Test-driven Ontology Development in Protégé

Konstantin Schekotihin, Patrick Rodler, Wolfgang Schmid
University of Klagenfurt
Universitätsstr. 65-67
Klagenfurt, Austria
Email: firtstname.lastname@aau.at

Matthew Horridge, Tania Tudorache
Stanford University
1265 Welch Rd
Stanford, California, USA
Email: lastname@stanford.edu

CEUR-WS.org/Vol-2285/ICBO_2018_paper_31.pdf

***Abstract*—Over the past decade, various quality assurance methodologies have emerged in the field of software engineering for preventing, detecting, and fixing faults and bugs in software. In particular, Test-driven Development (TDD) is now a popular quality assurance technique whereby extensive usage of test cases can be used to enforce the correctness of software artifacts. While testing has made its way into the field of ontology engineering, where some techniques for testing ontologies are now used in prominent biomedical ontology projects, *Test-driven Development* has yet to achieve significant uptake.**

In this paper we propose a *Logic-Based Test-driven Ontology Development* methodology, which takes cues from Test-Driven Development in the field of software engineering. Our hope is that this will encourage a "test-first" approach to ontology engineering and that it will form part of the arsenal available to ontology engineers in order to help them produce and maintain high quality ontologies. Test cases in our framework are represented by simple statements describing expected and/or unwanted logical consequences of an intended ontology. As with Test-driven Development in software engineering, our approach encourages small, frequent iterations in the testing and development life-cycle. We provide and present tool support for our approach in the form of OntoDebug – a plug-in for the ontology editor Protégé.

I. Introduction

Ontology creation and maintenance are complex and error-prone activities. In part, this is due to the difficulties that experts have with respect to encoding domain knowledge in the form of logical descriptions [11]. It can also be due to the misuse or misunderstanding of the knowledge representation language that is used to build the ontology, fundamental errors in the used modeling schemata and, attempts to fix bugs in logical definitions without understanding their causes [14], [22]. Well-known examples of such modeling errors reported in [15] lead to incorrect inferences, such as "a foot is part of a pelvis" or "diabetes is a disease of the abdomen". Identification of the causes of such erroneous inferences is a complex task that might require a developer to review hundreds or even thousands of logical statements. In large projects that involve multiple experts working on an ontology, such review processes can be very complex and might require multiple domain experts to analyze subsets of the ontology together. Our experience of such projects, and observations of large biomedical ontology engineering projects, has convinced us that having the possibility of specifying test cases is critical for ensuring the quality of published ontologies.

The problems listed above are not unique for the ontology development process. Indeed, similar problems can be observed in various areas where experts try to formalize their knowledge about the domain using declarative or procedural languages. For instance, in software development, bugs might also be introduced due to inability to write a correct algorithm, a misunderstanding of language constructs, incorrect modeling, etc. Over the past few decades, various methodologies have been suggested that aim to simplify the development processes with work-flows that help to prevent, identify and localize bugs early. Examples of such methodologies in software engineering include the waterfall model [3], V-Model [6], or agile methods [2]. The main idea behind all these development models is to organize knowledge-based creative work in a controlled and systematic way.

Test-driven development (TDD) is one of such methodologies in which the specification of requirements for a software artifact and the subsequent verification of the specification is done by means of test cases [1]. The TDD process involves small iterations in which developers first specify a number of test cases, then provide an implementation, and finally verify it by running the test cases. If some test cases fail, then the implementation is revised. When all tests are satisfied, the process starts over from the definition of more test cases. These small iterations continue until the artifact is finished.

There are two types of test cases considered in the literature: syntactic and logic-based. The syntactic approaches are mostly focused on the analysis of ontology axioms using some predefined [4] or programmed [16] patterns, SPARQL queries [8], etc. Each pattern corresponds to some good (bad) ontology development practice and defines lexical structures that should (must not) appear in the ontology. The notion of a logic-based test case – a (set of) axiom(s) that must or must not be entailed by the target ontology – was first introduced in [7] and later appeared in, e.g., [24], [28]. The suggested approaches allow for ontology testing and debugging to ensure correctness of the knowledge encoded in the current version of an ontology. That is, all defined test cases specifying (non-)entailments of the target ontology must hold for the ontology at hand.

