailments:

$$\mathcal{O}_1^* : \emptyset \quad \mathcal{O}_2^* : \{B(w)\} \quad \mathcal{O}_3^* : \{B(w), C(w)\} \quad \mathcal{O}_4^* : \{B(w), C(w), D(w)\}$$

These entailments can be obtained, e.g., with the help of the realization service [2] of a Description Logic reasoner [77, 78] and can serve us as test cases.

Let us assume that the user knows that $D(w)$ must be entailed and adds it as a positive test case, i.e., the diagnosis problem instance is now

$$DPI = \langle \mathcal{O}, \{A(v), A(w)\}, \{B(v), D(w)\}, \{R(w)\} \rangle$$

Given this additional information, a model-based debugger will return only one single diagnosis, $\mathcal{D}_4 = [ax_4]$. All other diagnoses, that existed for the problem instance without the new test case, are no longer minimal diagnoses. Specifically, applying diagnosis $\mathcal{D}_1$, $\mathcal{D}_2$, or $\mathcal{D}_3$ does not affect axiom ax_4, which is however, due to $D(w) \in P$, responsible for the unwanted entailment $R(w) \in N$.

Sequential diagnosis algorithms usually make analyses of this type to determine queries (test cases) that are likely to narrow down the set of remaining diagnoses. At the end, the user only has to categorize such system-generated queries and acts as an *oracle* for the debugger.

3.2.2. Formal Characterization: Oracle and Queries

The terms oracle and query can be formally described as follows. An oracle categorizes each element of a set of axioms either as a positive or a negative test case, i.e., either an axiom has to be entailed or it must not be entailed.

Definition 3 (Oracle)**.** *Let* $\mathbf{Ax}$ *be a set of axioms. Furthermore, let* $ans : \mathbf{Ax} \to \{P, N\}$ *be a function which assigns axioms in* $\mathbf{Ax}$ *to either the positive or the negative test cases. Then, we call* ans *an* oracle *w.r.t. the intended ontology* $\mathcal{O}^*$*, iff for* $ax \in \mathbf{Ax}$ *both of the following conditions hold:*

$$ans(ax) = P \implies \mathcal{O}^* \cup \mathcal{B} \models ax$$
$$ans(ax) = N \implies \mathcal{O}^* \cup \mathcal{B} \not\models ax$$

Note that the function ans can either be total or partial. In the first case, the oracle (user) is a *full domain expert* and able to classify all queried axioms; in the latter case, this might only be the case for some of the axioms.

Since our goal is to narrow down the set of possible diagnoses, a debugger should propose only queries that guarantee the acquisition of *relevant* information. In other words, each query should eliminate at least one diagnosis, given *any* answer of a full domain expert. Generally, a query consists of one or more axioms and can be characterized as follows.[6]

Definition 4 (Query)**.** *Let $\langle \mathcal{O}, \mathcal{B}, P, N \rangle$ be a DPI, $\mathbf{D}$ be a set of diagnoses for this DPI, and Q be a set of axioms. Moreover, let $Q^P_{ans} := \{q \in Q \mid ans(q) = P\}$ and $Q^N_{ans} := \{q \in Q \mid ans(q) = N\}$ denote the subsets of Q assigned to P and N by an oracle ans.*

Then we call Q a query *for $\mathbf{D}$ iff, for any classification Q^P_{ans}, Q^N_{ans} of the axioms in Q of a full domain expert oracle ans, at least one diagnosis in $\mathbf{D}$ is no longer a diagnosis for the new DPI $\langle \mathcal{O}, \mathcal{B}, P \cup Q^P_{ans}, N \cup Q^N_{ans} \rangle$.*

Different strategies were proposed in the literature to determine "good" or optimal queries, see e.g., [27, 80]. Usually, this is accomplished by computing a set of diagnoses and by analyzing the effects of applying the different diagnoses with respect to a potential query. Complementary to this approach, a recent work suggests novel ways of diagnosis computation to reduce the user's time and effort for query answering [72].

In general, a byproduct of the process of determining the queries is a *quality* estimate for each resulting query. Such a quality measure can, for example, be based on the expected information gain after the user has answered the query

[6]Whenever we speak of a "query" throughout this work, we mean a query in terms of Definition 4, which must not be confused, e.g., with the concept of a query in terms of a query language such as OWL-QL [79]. In our scenario, queries are answered *based on the knowledge of an oracle about the intended ontology*, with the aim to locate faults in an ontology. Queries in terms of query languages are answered *based on the knowledge specified in an ontology, knowledge graph, etc.* in order to find answers to questions of relevance.

[21], on reinforcement learning [81], or on criteria [54, 82] adopted from the field of active learning [83]. Finally, since the generation of queries requires potentially costly calls to an underlying reasoner, approaches exist that aim to minimize the number of these computations [20, 24, 26, 54].

3.3. The OntoDebug Plug-In to Protégé

The described concepts for sequential and test case based MBD for ontologies were implemented in the OntoDebug plug-in of the widely-used Protégé ontology editor. There are two main situations when the user of the tool—possibly after some maintenance activities—might initiate a debugging session with the OntoDebug plug-in. First, the built-in reasoner of Protégé might detect that the given ontology is faulty, e.g. inconsistent or incoherent, in itself.[7]

Second, even if the ontology in itself is consistent and coherent, the user might want to ensure that the implemented ontology does correspond to the intended one by specifying one or more test cases. In case the test cases lead to the disclosure of unexpected entailments, an inconsistency or an incoherency, it is obvious that there is something wrong with the ontology.

One possible first step for the user when starting the debugging process with OntoDebug—independent of how the user detected that there is a problem—is to tell the system which parts of the ontology are definitely correct (and thus part of the background knowledge). This task can be accomplished using the functionality at the right-most side of the user interface of OntoDebug shown in Figure 1. In the example shown in the figure, the user works on problems of the "Koala" ontology of the Protégé project, an ontology that was created for educational purposes and contains typical problems that can occur during ontology development. Specifically, in the example, the user has declared among other things that the axiom "*BA* (bachelor of arts) is of type *Degree*" is definitely

[7] In contrast to other application areas of model-based diagnosis techniques—such as fault localization in electronic circuits [13, 21]—inconsistencies can be present in the context of ontology debugging problems without any initially given test cases (observations).

correct.

Once this optional step is done, the user can start the model-based debugging process. The tool, as mentioned above, then supports two general strategies.

- First, the user can inspect the list of diagnoses returned by OntoDebug to locate the fault and add additional test cases in case the list of diagnoses contains too many elements. Generally, the idea is that the provision of additional, carefully designed test cases, will help to narrow down the set of possible diagnoses, i.e., the possible causes for the problems in the ontology. In the example shown in Figure 1, the user has specified one positive test case ("*Student* is a subclass of *Person*") and a negative one ("*Person* is a subclass of *Marsupials*"), using the sub-window in the middle of the screen.

- The second supported debugging strategy is the query-based one. In this case, the tool will—based on the inconsistent (incoherent) ontology or the failing test cases—compute the first query to the user. In our example, the system determined a query consisting of two axioms shown in the top-left sub-window of the user interface. The two axioms to be categorized by the user are "*KoalaWithPhD* is a subclass of *Koala*" and "*KoalaWithPhD* is a subclass of *Person*." The user can answer the query by using the green and red plus and minus symbols (or leave some axioms uncategorized), and then submit the answer to the system. The system adds the user's feedback to the "Acquired Test Cases" and then restarts the computations using the additionally provided information. In case the information was sufficient to identify one single diagnosis as the cause of the problem, the user is pointed to the faulty parts of the ontology. Otherwise, the system computes a new query to be asked to the user and the cycle repeats until only one diagnosis remains.

Generally, one main difference is that in the test case based approach the users have to think by themselves about good test cases, while in the case of interactive debugging, the responses by the users to the system-generated

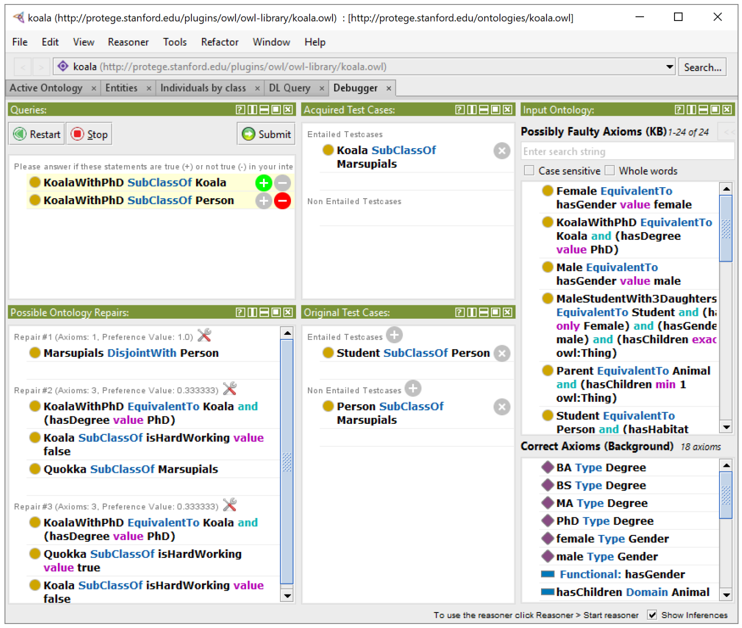

Figure 1: Interactive ontology debugging session with the latest version of OntoDebug.

queries are taken as additional test cases. In this latter case, the selection of the query, and correspondingly the test case, is based on an internal reasoning process that ensures that the most informative queries are chosen.

4. Research Questions

The main promise of interactive, query-based approaches is that they are able to systematically guide users (e.g., knowledge engineers or domain experts) through the debugging process and that after the interactive process the true cause of the observed discrepancies is found. In contrast, there is limited support for users in the more traditional model-based debugging setting, where the users have to provide test cases manually in order to incrementally narrow down the set of fault candidates.

As discussed in Section 2, computational analyses—such as measurements of time or an analysis of the number of required queries—can be insufficient to inform us about the usability and acceptance of the corresponding tools, as such measurements are based on certain assumptions. Such measurements can also not tell us in which ways query-based debugging is advantageous over a test-case based approach.

To address these open questions, we conducted a number of controlled (laboratory) studies, mainly consisting of ontology debugging exercises. We focus on the following main research questions in the context of model-based debugging:

RQ1 Is the debugging process more *effective* when users are supported by a query-based debugging tool than when test cases are the only means to locate faults?

RQ2 Is the process more *efficient* when users are supported by a query-based debugging tool?

RQ3 To what extent do the assumptions of MBD debugging techniques hold?

- RQ3.1 For the case of approaches based on test cases and candidate ranking: Do users have "perfect bug understanding", i.e., do they reliably recognize the true cause of a discrepancy within a list of diagnoses?
- RQ3.2 For the case of the query-based approach: Do users make errors when acting as oracles?

The following main studies were designed and executed.

- In our *preliminary* study (*Study 1*), our goal was to gauge the general usefulness of a test case based debugging approach. We specifically also explored the importance of the ranking of the fault candidates in this experiment (RQ3.1). The study also served us to further improve the design of the main study (*Study 2*).
- In *Study 2*, we investigated the effectiveness and efficiency of the query-based and the test case based debugging approach (RQ1 and RQ2). In that context, we also examined the question of oracle errors (RQ3.2).

Additional pen-and-paper exercises were conducted in the context of both *Study 1* and *Study 2* with the goal of deepening our understanding of the (types of) errors that occur to users while debugging. These insights are then used to devise a heuristic prediction model for such errors (RQ3.2). We discuss *Study 1* in Section 5, *Study 2* in Section 6, and the additional studies in Section 7.

5. *Study 1*: Investigating MBD-debugging With Test Cases

5.1. Design of the Pre-Study

5.1.1. Task

The task of the participants in this study was to *find the faulty axioms* (true diagnosis) in a given faulty ontology *(i)* based on a provided description of the *intended* ontology in natural language *(ii)* using the OntoDebug tool described above *(iii)* by creating test cases manually (the query-based debugging functionality was not available to the users). The participants were explicitly instructed to *(iv)* constantly inspect the list of possible diagnoses throughout the debugging session and to *(v)* mark the true diagnosis once they detected it in the list. After a diagnosis was marked, the debugging session ended. In Figure 1, the list of diagnoses is shown in the bottom-left sub-window labeled with "Possible Ontology Repairs".

5.1.2. Ontologies

In order to make sure that the outcomes regarding the usefulness of the test case based debugging approach do not depend on the specifics of a certain ontology, two different ontologies describing two different domains were used in the study. The first one corresponded to a (simplified) model of the "Klagenfurt University" (*university* domain) and the second one was a real-world knowledge base made available by the "Communal IT Center of Carinthia" (*IT* domain). We prepared the ontologies for the study by injecting five faults into each of them such that the resulting ontologies were inconsistent and incoherent in themselves. That is, for both ontologies the true diagnosis included five faulty axioms (as shown in Table 1). The designed ontologies were similar in their size and complexity. For example, both included about 50 classes, 90 subclass relationships, and 20 object properties. Moreover, both included roughly equally complex logical formalisms and used the full expressivity of the Description Logic $\mathcal{SROIQ}$ (OWL 1.1 [84]) [2, 85].

Table 1: Faulty ontology axioms (university domain) in OWL Manchester Syntax [86].

Nr.	Faulty Axiom
1	`Department SubClassOf offers only Course`
2	`Library SubClassOf offers only Visitation`
3	`Research_Event SubClassOf has_Speaker only (Person and (has_Degree some Degree))`
4	`Assembly_Hall DisjointWith Room`
5	`Department DisjointWith Room`

5.1.3. Participants

We recruited 29 participants for the study. All participants were computer science students of our university and were enrolled in an ongoing master program course on knowledge engineering. During this course, the participants, who already had a background in logics, were introduced to model-based debugging, formal ontologies, Description Logics, and the OWL language. The participants also had first experiences in designing ontologies with Protégé and debugging them with OntoDebug. Overall, the participants were very homogeneous with respect to their knowledge and background.

5.1.4. Independent Variables

We considered two independent variables, the ontology to be debugged (*university* vs. *IT*) and the position (*visible* vs. *not visible*) of the true diagnosis in the list of diagnoses returned by the debugger. Each participant was randomly assigned to one ontology and one setting for the position of the true diagnosis.

Similar to the work in [30], we varied the position to assess the importance of the ranking of the diagnoses returned by the system. Specifically, in the *visible* case, the true diagnosis, which comprised all actually faulty axioms of the ontology, was placed within the top three diagnoses and always visible to the user. In the other case (*not visible*), the true diagnosis was further down the list. Generally, the diagnosis problem was designed in a way that the initial list of diagnoses before further test cases are specified is comparably large, including over 150 diagnoses in each case.

5.1.5. Dependent Variables

We made a variety of automated, *objective* measurements while the participants were executing the task, like the needed *time*, the *number of user interactions* (mouse clicks) in the debugger, and the *number of diagnoses still in the list* of diagnoses when the participants submitted the diagnosis which they thought is the correct one. In the context of *Study 1* the most important automated measurement was on the *correctness* of the debugging process in terms of (*i*) the fraction of correctly identified faulty axioms and (*ii*) the fraction of users who correctly identified all five faulty axioms (i.e. the true diagnosis).

Moreover, the participants had to specify their *subjective* degree of belief (*confidence*) in having solved the fault localization task correctly. For this, they should use a range between 0 (very *uncertain* that the marked diagnosis is the true one) and 100 (*certain* that the marked diagnosis *is* the true one).

5.2. Experiment Execution

The study was conducted in one of the computer labs of our university. The required software was pre-installed on the lab computers. All of the computers were identically equipped. After being informed about the tasks of the study and after the participants had declared their consent, they were provided with detailed material on paper. The handout essentially included a description of the domain that was incorrectly modeled by the ontology the participants had to debug. Thus, the paper characterized the *intended ontology* as discussed in Section 3.

The description was given as a natural language text, with important concepts highlighted. In particular, class and property names in the ontology were *italicized* and underlined, respectively. An example of such a description from the university domain is the following:

> From an organizational point of view, the University is subdivided into several *OrganizationalUnits*. Each *OrganizationalUnit* employs some *OfficeEmployee*(s) and some *Teacher*(s), has some *Room*(s) which is/are (an) *Office*(s), is directed by exactly one *Director* and is located in some *Building*. Two special types of *OrganizationalUnits* are the *Directorate* and the *HumanResourcesUnit*.

Before the participants started their task, they received another brief tutorial on how to debug an ontology with the OntoDebug tool. They used the "Koala" ontology that is available in Protégé (cf. Sec. 3.3) for that purpose. During the experiment, the participants were not allowed to talk to each other. The participants were supervised by three instructors, who were present to answer questions in case of problems with the software.

5.3. Outcomes of Study 1

The measurements obtained in *Study 1* are summarized by Figures 2 and 3. As mentioned above, the main question of this pre-study was (i) to gauge the general usefulness of MBD-debugging with test cases and (ii) to assess the importance of the ranking of the diagnoses. Furthermore, a side goal was to obtain experiences regarding the study design for the main study (*Study 2*).

5.3.1. General Usefulness of Model-based Debugging

On average, the participants took about 28 minutes and 81 mouse clicks for the task before they submitted their solution.[8] Overall, the participants correctly identified as many as 77 % of the problematic axioms,[9] i.e., almost

[8]The standard deviation was 12.6 minutes (time) and 35 mouse clicks, respectively.
[9]The standard deviation was 21 %.

four out of the five injected faults shown in Table 1 were eliminated. From the 29 participants, 10 (34.4 %) correctly identified the true diagnosis, i.e., all five faulty axioms (cf. Figure 2).

Overall, we find this result very positive, given the complexity of the task. The study clearly indicates that model-based debugging is actually helpful for knowledge engineers. Since we did not observe any statistically significant differences between the observations that were made for two different ontology debugging problems (*university* and *IT*), we are confident that the usefulness of the approach is not limited to just one domain.

There were various reasons why some participants did not successfully find all faults. A main issue appeared to be a certain lack of attentiveness and precision when reading the natural language specification of the intended ontology. Based on these observations, we revised some of the specifications, e.g., by removing possible ambiguities, when designing *Study 2*. To some extent, it also seemed that some participants did not properly understand the semantics of certain elements of the knowledge representation language.