Unfortunately, most of the existing approaches do not provide the level of tool support necessary for the successful application of TDD in practice. For instance, they can visualize

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

changes in entailments wrt. the last modifications of an ontology [12] or provide testing services for specific ontologies [29]. In the most advanced approaches, like OntologyTest [8] or TDDOnto2 [5], a developer can define syntactic and/or logic-based tests for some selected types of axioms, e.g. class or assertion ones, and verify whether the tests hold for a given ontology. However, these tests can only be used to express requirements to entailed axioms and cannot be applied to identify various types of unwanted entailments. Moreover, if a test case fails, such tools provide no support for localization and repair of detected bugs.

Contributions. In this paper we present a novel approach to *logic-based test driven development (TDD)* for ontology engineering. We show how logic-based TDD can be incorporated into ontology engineering work-flows and we present tools that support the process. Our contributions are as follows:

- We present an approach to logic-based TDD that aims to facilitate the creation of high-quality ontologies by means of a precise requirements specification and a tight verification loop.
- We show how a logic-based TDD process can be executed in Protégé using the *OntoDebug*[1] plug-in. This plugin supports the definition of test cases in the form of general class axioms which must be entailed or not entailed by the target ontology [24]. In this way the developer can specify requirements for the target ontology and automatically verify them using the plug-in.
- For failed tests, we demonstrate how a user can identify causes of the problem in the target ontology using the debugging interface provided by OntoDebug. This part of our framework implements a number of published algorithms and strategies [7], [10], [17], [18], [19], [20], [21], [25], [26] allowing for a high-performance localization of fault causes using off-the-shelf reasoners available in Protégé, such as Pellet [27] or Hermit [13].
- Finally, we present an overview of the possible ways of fixing an ontology in the TDD cycle using the default Protégé toolbox as well as the repair interface of OntoDebug. Our approach facilitates the testing of alterations to the target ontology in a safe, non-intrusive way – that is, modifications are performed on a virtual copy of the ontology and are applied to the target ontology only if the user is satisfied with the result and all test cases pass.

In the following we present the logic-based TDD methodology in Section II and show how it can be implemented using Protégé with the OntoDebug plug-in in Section III.

II. Test-driven Ontology Development

Test-driven ontology development (TDD) is an ontology development process that involves the *specification* of requirements with respect to (a part of) the target ontology, the *implementation* of logical descriptions/definitions that are required to support the test cases, as well as the *verification* of the obtained ontology with respect to the specification. The requirements are specified as *test cases*, each of which is a general class axiom (GCI). We also assume that any existing test cases that are present in the existing ontology are consistent with it.

[1]See http://isbi.aau.at/ontodebug for the source code and documentation.

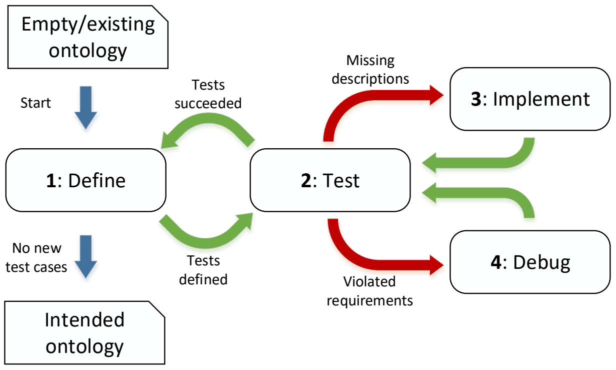


Fig. 1. Workflow of test-driven ontology development

The TDD work-flow, shown in Fig. 1, starts with either an empty ontology, or an existing ontology that needs to be extended. It comprises four main activities: definition of test cases (***Define***), their execution (***Test***), ontology editing (***Implement***), and ontology debugging (***Debug***). Where, the latter step only takes place if any test cases fails. In what follows we discuss each activity in more detail:

a) ***Define****:* In the first step, the developer defines some test cases. Test cases are split into two categories (1) *Positive* test cases and (2) *Negative* test cases. For both types, an axiom is specified that, must be entailed in the case of positive test cases (for the test to pass), and must not be entailed in the case of negative test cases (for the test to pass). Positive test cases may also have a set of *supporting axioms* specified that are not part of the ontology under test, but are required in combination with the ontology for the axiom to be entailed and for the test to pass. For example, one could add the supporting axiom `x Type Secretary` (for a fictitious new individual `x`) in order to check for `x Type Employee` i.e. whether secretaries are correctly modeled as employees.

A positive test case can be seen as a check that the ontology under test contains enough (correct) axioms such that the specified axiom holds as an entailment. A negative test case can be seen as a check that the ontology under test does not contain axioms such that the specified GCI holds as an entailment. An example of a negative test case would be the axiom `Student SubClassOf Professor` which asserts that students must not be entailed to be professors.