5.3.2. Importance of Ranking of Candidates (RQ3.1)

In the context of RQ3.1, our goal was to investigate if the capability of a debugger to rank the true diagnosis higher in a list of candidates directly translates into a more effective debugging process. Table 2 shows in how many cases the true diagnosis—which comprises all five injected faults—was found, depending on whether it was among the top-ranked (visible) candidates or not.

Table 2: Relationship between full correctness of the debugging task and visibility of the true diagnosis in the list of diagnoses presented to the participant.

		true diagnosis visible	
		yes	no
true diagnosis found	yes	5	5
	no	9	10

Interestingly, the observations shown in Table 2 do not provide evidence that

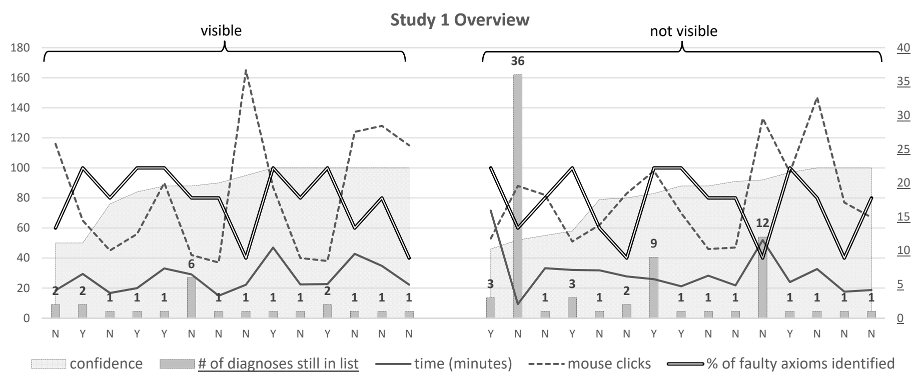

Figure 2: Overview of the outcomes of *Study 1*. The figure shows the measurements for the dependent variables for all 29 debugging sessions, grouped by the position ("visible" left, "not visible" right) of the true diagnosis in the diagnoses list, and sorted from low to high confidence. The labels along the x-axis indicate whether the true diagnosis was found ("Y") or not ("N") during the respective session. Variables plotted w.r.t. the right y-axis are underlined.

the users were more effective when the true diagnosis was always visible.[10] Such a non-effect of varying the position of the fault in a ranked list was also reported in [30].

Moreover, 10 of the 14 participants of the group where the true diagnosis was ranked highly continued specifying test cases until only one diagnosis was left in the list (cf. Figure 2)—even though all participants were explicitly instructed to constantly inspect the list of diagnoses and mark the true diagnosis once they detected it in the list. A large number of participants therefore did not recognize the actual fault even though it was shown to them.

These findings challenge the assumption of a "perfect bug understanding" of the users, i.e., they do *not* always immediately identify a fault when they are pointed to it. In other words, even if the true diagnosis was visible to the participants, they *(i)* did not recognize it in the majority of the cases and *(ii)* did

[10] This is supported by Fisher's Exact Test [87] (p-value = 1.00).

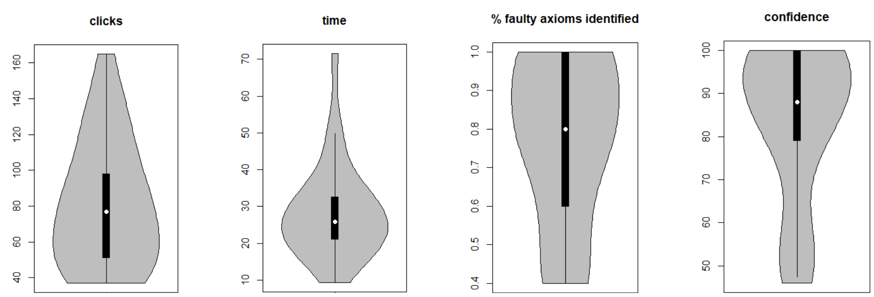

Figure 3: Vioplots showing the distribution of the dependent variables in *Study 1*.

not identify it more often than other participants to which the true diagnosis was not (always) visible. As a result, fault ranking metrics should therefore not be considered as the only measure when different algorithmic debugging strategies are compared [30].

5.3.3. Additional Observations (Study 1)

Positive test cases are more reliable: From the 244 test cases provided by the participants (8 on average per debugging session), the majority (71 %) were *positively* formulated, i.e., they described required entailments. The participants therefore seemed to feel more comfortable specifying things that must be entailed than those that must not. An analysis of the fault rates for positive and negative test cases indeed confirmed that negative ones, i.e., formulated non-entailments, were significantly[11] more often faulty (24 % vs. 10 %, see Table 3). This result suggests that it is better to ask users questions with a bias towards the positive answer[12] in query-based KB debugging, in order to minimize the occurrence of oracle errors.

Users can be overconfident: The participants of the study were partially overconfident (cf. Figure 2). The average confidence value expressed by the partici-

[11] According to a Chi-Squared Test with $\alpha = 0.01$ (p-value = 0.00496) as well as a (two-tailed) Fisher's Exact Test with $\alpha = 0.01$ (p-value = 0.008).

[12] A "bias towards the positive answer" means that the estimated probability of getting a positive answer to the question is higher than that of a negative answer.

Table 3: Relationship between the type of formulated test case and its faultiness.

		type of test case	
		positive	negative
test case faulty	yes	18	17
	no	156	53

pants regarding the correctness of the identified diagnosis was at about 83 % (cf. Figure 3). While this roughly corresponds to the percentage of the identified faults, only 34 % of the participants have correctly located *all* five faults. Interestingly, the confidence of those participants who did not find the true diagnosis was even slightly higher than the confidence of the successful participants.

Hence, subjective confidence estimates have to be handled with care [81] when they are intended to be used to guide the debugging process [27].

Users consider themselves as imperfect oracles: We found that only 31 % of all users, and an even lower 20 % of the ones that successfully found all faults, were *fully* confident about the correctness of their debugging actions (cf. Figure 2). This teaches us that humans generally do not regard themselves as perfect oracles for knowledge engineering tasks, which questions the frequently made "perfect oracle" assumption. We pick up on this discussion again in Sec. 6.3.4 and Sec. 7.

Completion time and user activity as success predictors (cf. Figure 2): Participants who correctly identified the true diagnosis required on average more time (33 minutes) and specified more test cases (10). However, they needed fewer interactions (71 clicks) than those that submitted a wrong diagnosis (26 minutes, 8 test cases, 87 clicks). This indicates that successful users worked more thoroughly and were more persistent in their testing activity. Unsuccessful ones, in contrast, required more interactions as they more frequently edited, deleted or re-added test cases. An atypically high editing activity can thus be considered as an indicator that a user requires more assistance for the given task.

6. *Study 2*: On the Usefulness of Query-based Debugging

Having established that model-based debugging leads to a good debugging performance, the goal of *Study 2* was to answer our main research questions RQ1 and RQ2 on the efficiency and effectiveness of query-based debugging as opposed to a test case based approach. In other words, do users need less time/effort when supported by a query-based debugger (efficiency) and do they find more faults (effectiveness)?

6.1. Design of the Study

6.1.1. Task

As in the pre-study, the general task of the participants was to find the actually faulty axioms (true diagnosis) in given faulty ontologies *(i)* based on a provided description of the *intended* ontology in natural language *(ii)* using the OntoDebug tool. However, now *(iii)* every participant had to debug two ontologies, one using the query-based and the other using the test case based approach.

6.1.2. Ontologies

Similar ontologies were used as in the pre-study—one describing a *university*, and one describing an *IT* domain, and both again corresponding to the Description Logic $\mathcal{SROIQ}$. Again, we prepared the ontologies for the study by injecting a number of faults into each of them, leading to both inconsistency and incoherency. However, the ontologies were roughly 20 % larger in terms of their size (e.g., number of axioms and classes) than the ones used in *Study 1*; still, the size and complexity of both ontologies was roughly equal. The ontologies were enlarged to achieve a higher number of fault candidates. Concretely, the size of the initial list of diagnoses for both ontologies was now over 1000. This made the diagnosis problems objectively harder than in the pre-study. The reason for this was to compensate for the lower number of participants (23) in *Study 2*, which makes it somewhat harder to achieve statistically significant results. Because, if any effects (e.g., regarding time or user interactions) of employing the query-based debugging are present, then they are likely to be larger for harder debugging problems.

6.1.3. Participants

For *Study 2* we could draw on 23 participants. Again, all of them were attendees of a university master program course on knowledge engineering. However, the focus of the course was now shifted towards Semantic Web technologies to achieve a better preparation of the students for the study. As a consequence, the participants of *Study 2* had a better education on model-based diagnosis, formal ontologies, ontological reasoning, and the used knowledge representation language than those of *Study 1*. Moreover, they had more experience with Protégé and OntoDebug.

6.1.4. Independent Variables

The two independent variables we used were the ontology to be debugged (*university* vs. *IT*) and the debugging strategy (*query-based* vs. *test case based*). We used a within-subjects experiment design in this study, which involves each participant consecutively working on both ontologies and consecutively using both debugging strategies. Thus, we randomly assigned each participant to one of the following configurations:[13]

- Task 1: *university* with *queries*. Task 2: *IT* with *test cases*.
- Task 1: *university* with *test cases*. Task 2: *IT* with *queries*.
- Task 1: *IT* with *queries*. Task 2: *university* with *test cases*.
- Task 1: *IT* with *test cases*. Task 2: *university* with *queries*.

6.1.5. Dependent Variables

In terms of measurements, we recorded the same aspects as in *Study 1* (see Section 5.1.5), i.e., *time*, *number of user interactions*, *number of diagnoses still*

[13] Note, the random variation of the order of the tasks is important to avoid systematic learning effects.

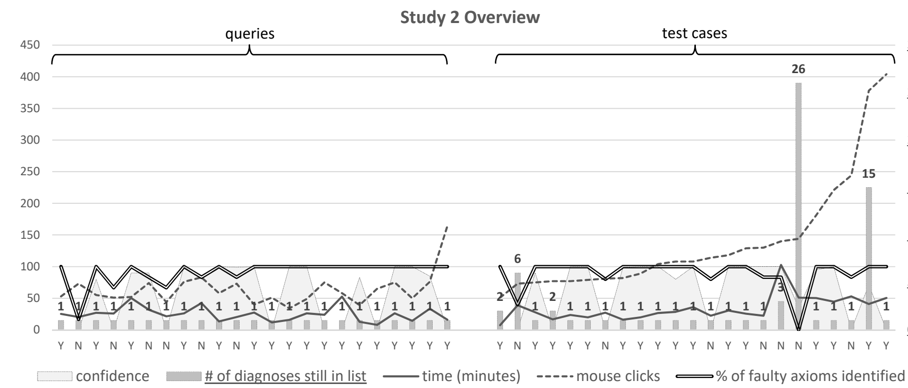

Figure 4: Overview of the outcomes of *Study 2*. The figure shows the measurements for the dependent variables for all 46 debugging sessions (2 per user), grouped by the used debugging strategy (“queries” left, “test cases” right; i-th x-axis entry starting from the left in the “queries” block refers to the same user as i-th x-axis entry starting from the left in the “test cases” block). Records are sorted by the number of mouse clicks of the “test cases” sessions from low to high. The labels along the x-axis indicate whether the true diagnosis was found (“Y”) or not (“N”) during the respective session. Variables plotted w.r.t. the right y-axis are underlined.

in the list, *correctness* (fraction of faulty axioms found, fraction of users finding true diagnosis), and *confidence*.

6.2. Experiment Execution

The experiment execution was exactly the same as in *Study 1*, see Section 5.2.

6.3. Outcomes of Study 2

6.3.1. Effectiveness of Query-based Debugging (RQ1)

To assess the effectiveness of the two debugging strategies, we analyzed how many of the faulty axioms were successfully identified by the participants. Across both ontologies, the participants on average found 91.3 % of the faults when they were supported by the query-based debugger and 89.1 % when the debugging process was based on test cases (as in *Study 1*).[14] Figures 4 and 5

[14]The standard deviation amounts to 19 % (queries) and 23 % (test cases).

show the (distribution of the) achieved success rates for both debugging techniques. The differences were not statistically significant. We therefore conclude that in this experiment, the query-based approach did *not* further increase the effectiveness of the debugging process.

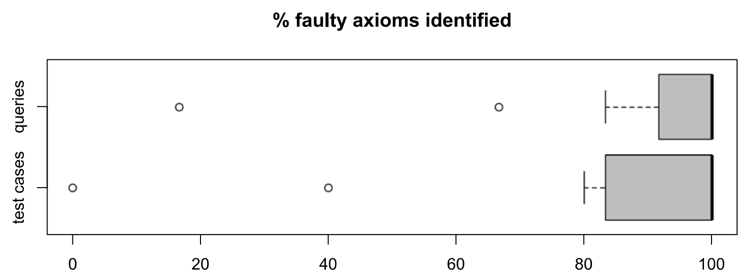

Figure 5: Boxplots showing the distribution of the % of identified faulty axioms per debugging session in *Study 2* for the query-based vs. the test case based approach.

Note, however, that in both cases the success rate was higher than in *Study 1*, where about 77 % of the faults were identified by the participants. We attribute this to the fact that—based on the learnings from *Study 1*—we were more successful in motivating the participants to work more carefully. In addition, the participants of *Study 2* were, as mentioned, better trained in ontology engineering than those of *Study 1*. As a result, it became difficult to greatly increase the already high success rate (89.1 %) obtained by participants who relied on test case based debugging.

Like in *Study 1*, we also looked at how many of the participants could correctly identify *all* faulty axioms (i.e. the true diagnosis) in each ontology. We again found no statistically significant difference between the two debugging approaches (cf. Table 4). Generally, across the ontologies, the fraction of fully successful trials was much higher than in *Study 1*. About 72 % of the participants were able to find all problems in the respective ontologies. Interestingly, we found differences for the two ontologies this time. Over 85 % of the participants were able to find all faults in the *university* ontology, with no significant differences with respect to the debugging method. However, in the *IT* domain, only 57 % were fully successful. A potential reason for this result could lie in the prior knowledge of the participants with regard to the two domains. More research is however required to better understand this phenomenon.

Table 4: Relationship between the used debugging approach and the success in finding the true diagnosis.

		debugging approach	
		queries	test cases
true diagnosis found	yes	17	16
	no	6	7

6.3.2. *Efficiency of Query-based Debugging (RQ2)*

To assess if the query-based debugging technique helps users to accomplish the debugging task faster and with less effort, we compared both the overall time needed by the participants and the number of required user interactions (mouse clicks) in the debugging tool across the two debugging strategies. The (distribution of the) time and user interaction measurements throughout *Study 2* is summarized by Figures 4, 6 and 7.

Participants who were supported by the query-based debugging tool on average needed 24.9 minutes. When using test cases without query support, the average time was 34.0 minutes[15], which amounts to an overhead of 37 %. Looking at the number of required user interactions, the differences are even stronger. With the query-based debugging tool, the number of mouse clicks was more than halved and reduced from about 139 to 64 clicks on average.[16] The differences regarding both time and interactions were statistically significant according to a Wilcoxon Rank-Sum Test;[17] in the case of time to the level

[15]The standard deviation comes to 11 minutes (queries) and 19 minutes (test cases).

[16]Standard deviation: 25 clicks (queries) and 90 clicks (test cases).

[17]Since literature is not always consistent when referring to Wilcoxon's test(s), note that we stick to the description of the test(s) given in [87]. Further note that Wilcoxon's Rank-Sum test compares *independent* samples whereas Wilcoxon's Signed Rank test compares *paired* data.

$\alpha = 0.05$ (p-value $= 0.0418$), and for clicks to the level $\alpha = 0.00001$ (p-value < 0.00001).[18]

Overall, we conclude from the experiments that query-based debugging support is beneficial in terms of the efficiency of the debugging process.

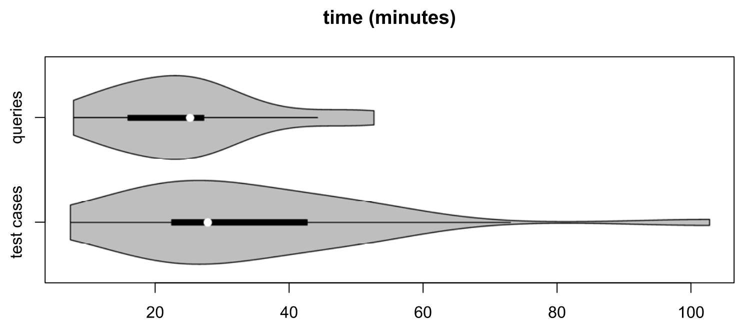

Figure 6: Vioplots showing the distribution of the debugging task completion times in *Study 2* for the query-based vs. the test case based approach.

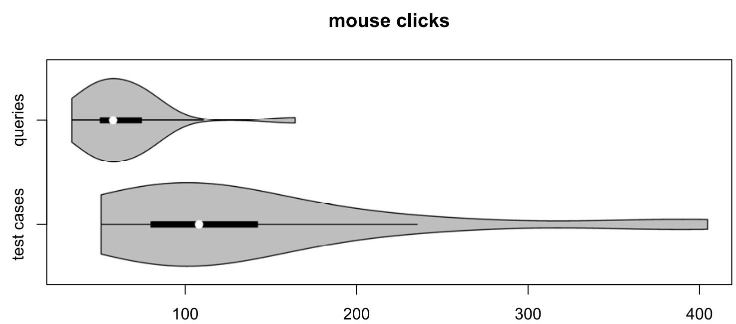

Figure 7: Vioplots showing the distribution of the number of user interactions to complete the debugging task in *Study 2* for the query-based vs. the test case based approach.

[18] Also, when viewing the data as paired (each participant did use both queries and test cases, but each for a different ontology), the results in both cases are highly significant (for $\alpha = 0.05$ and $\alpha = 0.0001$, respectively).

6.3.3. Additional Observations (Study 2)

Users feel equally confident using both debugging approaches: While again overconfident in general (cf. Section 5.3.3), the participants were approximately equally confident about having made no mistakes in the debugging process at all, both when using queries and test cases. Specifically, the average confidence in case of query assistance was 93 % and 92 % when using test cases.[19]

Intuitive focus on mere query answering: Interestingly, without giving the participants who used the query-based debugger any instructions to do so, all of them continued answering queries until a single diagnosis was left (cf. Figure 4). Apparently, they therefore did not rely on the list of diagnoses when using the query-based approach. When relying on test case based debugging, in contrast, more than one quarter of the users selected their solution from a list of more than one diagnosis. In other words, at a certain point they stopped specifying further test cases and considered it more efficient to inspect the candidate list. We interpret this as a sign that test case based debugging was more tiring, and thus more demanding for the users than query answering.