It should be noted that, in order to specify a test case the user should clearly understand the logical (non-)consequences of the ontology under test, or, in other words, the target domain to be modeled. Furthermore, as is the case with software development, aiming for one hundred percent test coverage is hard. Therefore, in practice, we expect that test cases will be defined for portions of the ontology under test and newly added extensions to the ontology. We hope that the positive effect of this approach is that it will help to ensure the quality of newly added class descriptions.

b) ***Test:*** In the testing step, the developer executes the test cases. If all test cases pass then the current TDD iteration is finished. As a result, the ontology under test corresponds to the intended one with respect to the portions covered by the test cases. In this case, the developer can return to the specification of further test cases or finish the TTD process, meaning that no further test cases are envisaged.

If some test case fails, one or both of the following scenarios are given: *(i) missing descriptions*, i.e. the developed ontology lacks axioms that support the entailment required by the test case, and *(ii) violated requirements*, i.e. axioms in the developed ontology *contradict* the specified requirements.

The first scenario only occurs if there is a positive test case which requires the ontology under test (plus supporting axioms) to entail an axiom, but this axiom is not entailed. Therefore, the developer should proceed to the *implement* step of the process below and extend the ontology.

The second scenario indicates that axioms in the ontology under test (plus supporting axioms) do not allow all test cases and other quality requirements (e.g., consistency, no unsatisfiable classes) to be satisfied. More precisely, when assuming all positive test cases (which constitute required consequences) being added to the ontology, the resulting ontology violates at least one negative test case (i.e., has an unwanted consequence) or becomes inconsistent or incoherent.

Due to logical monotonicity, a further development of the ontology (in terms of adding additional axioms) in this second scenario is meaningless, since the obtained result cannot correspond to the intended ontology as specified by the test cases and other requirements. Therefore, the developer should proceed to the *debug* step and try to improve the ontology by localizing and repairing the axioms responsible for the observed deficiencies.

c) ***Implement:*** This is a standard development task in Protégé for which the editor was originally designed. The added value of adhering to the suggested TDD paradigm is that it helps the developer to differentiate between ontology parts that need further implementation (addition of axioms) and those that require refactoring (deletion or modification of axioms), based on the outcome of test case verifications. That is, failed test cases referring to missing axioms call for further implementation activities in the respective part of the ontology, whereas failed test cases pointing out inconsistencies suggest repair activities in the relevant ontology parts (see *Debug* below). To keep the process simple and manageable, TDD, by its very nature, postulates that developers stick to local and small changes and proceed by systematic specification and verification of test cases.

d) ***Debug:*** The debugging phase starts when the test cases have been executed and one or more of them fails, i.e. when the target ontology violates the requirements defined by the failed test cases, or when the ontology is inconsistent or incoherent. The goal of this procedure is to determine the faulty axioms in the ontology and to support the developer while repairing those axioms. Taking into account the complexity of inferences possible in expressive languages, such as OWL [9], the debugging task is almost impossible without sophisticated tool support. In Protégé one can use two mechanisms for fault localization: computation of justifications [10] and debugging with OntoDebug [23].

III. Protégé support for TDD

Protégé is one of the most popular open-source tools for the development of ontologies and it provides rich functionality for the creation, modification, querying, and visualization capability. In addition, Protégé has built-in support for various extensions – plug-ins – that can enhance user experience by providing additional functionality. OntoDebug is one such plug-in that in conjunction with the standard Protégé functionality provides the support that is necessary for TDD activities (see Fig. 2), including:

a) *definition of test cases*, allowing for specification of requirements to the intended ontology;
b) *execution of test cases*, indicating if there are any violations of requirements or missing axioms;
c) *ontology development*, enabling definition of axioms, automated refactoring, visualizations, etc.;
d) *fault localization*, identifying ontology axioms that must be corrected such that the intended ontology can be formulated; and
e) *repair*, helping the developer to correct the localized faulty axioms.[2]

In what follows, we discuss this functionality in more detail. We use the simple Koala tutorial ontology to exemplify the main notions.