Query answering is more efficient than test case specification: As both the query-based and the test case based approach result in the addition of a new test case per iteration[20], we compared the time users needed per answered query and per specified test case, respectively. The result is very clear (cf. Figure 8). The average test case specification time (≈2:20 min) was almost 60 % (and statistically significantly[21]) higher than the average query answering time (≈1:30 min).[22] This shows that it is more efficient to classify pre-selected axioms as (non-)entailments than to think about specific axioms *and* classifying them. Overall, this result demonstrates the potential of query-based sequential diag-

[19]Standard deviation: 8 % (queries) and 17 % (test cases).

[20]In the query-based scenario the test case is selected by the debugger and classified (as positive or negative) by the user, whereas in the test case based scenario the test case itself and its classification is chosen by the user.

[21]According to a Wilcoxon Rank-Sum Test with $\alpha = 0.001$ (p-value $= 8.96 * 10^{-13}$).

[22]Standard deviation: ≈1:30 (queries) and ≈2:50 (test cases).

nosis approaches to reduce debugging efforts.

Query optimization pays off: The average number of queries (11.6) that had to be answered until the true diagnosis was found by the users was lower than the average number of test cases (13.1) the users specified to isolate the true diagnosis.[23] This shows that automatic (and optimized[24]) test case selection tends to be more efficient than manual test case specification. In other words, the automated approach is better than users in selecting test cases that discriminate (well) between the candidates.

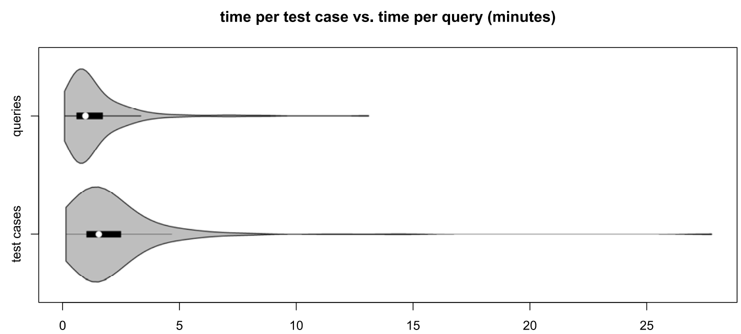

Figure 8: Vioplots showing the distribution of the time participants required to specify a test case vs. the time they required to answer a query in *Study 2*.

6.3.4. Existence of Oracle Errors (RQ3.2)

Both in *Study 1* and *Study 2*, we observed that it is not uncommon that participants make errors when specifying test cases and when answering the system's queries. While in either case the large majority of the inputs provided by the participants was correct, at least one mishap occurred to a considerable fraction of participants in both studies. Even in the main *Study 2*, where the participants were instructed more intensively and where the participants had a better formal education on ontology engineering, about one quarter of the participants made at least one mistake. In the context of the study, mistakes

[23]Stadard deviation: 3.3 (queries) and 4.8 (test cases).

[24]We used entropy-based query optimization as described in [27] in our study.

were made equally for the test case specification and the query answering tasks.

Our observations therefore point to a largely open issue in algorithmic testing and debugging approaches, which are usually based on the assumption that there are no oracle errors. Only a few works exist in the literature, which specifically address the problem of wrong user inputs, e.g., in the context of spreadsheet testing [88], Spectrum-based Fault Localization procedures [89], or general software testing [90].

Next, in Section 7, we will take first steps to address this largely open research question in the context of query-based knowledge base debugging. Specifically, we will describe an initial prediction model that allows us to estimate the probability of oracle errors depending on the complexity of the queries asked to the user.

7. Predicting Oracle Errors based on Query Complexity

When designing a query-based debugging method, different options are available with respect to what types of queries are asked to the users. A closer look at the wrong user inputs in *Study 1* and *Study 2* revealed that from the faulty test case specifications about two thirds had a non-trivial syntactic structure, involving, for example, complex class expressions with intersection, union, or complement operators, as defined in the OWL specification [70]. This supports the intuitive assumption that the syntactic complexity of the required inputs is correlated with the probability of a user error.

The goal of the work described in this section is to develop a first model that allows us to estimate the probability of user error for a given query in a quantitative way. The model can then be used by designers of interactive debugging systems, for example, in order to vary the complexity of the queries depending on the assumed expertise of the user. Alternatively, the model can be used to provide additional hints to the user in case of complex queries.

The proposed model was developed and evaluated with the help of two additional studies, which were performed in the context of *Study 1* and *Study 2*.

The first of these studies, termed *Study E1*, aimed at (i) verifying the conjecture that an axiom's syntactic complexity has indeed a *significant* impact on how well it is understood, and (ii) collecting data as a basis for the design of the prediction model. The second study, termed *Study E2*, was conducted to assess the utility of the model.

7.1. Collecting Data for the Prediction Model (Study E1)

We designed a pen-and-paper study, where the task of the participants—the same ones as in *Study 1*—was to determine the correct translation of axioms written in OWL (Manchester Syntax [86]) into natural language and vice versa. Each participant was provided with ten axioms that were randomly chosen from a larger pool of manually-prepared axioms. The axioms themselves, which again related to the university and IT domain, were designed to have different complexity levels. A simple axiom, for example, would be `X SubClassOf Y`, where `X` and `Y` are class names from the respective domain. More sophisticated axioms involved complex class expressions such as `not(X and Y)` or `p some (X or Y)` which use, e.g., property restrictions and different logical operators. An example of a more complex axiom would be `UndergradStudent SubClassOf not (hasDegree some Degree)`.

For each given axiom, the participants were provided with three possible translations, where only one of them was correct. They then had to assign confidence scores to these answer options that express their degree of belief in the correctness of the respective answer.

To verify our hypothesis that syntactically more complex axioms are more difficult to comprehend, we proceeded as follows. First, we gathered the confidence scores the participants gave to the correct answers for all the translation tasks. Next, we asked two experts to classify the syntax patterns that occurred in the exercises as either particularly hard or particularly easy or neither. We then compared the recorded confidence scores between the group of hard and the group of easy syntax patterns. The average score was 0.55 for the former and 0.95 for the latter group. The statistical significance of this difference was revealed by a Wilcoxon Rank-Sum Test with level $\alpha = 0.01$ (p-value = 0.0015). That is, axioms of higher complexity indeed led to a lower success rate of the translation task. Overall, this finding supports the relevance of a syntax-based prediction model.

To obtain further insights regarding which syntactic features cause difficulties for the users, we manually inspected all answers of the participants. As a result, we identified the following major factors that increase the complexity for the participants: (a) nesting of class expressions, (b) negation in general, (c) negated expressions that are not represented in "negation normal form" (NNF), i.e., which include negated complex class expressions.

7.2. Design of the Prediction Model

Based on the lessons learned from the different studies and on our researcher expertise, we constructed a rule-based prediction model, which takes a query in OWL as an input and returns a score that expresses how likely it is that the query will be properly understood. Viewed differently, the model will tell us the likelihood of an oracle error for the given query.

The idea of the model is to recursively reduce a query to the axioms it consists of, and to then decompose these axioms to the class expressions they comprise. These expressions are in turn successively split into smaller sub-expressions, and so forth, until atomic classes are obtained. Based on the encountered syntactic structure, the model uses respective weights to compute the final query score when the recursion unwinds. The weights are defined based on the observations of our study.

For instance, the model assigns `X SubClassOf Y` a score of 1 (maximum "easiness") because such axioms were always correctly understood by the participants. In contrast, the score for `X SubClassOf not (p some Z)` would be 0.25 due to the involved negation and property restriction. Note that the axiom `X SubClassOf p only (not Z)` that expresses the same fact but is written differently in NNF would be indeed rated as being easier (score 0.29) by the model, which is in accordance with our observations.

To initially validate our model we performed a correlation analysis based on *Study E1*. The analysis revealed that the predictions for the exercises from *Study E1* are well in line with the success rates we had observed in the study (Pearson's $r = 0.53$). For the sake of brevity, we only sketched the main idea of the model here. The exact definition of the model can be found in Appendix A.

7.3. Evaluation of the Prediction Model (Study E2)

Study E2, which involved the participants of *Study 2*, was a pen-and-paper exercise that we conducted to validate the predictive power of our model directly, i.e., through a query answering task. In the study, each participant was provided with a natural language description of a university domain and 25 queries in OWL Manchester Syntax, each consisting of one axiom. The queries were randomly selected from a pool of logical axioms ax_i involving 51 syntactic patterns of different complexities, with scores predicted by our model ranging from 0.05 (hard) to 1 (easy). For each query, the task was to decide if it is true or false in the given domain. The correct answers to all 25 questions were given in the natural language text, i.e., the participants did not have to make any assumptions to correctly answer the queries. The participants were again asked to provide, for each query, on a scale from 0 to 100, (i) a difficulty assessment and (ii) their confidence in the given answer.