Example 1. Koala[3] is an educational ontology, which exhibits modeling problems that occur due to misunderstandings of description logics. This ontology defines a number of classes, such as **`Person`**, **`Marsupials`**, **`Koala`**, **`Quokka`**, **`Female`**, etc., and some properties like **`isHardWorking`**, **`hasDegree`**, etc.

a) ***Definition of test cases:*** Positive and negative test cases can be defined using the “Saved Test Cases” view of the OntoDebug plug-in, shown in Fig. 3. In our example, the developer has specified three test cases. The two positive tests define expected entailments of the intended ontology, namely, that females and students are persons. The negative test case asserts that the axiom **`Person`** `SubClassOf` **`Marsupials`** must not be entailed by the ontology.

b) ***Execution of test cases:*** OntoDebug supports two modes: manual and automatic test verification. Complete verification of all test cases must be started manually using the “Start” button highlighted in Fig. 3. Manual verification

[2]It should be noted that OntoDebug is still under active development. It is possible that future versions might deviate in functionality as well as in look & feel.

[3]This ontology can be loaded from bookmarks in `Open from URL...` menu of Protégé

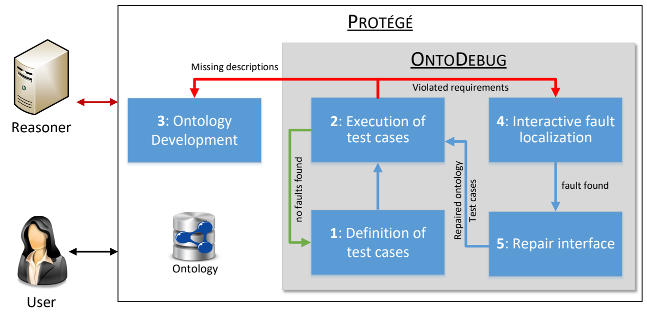
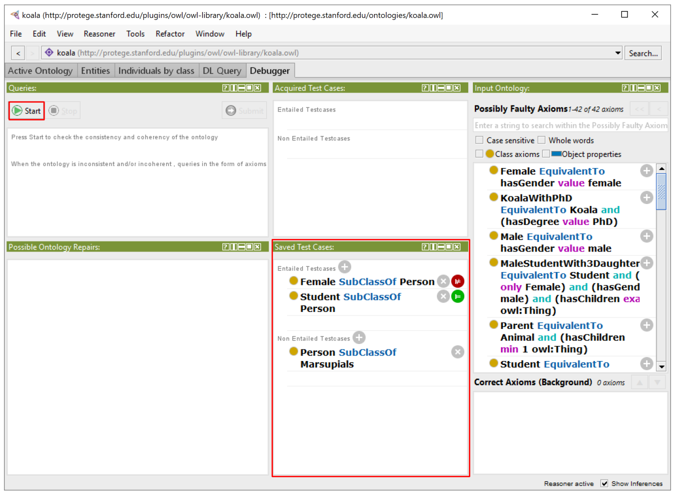


Fig. 2. Test-driven development in Protégé with OntoDebug



koala (http://protege.stanford.edu/plugins/owl/owl-library/koala.owl) : [http://protege.stanford.edu/ontologies/koala.owl]

File Edit View Reasoner Tools Refactor Window Help

koala (http://protege.stanford.edu/plugins/owl/owl-library/koala.owl)

Active Ontology | Entities | Individuals by class | DL Query | Debugger

Queries:

Start Stop Submit

Press Start to check the consistency and coherency of the ontology

When the ontology is inconsistent and/or incoherent , queries in the form of axioms

Acquired Test Cases:

Entailed Testcases

Non Entailed Testcases

Possible Ontology Repairs:

Saved Test Cases:

Entailed Testcases
- Female SubClassOf Person
- Student SubClassOf Person

Non Entailed Testcases
- Person SubClassOf Marsupials

Input Ontology:

Possibly Faulty Axioms 1-42 of 42 axioms

Enter a string to search within the Possibly Faulty Axiom

☐ Case sensitive ☐ Whole words

☐ Class axioms ☐ Object properties

- Female EquivalentTo hasGender value female
- KoalaWithPhD EquivalentTo Koala and (hasDegree value PhD)
- Male EquivalentTo hasGender value male
- MaleStudentWith3Daughter EquivalentTo Student and (only Female) and (hasGend male) and (hasChildren exa owl:Thing)
- Parent EquivalentTo Animal and (hasChildren min 1 owl:Thing)
- Student EquivalentTo

Correct Axioms (Background) 0 axioms

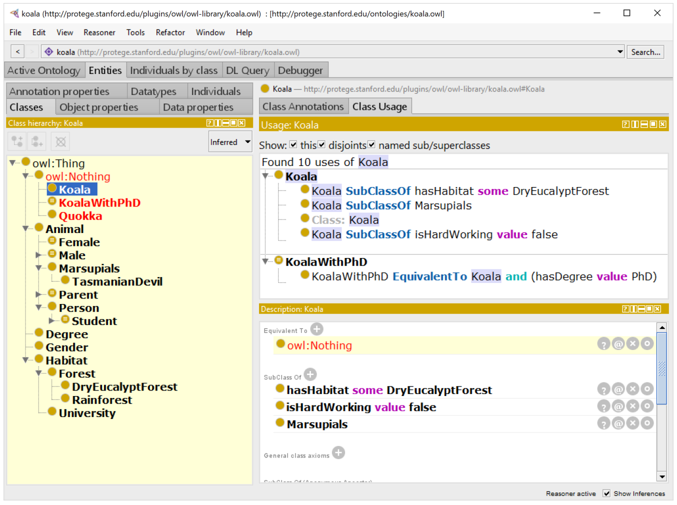
Reasoner active ✓ Show Inferences

Fig. 3. Specification of test cases for the Koala ontology

is the default behaviour (like reasoner activation in Protégé) since frequent automatic triggering of reasoning services might result in undesired freezing of the user interface. If test failures are encountered, or the ontology is inconsistent, or the ontology comprises unsatisfiable classes, then OntoDebug will automatically proceed to the debugging step.

Test results are indicated with a green or a red icon to the right of the test specification. For example, since the Koala ontology entails that **`Student`** is a subclass of **`Person`**, the corresponding test case is labeled with a green icon. In contrast, the test case stating that **`Female`** is a subclass of **`Person`** is not entailed by the ontology and is therefore assigned the red icon. The developer should therefore switch to Protégé's "Entities" tab and augment the ontology with axioms to ensure that this is entailed.

*c) **Ontology development**:* In the development stage the ontology is modified according to the criteria defined in the test cases using a variety of tools available in Protégé. To verify the progress, i.e. whether all defined requirements are fulfilled, the developer can switch to OntoDebug and check the labels of the positive test cases or start a reasoner. The latter step allows for verification of the ontology for consistency, satisfiability of its classes, getting the inferred axioms and analyzing justifications



koala (http://protege.stanford.edu/plugins/owl/owl-library/koala.owl) : [http://protege.stanford.edu/ontologies/koala.owl]

File Edit View Reasoner Tools Refactor Window Help

koala (http://protege.stanford.edu/plugins/owl/owl-library/koala.owl)

Active Ontology | Entities | Individuals by class | DL Query | Debugger

Annotation properties | Datatypes | Individuals | Classes | Object properties | Data properties

Class hierarchy: Koala — Inferred

- owl:Thing
  - owl:Nothing
    - Koala
    - KoalaWithPhD
    - Quokka
  - Animal
    - Female
    - Male
    - Marsupials
      - TasmanianDevil
    - Parent
    - Person
      - Student
  - Degree
  - Gender
  - Habitat
    - Forest
      - DryEucalyptForest
      - Rainforest
    - University

Koala — http://protege.stanford.edu/plugins/owl/owl-library/koala.owl#Koala

Class Annotations | Class Usage

Usage: Koala

Show: ✓ this ✓ disjoints ✓ named sub/superclasses

Found 10 uses of Koala

- Koala
  - Koala SubClassOf hasHabitat some DryEucalyptForest
  - Koala SubClassOf Marsupials
  - Class: Koala
  - Koala SubClassOf isHardWorking value false
- KoalaWithPhD
  - KoalaWithPhD EquivalentTo Koala and (hasDegree value PhD)

Description: Koala

Equivalent To
- owl:Nothing

SubClass Of
- hasHabitat some DryEucalyptForest
- isHardWorking value false
- Marsupials

General class axioms

Reasoner active ✓ Show Inferences

Fig. 4. Detected fault in the Koala ontology

for these inferences with the Protégé Explanation Plug-In.

If, during development and regular reasoning, the developer notices an erroneous entailment, she can add it as a negative test case at the press of a button. Similarly, correct entailments can be added as positive test cases. Violations of consistency and coherency requirements do not require any additional definitions and OntoDebug can start with the fault localization immediately.

Example 2. Reconsider the Koala ontology, in which, due to mistakes made while developing the ontology, manual invocation of a reasoner infers unsatisfiable classes: **`Koala`**, **`KoalaWithPhD`**, and **`Quokka`**. Protégé shows these classes in red (see Fig. 4) indicating the developer that the requirement for coherency is violated. Given these three violations, the developer can proceed to OntoDebug and start the debugging session.