From the subjects' questionnaires, we extracted, grouped by syntactic pattern, (a) the percentage of correct answers, (b) the users' average confidence in their answer, and (c) the average subjective difficulty. A comparison of these three response variables with the model predictions yielded high correlation coefficients of 0.36, 0.52, -0.70 for (a), (b) and (c), respectively. Moreover, to assess the statistical significance of the model's predictive power, we ranked all queries according to their score as per our prediction model and performed a median split of the axioms into two groups, one including the easy and one the hard syntactic patterns. An analysis of the response variables (a), (b) and (c) for these two groups revealed that there is a significant between-group difference (Wilcoxon Rank-Sum Test, p-values $< 10^{-5}$, $< 10^{-5}$ and 0.0197) which confirms the predictive power of the proposed model. As a result, axioms that were estimated to be hard according to the model *(i)* in fact led to a higher failure rate, *(ii)* were actually perceived to be harder, and *(iii)* resulted in a lower confidence of the users in their answers. The same relationship holds in the other direction.

As a side note, the prediction model, in case it did not exactly predict the observed success rate, tended to underestimate the success probability. As a consequence, whenever the model predicted that a query is easy (i.e., had a score close to 1), it actually proved to be *very well* understood by the users. Hence, using methods in a query-based debugger that are able to generate "easy questions" with respect to such a prediction model is expected to be beneficial to avoid oracle errors. Examples of such methods can be found in [25, 26, 80].

7.4. Discussion

Overall, our results indicate that our model, although still preliminary, is able to assess the complexity of a given query with good reliability. Clearly, more research is required to further develop the model and to validate it for other problem settings. Nonetheless, we see the results as an important first step in the direction, which can be used when designing an interactive debugging environment.

Furthermore, the model can also be used for other purposes related to debugging, e.g., as an estimator of the prior fault information provided to a debugger. For instance, a higher fault probability could be assigned to axioms in the KB that are rated as hard by the prediction model. As pointed out and empirically proven by several works [81, 27, 82], reliable fault probabilities are a crucial ingredient to efficient fault localization but are often difficult to estimate.

8. Research Limitations

Our research does not come without limitations. First, the number of participants in the different studies, while being larger as in some previous studies on the topic, could be higher and we plan to do additional experiments in the future with a larger set of participants. The participants of our studies were computer science students and all had a comparable background. We argue that this participant group is representative for at least a part of the population of real-world knowledge engineers, i.e., those that have a formal education in computer science.

The experiments conducted in *Study 1* and *Study 2* are based on two specific knowledge bases (ontologies). While we thereby tried to make sure that the insights are not limited to one single domain, our experiments were based on ontologies with a comparable level of complexity. To what extent our insights generalize to much larger knowledge bases, therefore cannot be concluded from the made experiments.

The prediction model presented in Section 7 is still preliminary and must be seen more as a *general* indicator than a fully precise, optimized predictor. In fact, the scores that describe the complexity of an axiom are, for now, estimates that base on one single study and on our own researcher expertise. However, our model evaluation clearly indicates that the rules, i.e., *the way of* using the structure of an axiom for the estimation (e.g., deeper nesting of sub-clauses is harder), are plausible.

9. Summary

Tool support for debugging is not only relevant for traditional software systems, but also for knowledge-based systems. In the field of general software engineering, more and more research works are published which aim at better understanding the true value of such debugging tools for developers. In the field of knowledge-based systems, research on this topic is still limited. With this work, we aim to contribute new insights regarding the usefulness of query-based knowledge base debugging in contrast to a more traditional test case based approach.

We conducted different user studies to address some of the open questions.

The studies showed that users who were supported by any of the two forms of a model-based debugger were able to successfully locate a large fraction—in one study almost all—of the faults in the given knowledge bases. This emphasizes the usefulness of model-based knowledge debugging in general. The query-based approach furthermore proved to be advantageous in terms of the efficiency and, thus, the required user effort of the debugging process. Users not only needed less time and fewer mouse clicks to locate the faults, the internal, optimizing query selection strategy also reduces the number of test cases that are needed to isolate the true cause of the observed problems.

Finally, the studies revealed certain other phenomena of knowledge base debugging processes. One main insight is that measuring the capability of a debugging method to properly rank the fault candidates should not be the only measure to compare different strategies. Another important aspect is that users sometimes provide wrong inputs to the debugging process. Future debuggers should therefore be able to take this aspect into account. In this work, we made a first step in this direction and proposed and evaluated a model that predicts the reliability of the user input for a query of a given complexity. Such predictions can, for example, be used in future systems to decide on which types of queries should be asked to the user in query-based approaches.

Acknowledgements

This work was supported by the Carinthian Science Fund (KWF), contract KWF-3520/26767/38701.

References

[1] S. J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach (Third Edition), Pearson, 2015.

[2] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, 2003.

[3] M. L. Pinedo, Scheduling: theory, algorithms, and systems, 5th Edition, Springer, 2016.

[4] A. Felfernig, L. Hotz, C. Bagley, J. Tiihonen, Knowledge-based configuration: From research to business cases, Morgan Kaufmann, 2014.

[5] D. Jannach, M. Zanker, A. Felfernig, G. Friedrich, Recommender Systems: An Introduction, Cambridge University Press, 2010.

[6] J. Ceraso, A. Provitera, Sources of error in syllogistic reasoning, Cognitive Psychology 2 (4) (1971) 400–410.

[7] P. N. Johnson-Laird, Deductive reasoning, Annual review of psychology 50 (1999) 109–135.

[8] A. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, C. Wroe, OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns, in: 14th International Conference Engineering Knowledge in the Age of the Semantic Web (EKAW 2004), 2004, pp. 63–81.

[9] C. Roussey, O. Corcho, L. M. Vilches-Blázquez, A catalogue of OWL ontology antipatterns, in: International Conference On Knowledge Capture (K-CAP 2009), 2009, pp. 205–206.

[10] N. F. Noy, A. Chugh, W. Liu, M. A. Musen, A framework for ontology evolution in collaborative environments, in: International Semantic Web Conference (ISWC 2006), 2006, pp. 544–558.

[11] Q. Ji, P. Haase, G. Qi, P. Hitzler, S. Stadtmüller, RaDON—repair and diagnosis in ontology networks, in: European Semantic Web Conference (ESWC 2009), Springer, 2009, pp. 863–867.

[12] C. Meilicke, Alignment Incoherence in Ontology Matching, Ph.D. thesis, Universität Mannheim (2011).

[13] R. Reiter, A Theory of Diagnosis from First Principles, Artificial Intelligence 32 (1) (1987) 57–95.

[14] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, Consistency-based diagnosis of configuration knowledge bases, Artificial Intelligence 152 (2) (2004) 213–234.

[15] C. Mateis, M. Stumptner, D. Wieland, F. Wotawa, Model-Based Debugging of Java Programs, in: Fourth International Workshop on Automated Debugging (AADEBUG 2000), 2000.

[16] G. Friedrich, K. M. Shchekotykhin, A General Diagnosis Method for Ontologies, in: 4th International Semantic Web Conference (ISWC 2005), 2005, pp. 232–246.

[17] D. Jannach, T. Schmitz, Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach, Automated Software Engineering 23 (1) (2014) 105–144.

[18] G. Friedrich, M. Stumptner, F. Wotawa, Model-based diagnosis of hardware designs, Artificial Intelligence 111 (1-2) (1999) 3–39.

[19] L. Console, G. Friedrich, D. T. Dupré, Model-based diagnosis meets error diagnosis in logic programs, in: Automated and Algorithmic Debugging (AADEBUG 1993), 1993, pp. 85–87.

[20] P. Rodler, Interactive Debugging of Knowledge Bases, Ph.D. thesis, Alpen-Adria Universität Klagenfurt (2015).

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

[22] K. M. Shchekotykhin, G. Friedrich, P. Rodler, P. Fleiss, Sequential diagnosis of high cardinality faults in knowledge-bases by direct diagnosis generation, in: ECAI, Vol. 263, 2014, pp. 813–818.

[23] P. Rodler, K. Shchekotykhin, P. Fleiss, G. Friedrich, Balancing Brave and Cautious Query Strategies in Ontology Debugging, in: Joint Workshop on Knowledge Evolution and Ontology Dynamics (EvoDyn2011), 2011.

[24] K. Schekotihin, T. Schmitz, D. Jannach, Efficient sequential model-based fault-localization with partial diagnoses, in: International Joint Conference on Artificial Intelligence (IJCAI 2016), 2016, pp. 1251–1258.

[25] P. Rodler, W. Schmid, K. Schekotihin, A generally applicable, highly scalable measurement computation and optimization approach to sequential model-based diagnosis, CoRR abs/1711.05508.

[26] P. Rodler, W. Schmid, K. Schekotihin, Inexpensive cost-optimized measurement proposal for sequential model-based diagnosis, in: 28th International Workshop on Principles of Diagnosis (DX'17), Vol. 4, 2018, pp. 200–218.

[27] K. Shchekotykhin, G. Friedrich, P. Fleiss, P. Rodler, Interactive Ontology Debugging: Two Query Strategies for Efficient Fault Localization, Journal of Web Semantics 12-13 (2012) 88–103.

[28] K. Schekotihin, P. Rodler, W. Schmid, Ontodebug: Interactive ontology debugging plug-in for protégé, in: 10th International Symposium on Foundations of Information and Knowledge Systems (FoIKS 2018), 2018, pp. 340–359.