*d) **Fault Localization**:* After a fault is detected, the fault localization process starts with definition of a "background theory". This is a set of ontology axioms that should be considered as correct by the debugger, thus, allowing it to focus on the relevant ontology parts. In the user interface of OntoDebug, shown in Fig. 5 (1), this can be done by finding, selecting, and moving axioms from the list of *possibly* faulty axioms to the list of *correct* axioms. Now imagine that the developer has modified only **`SubClassOf`** axioms before observing the fault in Example 2 and would like the debugger to focus on these axioms only. In this case, all assertions and property restrictions should be moved from the list of Possibly Faulty Axioms to the list of Correct Axioms in the Input Ontology view (1).

After this step the developer starts an interactive debugging session by pressing the "Start" button, just as for the verification of test cases. In the first step, the debugging algorithms compute repairs, presented in Fig. 5 (2), which are subsets of possibly faulty axioms. All axioms of a repair must be modified or in the simplest case merely removed in order to correct the fault in the developed ontology.

In case the information provided by test cases is insufficient the debugger might find a number of alternative repairs. Therefore, additional test cases must be acquired allowing for identification of the correct repair. The acquisition process is completely automated in OntoDebug. Thus, the debugger generates and asks the developer a sequence of queries about desired entailments and non-entailments of the intended ontology in the Queries view, shown in Fig. 5 (3). The acquisition of new test cases continues until the developer spots the correct repair in the list or only one repair remains.

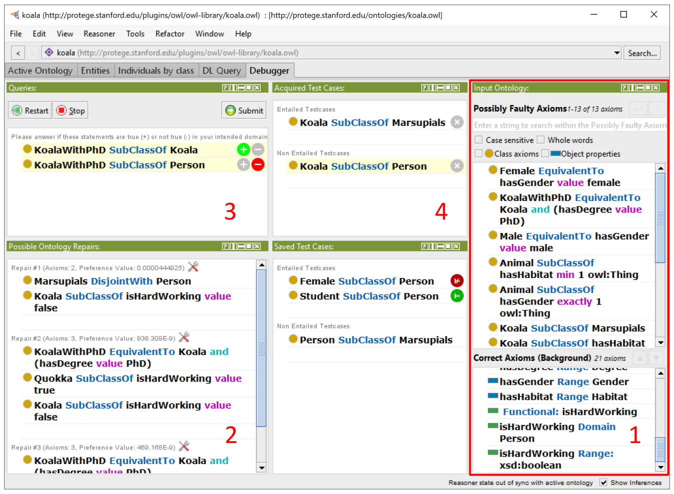
Example 3. Continuing with the Koala example, the three classes in this ontology are unsatisfiable since the developer used one of the two properties **`isHardWorking`** and **`hasDegree`** with the domain **`Person`** to describe each of the three classes listed above. Given this domain definition, the reasoner derives that **`Koala`**, **`KoalaWithPhD`**, and **`Quokka`** are subclasses of the class **`Person`**. However, the ontology comprises an axiom that defines classes **`Marsupials`** and **`Person`** as disjoint. This results in unsatisfiability of the three classes, which are defined as subclasses of the class **`Marsupials`**.

When the developer starts the debugger it is able to identify 10 possible repairs and the interactive debugging session is started. In the second iteration of this session, see Fig. 5 (3), the developer is asked whether or not "a Koala with PhD is a Koala" and "a Koala with PhD is a Person" must be entailed by the ontology. These queries can be answered by the developer using the common knowledge about the domain of koalas. Thus, the first axiom is classified positively whereas the second axiom – negatively.

After the "Submit" button is pressed, OntoDebug adds the axioms to the set of positive or negative test cases according to the answers of the previous questions. In our example, the first question comprised two axioms: "a Koala is a Marsupial" which was answered positively and "a Koala is a Person" which was classified as negative, as presented in Fig. 5 (4).

OntoDebug can tolerate cases when the developer has trouble with the classification of axioms when answering questions. For instance, if some axiom cannot be classified as either positive or negative, the developer can simply skip it and leave the classification unspecified. OntoDebug will then try to find a different question whose axioms might be easier for the developer to classify. In addition to this, if the developer has misclassified an axiom and would like to backtrack, she can simply remove this axiom from either of the Acquired Test Cases lists. OntoDebug will automatically update the set repairs and suggest a new question.