[29] K. Schekotihin, P. Rodler, W. Schmid, M. Horridge, T. Tudorache, A protégé plug-in for test-driven ontology development, in: Proceedings of the 9th International Conference on Biological Ontology (ICBO 2018), Corvallis, Oregon, USA, August 7-10, 2018., 2018.
URL http://ceur-ws.org/Vol-2285/ICBO_2018_paper_9.pdf

[30] C. Parnin, A. Orso, Are automated debugging techniques actually helping programmers?, in: International Symposium on Software Testing and Analysis (ISSTA '11), 2011, pp. 199–209.

[31] R. Ramler, D. Winkler, M. Schmidt, Random test case generation and manual unit testing: Substitute or complement in retrofitting tests for legacy code?, in: 38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2012), 2012, pp. 286–293.

[32] M. Staats, S. Hong, M. Kim, G. Rothermel, Understanding user understanding: Determining correctness of generated program invariants, in: International Symposium on Software Testing and Analysis (ISSTA 2012), 2012, pp. 188–198.

[33] G. Fraser, M. Staats, P. McMinn, A. Arcuri, F. Padberg, Does automated white-box test generation really help software testers?, in: International Symposium on Software Testing and Analysis (ISSTA 2013), 2013, pp. 291–301.

[34] A. Kalyanpur, B. Parsia, M. Horridge, E. Sirin, Finding all Justifications of OWL DL Entailments, in: 6th International Semantic Web Conference (ISWC 2007), 2007, pp. 267–280.

[35] M. Horridge, B. Parsia, U. Sattler, Laconic and Precise Justifications in OWL, in: 7th International Semantic Web Conference (ISWC 2008), 2008, pp. 323–338.

[36] U. Junker, QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems, in: AAAI, AAAI Press / The MIT Press, 2004, pp. 167–172.

[37] M. Gebser, J. Pührer, T. Schaub, H. Tompits, S. Woltran, spock: A debugging support tool for logic programs under the answer-set semantics, in: Applications of Declarative Programming and Knowledge Management, 2009, pp. 247–252.

[38] J. Oetsch, J. Pührer, H. Tompits, Catching the ouroboros: On debugging non-ground answer-set programs, Theory and Practice of Logic Programming 10 (4-6) (2010) 513–529.

[39] B. Parsia, E. Sirin, A. Kalyanpur, Debugging OWL ontologies, in: Proceedings of the 14th international conference on World Wide Web, 2005, pp. 633–640.

[40] S. Schlobach, Z. Huang, R. Cornet, F. van Harmelen, Debugging incoherent terminologies, Journal of Automated Reasoning 39 (3) (2007) 317–349.

[41] S. Schlobach, R. Cornet, Non-standard reasoning services for the debugging of description logic terminologies, in: Proceedings of the 18th International Joint Conference on Artificial Intelligence, 2003, pp. 355–362.

[42] A. Kalyanpur, Debugging and Repair of OWL Ontologies, Ph.D. thesis, University of Maryland, College Park (2006).

[43] F. Baader, R. Peñaloza, Axiom pinpointing in general tableaux, in: Proceedings of the 16th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, 2007, pp. 11–27.

[44] F. Baader, R. Peñaloza, Automata-based axiom pinpointing, J. Autom. Reasoning 45 (2) (2010) 91–129.

[45] X. Cheng, G. Qi, An algorithm for axiom pinpointing in EL+ and its incremental variant, in: 20th ACM Conference on Information and Knowledge Management (CIKM 2011), 2011, pp. 2433–2436.

[46] A. Ozaki, R. Peñaloza, Consequence-based axiom pinpointing, in: 12th International Conference on Scalable Uncertainty Management (SUM 2018), 2018, pp. 181–195.

[47] Y. Kazakov, P. Skocovský, Enumerating justifications using resolution, in: 9th International Joint Conference on Automated Reasoning (IJCAR 2018), 2018, pp. 609–626.

[48] R. Peñaloza, B. Sertkaya, Understanding the complexity of axiom pinpointing in lightweight description logics, Artificial Intelligence 250 (2017) 80–104.

[49] A. Kalyanpur, B. Parsia, M. Horridge, E. Sirin, Finding all justifications of OWL DL entailments, in: Proceedings of the 6th International Semantic Web Conference, 2007, pp. 267–280.

[50] K. M. Shchekotykhin, Interactive query-based debugging of ASP programs, in: 29th AAAI Conference on Artificial Intelligence (AAAI 2015), 2015, pp. 1597–1603.

[51] K. M. Shchekotykhin, G. Friedrich, P. Fleiss, P. Rodler, Interactive ontology debugging: Two query strategies for efficient fault localization, Journal of Web Semantics 12 (2012) 88–103.

[52] B. C. Grau, E. Jiménez-Ruiz, E. Kharlamov, D. Zheleznyakov, Ontology evolution under semantic constraints, in: KR, AAAI Press, 2012.

[53] U. Furbach, C. Schon, Semantically guided evolution of aboxes, in: TABLEAUX, Vol. 8123 of Lecture Notes in Computer Science, Springer, 2013, pp. 134–148.

[54] P. Rodler, On active learning strategies for sequential diagnosis, in: 28th International Workshop on Principles of Diagnosis (DX'17), Vol. 4, 2018, pp. 264–283.

[55] H. Wang, M. Horridge, A. L. Rector, N. Drummond, J. Seidenberg, Debugging OWL-DL ontologies: A heuristic approach, in: Proceedings of the 4th International Semantic Web Conference, 2005, pp. 745–757.

[56] C. Roussey, Ó. Corcho, O. Sváb-Zamazal, F. Scharffe, S. Bernard, Antipattern detection in web ontologies: an experiment using SPARQL queries, in: The Second International Workshop on Debugging Ontologies and Ontology Mappings (WoDOOM), 2012, pp. 45–56.

[57] A. L. Rector, S. Brandt, T. Schneider, Getting the foot out of the pelvis: modeling problems affecting use of SNOMED CT hierarchies in practical applications, Journal of the American Medical Informatics Association 18 (4) (2011) 432–440.

[58] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, G. Tack, Minizinc: Towards a standard CP modelling language, in: Principles and Practice of Constraint Programming (CP 2007), 2007, pp. 529–543.

[59] M. A. Musen, The Protégé Project: A Look Back and a Look Forward, AI Matters 1 (4) (2015) 4–12.

[60] O. Febbraro, K. Reale, F. Ricca, Aspide: Integrated development environment for answer set programming, in: 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011), 2011, pp. 317–330.

[61] J. Wielemaker, T. Schrijvers, M. Triska, T. Lager, SWI-Prolog, Theory and Practice of Logic Programming 12 (1-2) (2012) 67–96.

[62] C. Dodaro, P. Gasteiger, B. Musitsch, F. Ricca, K. Shchekotykhin, Interactive debugging of non-ground ASP programs, in: International Conference on Logic Programming and Nonmonotonic Reasoning (LPNR 2015), 2015, pp. 279–293.

[63] K. Leo, G. Tack, Debugging unsatisfiable constraint models, in: Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2017), 2017, pp. 77–93.

[64] A. Kalyanpur, B. Parsia, E. Sirin, B. C. Grau, J. A. Hendler, Swoop: A Web Ontology Editing Browser, Journal of Web Semantics 4 (2) (2006) 144–153.

[65] M. Horridge, S. Bail, B. Parsia, U. Sattler, The cognitive complexity of OWL justifications, in: Proceedings of the 10th International Semantic Web Conference, 2011, pp. 241–256.

[66] O. Šváb-Zamazal, V. Svátek, Analysing ontological structures through name pattern tracking, in: Knowledge Engineering: Practice and Patterns (EKAW 2008), 2008, pp. 213–228.

[67] O. Corcho, C. Roussey, L. M. V. Blázquez, I. Pérez, Pattern-based OWL Ontology Debugging Guidelines, in: Workshop on Ontology Patterns, 2009, p. 68.

[68] N. F. Noy, M. Crubézy, R. W. Fergerson, H. Knublauch, S. W. Tu, J. Vendetti, M. A. Musen, Protégé-2000: an open-source ontology-development and knowledge-acquisition environment, in: AMIA 2003 Open Source Expo, 2003, pp. 953–953.

[69] P. Rodler, K. Schekotihin, Reducing model-based diagnosis to knowledge base debugging, in: 28th International Workshop on Principles of Diagnosis (DX'17), Vol. 4, 2018, pp. 284–296.

[70] OWL 2 Web Ontology Language. Structural Specification and Functional-Style Syntax (Second Edition), Online at: http://www.w3.org/TR/owl2-syntax/.

[71] G. Qi, A. Hunter, Measuring incoherence in description logic-based ontologies, in: 6th International Semantic Web Conference (ISWC 2007), 2007, pp. 381–394.

[72] P. Rodler, M. Herold, StaticHS: A variant of Reiter's hitting set tree for efficient sequential diagnosis, in: 11th International Symposium on Combinatorial Search (SOCS 2018), 2018, pp. 72–80.

[73] A. Darwiche, Decomposable negation normal form, Journal of the ACM 48 (4) (2001) 608–647.

[74] Y.-F. Jiang, L. Lin, The computation of hitting sets with boolean formulas, Chinese Journal of Computers 26 (8) (2003) 919–924.

[75] P. Torasso, G. Torta, Model-based diagnosis through OBDD compilation: A complexity analysis, in: Reasoning, Action and Interaction in AI Theories and Systems, 2006, pp. 287–305.

[76] A. Metodi, R. Stern, M. Kalech, M. Codish, A novel sat-based approach to model based diagnosis, Journal of Artificial Intelligence Research 51 (2014) 377–411.

[77] R. Shearer, B. Motik, I. Horrocks, Hermit: A highly-efficient OWL reasoner, in: 5th Workshop on OWL: Experiences and Directions, Vol. 432 of CEUR Workshop Proceedings, 2008.

[78] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, Y. Katz, Pellet: A practical OWL-DL reasoner, Journal of Web Semantics 5 (2) (2007) 51–53.

[79] R. Fikes, P. Hayes, I. Horrocks, Owl-ql—a language for deductive query answering on the semantic web, Web semantics: Science, services and agents on the World Wide Web 2 (1) (2004) 19–29.

[80] P. Rodler, Towards better response times and higher-quality queries in interactive knowledge base debugging, Tech. rep., Alpen-Adria Universität Klagenfurt, http://arxiv.org/pdf/1609.02584v2.pdf (2016).

[81] P. Rodler, K. Shchekotykhin, P. Fleiss, G. Friedrich, RIO: Minimizing User Interaction in Ontology Debugging, in: Web Reasoning and Rule Systems (RR 2013), 2013, pp. 153–167.

[82] P. Rodler, W. Schmid, On the impact and proper use of heuristics in test-driven ontology debugging, in: Rules and Reasoning - Second International Joint Conference (RuleML+RR 2018), 2018, pp. 164–184.

[83] B. Settles, Active Learning, Morgan and Claypool Publishers, 2012.

[84] P. F. Patel-Schneider, I. Horrocks, B. Motik, OWL 1.1 Web Ontology Language: Structural Specification and Functional-Style Syntax, online, https://www.w3.org/Submission/owl11-overview/ (December 2006).

[85] I. Horrocks, O. Kutz, U. Sattler, The Even More Irresistible SROIQ, in: 10th International Conference on Principles of Knowledge Representation and Reasoning, 2006, pp. 57–67.

[86] M. Horridge, N. Drummond, J. Goodwin, A. L. Rector, R. Stevens, H. Wang, The Manchester OWL Syntax, in: OWL Experiences and Directions Workshop, CEUR Workshop Proceedings, 2006.

[87] G. Bhattacharyya, R. Johnson, Statistical Concepts and Methods, John Wiley & Sons Inc, 1977.

[88] J. R. Ruthruff, M. Burnett, G. Rothermel, An empirical study of fault localization for end-user programmers, in: 27th International Conference on Software Engineering (ICSE '05), 2005, pp. 352–361.

[89] B. Hofer, F. Wotawa, Fault localization in the light of faulty user input, in: 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS 2015), 2015, pp. 282–291.

[90] X. Guo, M. Zhou, X. Song, M. Gu, J. Sun, First, debug the test oracle, IEEE Transactions on Software Engineering 41 (10) (2015) 986–1000.

Appendix A. Formal Characterization of the Complexity Prediction Model

The suggested prediction model for query complexity is a function M that maps a query Q—consisting of a set of OWL[25] axioms[26] —to a real-valued score in $(0, 1]$ where 1 means maximally easy and 0 maximally hard, respectively. Intuitively, $M(Q)$ can be interpreted as an estimate of the query's probability to be comprehended properly by a user.

The assumption behind the model is that different expressions (logical operators, quantifiers, etc.) appearing in axioms have different complexities. To describe these complexities, we use a set of weights that are chosen empirically and based on our expertise. These weights are incorporated into a set of manually defined recursive rules. We use these rules to derive the complexity of a given query by decomposing it stepwise to its smallest components.

Computationally, the underlying idea of the model is to first extract from the query the axioms it consists of. Each of these axioms is then reduced to the class expressions it comprises. The class expressions are then recursively split into smaller sub-expressions until atomic classes are obtained. Based on the structure of the axiom found by this recursive reduction, the model uses the specified weights to compute the complexity of the axiom. Finally, the complexities of all query axioms are combined to compute the final query score.

In the following, we will describe in more detail **(I)** how axiom complexities are used to determine the overall query complexity, **(II)** how axiom complexities are derived based on the class expressions occurring in them and **(III)** how the complexities of the class expressions are calculated.

The function M makes use of two additional functions. The function M_{ax} computes for a given OWL axiom its estimated probability in $(0, 1]$ of being

[25]Whenever we write OWL in this section, we mean the OWL 2 Web Ontology Language, as specified in [70].

[26]Currently, the model supports only class expression axioms. It can however be extended to cover object property, data property and assertion axioms as well.

understood correctly; M_{ce} computes for a given OWL class expression its complexity in terms of a real number in $[1, \infty)$.

(I) Overall query complexity: Let $Q = \{ax_1, \ldots, ax_k\}$ be a query consisting of the OWL axioms ax_i, $i \in \{1, \ldots, k\}$. Then, we define

$$M(Q) := \prod_{i=1}^{k} M_{ax}(ax_i)$$

That is, the probability of Q being answered correctly is equal to the probability of all axioms in Q being answered correctly (assuming independence between the axioms).

(II) Axiom complexity: Let ax be an OWL (class expression) axiom. An axiom ax has one of the following forms [70] for some integer $s \geq 2$ and arbitrary OWL class expressions $X_1, \ldots, X_s$:

- X_1 `SubClassOf` X_2
- `EquivalentClasses` $X_1 \ldots X_s$
- `DisjointClasses` $X_1 \ldots X_s$
- `DisjointUnion` $X_1 \ldots X_s$

We denote by $CE(ax)$ the set of all class expressions occurring in ax and specify

$$M_{ax}(ax) = \prod_{X_i \in CE(ax)} \frac{1}{M_{ce}(X_i)}$$

That is, the probability of understanding the entire axiom is equal to the probability of properly comprehending all class expressions occurring in the axiom (assuming independence between the user's understanding of the individual expressions). The estimated probability of comprehending a class expression is inversely proportional to the complexity of the expression, as assessed by M_{ce}.

(III) Class expression complexity:[27] We define M_{ce} recursively as follows. Let $X_1, X_2, X_3, X_4, X_5, X_6$ be (complex or atomic) OWL class expressions, A

[27]For brevity of notation we use Description Logic Syntax in the following description wherever possible. E.g., "$\sqcap$", "$\sqcup$", "$\neg$" stand for the OWL Manchester Syntax keywords `and`, `or` and `not`, respectively. For details see [86, Fig. 3].

an atomic OWL class, and C_1, C_2 complex OWL class expressions. With an atomic OWL class we associate a named class, $\top$, $\bot$, or an enumeration of individuals[28]. Further, let r_o be an OWL object property, r_d an OWL data property, r an OWL (data or object) property, R a data range, $\mathbf{Q} \in \{\forall, \exists\}$, $\mathbf{N} \in \{=, \leq, \geq\}$, as well as m a non-negative integer, v an individual and l a literal. Then:

$$
\begin{aligned}
M_{ce}(A \sqcap C_1) &= M_{ce}(C_1 \sqcap A) = M_{ce}(A) \cdot (1 + M_{ce}(C_1)) \\
&\text{if } C_1 = X_3 \sqcup X_4 \\
M_{ce}(C_1 \sqcap C_2) &= (1 + M_{ce}(C_1)) \cdot (1 + M_{ce}(C_2)) \\
&\text{if } C_1 = X_3 \sqcup X_4, C_2 = X_5 \sqcup X_6 \\
M_{ce}(X_1 \sqcap X_2) &= M_{ce}(X_1) \cdot M_{ce}(X_2) \\
M_{ce}(A \sqcup C_1) &= M_{ce}(C_1 \sqcup A) = M_{ce}(A) \cdot (1 + M_{ce}(C_1)) \\
&\text{if } C_1 = X_3 \sqcap X_4 \\
M_{ce}(C_1 \sqcup C_2) &= (1 + M_{ce}(C_1)) \cdot (1 + M_{ce}(C_2)) \\
&\text{if } C_1 = X_3 \sqcap X_4, C_2 = X_5 \sqcap X_6 \\
M_{ce}(X_1 \sqcup X_2) &= M_{ce}(X_1) \cdot M_{ce}(X_2) \\
M_{ce}(\mathbf{Q}\, r_o\, A) &= M_{ce}(\mathbf{N}\, m\, r_o\, A) = 1 + M_{ce}(A) \\
M_{ce}(\mathbf{Q}\, r_o\, C_1) &= M_{ce}(\mathbf{N}\, m\, r_o\, C_1) = 2 + M_{ce}(C_1) \\
M_{ce}(\mathbf{Q}\, r_d\, R) &= M_{ce}(\mathbf{N}\, m\, r_d\, R) = M_{ce}(\mathbf{N}\, m\, r) = 2 \\
M_{ce}(\texttt{ObjectHasValue}\, r\, v) &= 2 \\
M_{ce}(\texttt{ObjectHasSelf}\, r) &= 2 \\
M_{ce}(\texttt{DataHasValue}\, r\, l) &= 2 \\
M_{ce}(A) &= 1 \\
M_{ce}(\neg A) &= 1.25
\end{aligned}
$$

[28] OWL keyword `ObjectOneOf` [70].

$$M_{ce}(\neg C_1) = 2 \cdot M_{ce}(C_1)$$

Importantly, each class expression ce is evaluated from top to bottom, i.e., the first of the above equations that is applicable is used to assess ce.