*e) **Repair**:* After a correct repair is identified, the developer should introduce modifications to the developed ontology and modify at least all axioms of the repair. For this purpose one can use either the standard toolkit of Protégé or the repair interface of OntoDebug, presented in Fig. 6. This interface reuses Protégé's built-in editors and enables safe, non-invasive per-axiom modification. In addition, the user can use Protégé's


koala (http://protege.stanford.edu/plugins/owl/owl-library/koala.owl) : [http://protege.stanford.edu/ontologies/koala.owl]

Queries:

Please answer if these statements are true (+) or not true (-) in your intended domain
- KoalaWithPhD SubClassOf Koala
- KoalaWithPhD SubClassOf Person

Acquired Test Cases:

Entailed Testcases
- Koala SubClassOf Marsupials

Non Entailed Testcases
- Koala SubClassOf Person

Possible Ontology Repairs:

Repair #1 (Axioms: 2, Preference Value: 0.0000444925)
- Marsupials DisjointWith Person
- Koala SubClassOf isHardWorking value false

Repair #2 (Axioms: 3, Preference Value: 938.309E-9)
- KoalaWithPhD EquivalentTo Koala and (hasDegree value PhD)
- Quokka SubClassOf isHardWorking value true
- Koala SubClassOf isHardWorking value false

Repair #3 (Axioms: 3, Preference Value: 469.168E-9)
- KoalaWithPhD EquivalentTo Koala and (hasDegree value PhD)

Saved Test Cases:

Entailed Testcases
- Female SubClassOf Person
- Student SubClassOf Person

Non Entailed Testcases
- Person SubClassOf Marsupials

Input Ontology:

Possibly Faulty Axioms *1-13 of 13 axioms*

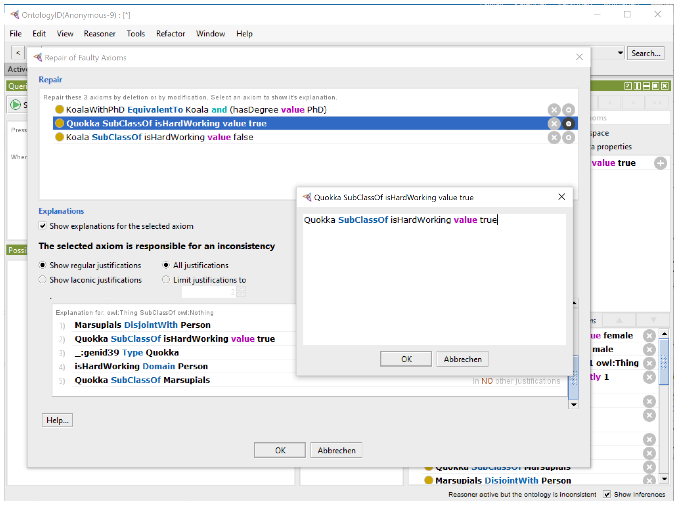
Enter a string to search within the Possibly Faulty Axiom

☐ Case sensitive ☐ Whole words ☐ Class axioms ☐ Object properties
- Female EquivalentTo hasGender value female
- KoalaWithPhD EquivalentTo Koala and (hasDegree value PhD)
- Male EquivalentTo hasGender value male
- Animal SubClassOf hasHabitat min 1 owl:Thing
- Animal SubClassOf hasGender exactly 1 owl:Thing
- Koala SubClassOf Marsupials
- Koala SubClassOf hasHabitat

Correct Axioms (Background) *21 axioms*
- hasGender Range Gender
- hasHabitat Range Habitat
- Functional: isHardWorking
- isHardWorking Domain Person
- isHardWorking Range: xsd:boolean

Reasoner state out of sync with active ontology ☑ Show Inferences

Fig. 5. Interactive debugging session (second iteration)


OntologyID(Anonymous-9) : [*]

Repair of Faulty Axioms

Repair

Repair these 3 axioms by deletion or by modification. Select an axiom to show it's explanation.
- KoalaWithPhD EquivalentTo Koala and (hasDegree value PhD)
- Quokka SubClassOf isHardWorking value true
- Koala SubClassOf isHardWorking value false

Explanations

☑ Show explanations for the selected axiom

The selected axiom is responsible for an inconsistency

(●) Show regular justifications (●) All justifications
( ) Show laconic justifications ( ) Limit justifications to 2

Explanation for: owl:Thing SubClassOf owl:Nothing
1) Marsupials DisjointWith Person
2) Quokka SubClassOf isHardWorking value true
3) _:genid39 Type Quokka
4) isHardWorking Domain Person
5) Quokka SubClassOf Marsupials

In NO other justifications

Quokka SubClassOf isHardWorking value true

Quokka SubClassOf isHardWorking value true

Reasoner active but the ontology is inconsistent ☑ Show Inferences

Fig. 6. Repair interface of OntoDebug

explanation functionality to analyze relations between some selected axiom of the repair and other axioms in the developed ontology that result in a fault. This feature is supposed to give the user a hint of how the axiom contributes to the fault in question and what to change in the axiom in order to obtain a repaired ontology.

Example 4. Consider an explanation for the second axiom of the selected repair, shown in Fig. 6 for the Koala example. This explanation comprises five relevant axioms which indicate that a **`Quokka`** is a **`Marsupials`** (axiom 5) and a **`Person`** (axioms 2 and 4) simultaneously. However, since **`Marsupials`** and **`Person`** are disjoint classes (axiom 1), the **`Quokka`** class is unsatisfiable.

In order to ensure that the modifications actually correct faults in the ontology, the repair interface provides a possibility to check whether all requirements are fulfilled or not simply by clicking on the "Ok" button. If modifications introduced in the repair process do not correct previously localized faults or they introduce new ones, violating any of the requirements, then the plug-in will start a new debugging session automatically. If this is not the case, then OntoDebug informs the developer that the obtained ontology is correct.

After all faults are repaired, the performed modifications can be committed to the ontology and all acquired test cases saved. In this way our debugger enables a continuous acquisition and maintenance of test cases, which is a key aspect of the TDD process. The more test cases, the higher the likelihood that the resulting ontology is fault-free.

When a TDD session finishes, the user can

a) return to the first step and start with the definition of new test cases providing specifications for some unimplemented parts of the ontology, or

b) finish the development and save the resulting ontology.

Finally, it should be noted that the current functionality provided by Protégé and OntoDebug does not focus on the validation of the entire ontology, but rather pursues a symptom-driven fault repair. That is, once a deficiency, e.g. an inconsistency or a violation of a test case, becomes evident, OntoDebug assists in restoring the ontology correctness in terms of the *given or acquired test cases*. As part of future work, we plan to develop strategies for test case suggestion given an ontology that *is* compatible with the desired criteria. Such strategies could, for instance, be guided by logical counterparts to "code smells" from the field of software engineering, such as OWL anti-patterns [22] or common modeling mistakes [14]. Finally, we envisage that it will also be possible in the future to make the execution of test cases part of the Continuous Integration style of development that is becoming popular in communities such as the biomedical ontology community.

IV. Conclusions

In this paper we have presented logic-based test-driven development (TDD) for ontology engineering. We have presented tools that support the approach in the form of the OntoDebug plug-in for Protégé. TDD is driven by test cases defined by an ontology engineer as a specification for an intended part of the ontology. These test cases are then used to verify that the subsequently developed ontology corresponds to the intended one. As a result, TDD encourages iterations of the development process such that in every iteration the developer is focused on a *small* part of the intended ontology. Moreover, the development process helps to ensure that a new iteration can only be started if the developer has first specified the required parts of the intended ontology as test cases. The debugging support provided by OntoDebug in Protégé ensures simple and efficient localization of faults by having the ontology engineer answer a sequence of relevant questions. The knowledge about the intended ontology obtained during a question and answering session is converted into further test cases, thus, further assisting ontology developers as they strive for the production of high quality ontologies.

References

[1] K. Beck, *Test-driven development: by example*. Addison-Wesley, 2003.

[2] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries *et al.*, "Manifesto for agile software development," 2001.

[3] H. D. Benington, "Production of large computer programs," *IEEE Ann. Hist. Comput*, vol. 5, no. 4, pp. 350–361, 1983.

[4] O. Corcho, C. Roussey, L. Manuel, V. Blázquez, I. Pérez, U. D. Lyon, U. Lyon, L. Umr, and A. Landais, "Pattern-based OWL Ontology Debugging Guidelines," in *ISWC*, 2009, pp. 68–82.

[5] K. Davies, C. M. Keet, and A. Lawrynowicz, "TDDonto2: A test-driven development plugin for arbitrary tbox and abox axioms," in *ESWC (Satellite Events)*, 2017, pp. 120–125.

[6] K. Forsberg and H. Mooz, "The relationship of system engineering to the project cycle," in *INCOSE International Symposium*, vol. 1, no. 1, 1991, pp. 57–65.

[7] G. Friedrich and K. Shchekotykhin, "A General Diagnosis Method for Ontologies," in *ISWC*, 2005, pp. 232–246.

[8] S. García-Ramos, A. Otero, and M. Fernández-López, "OntologyTest: A tool to evaluate ontologies through tests defined by the user," in *IWANN (2)*, 2009, pp. 91–98.

[9] B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. Patel-Schneider, and U. Sattler, "OWL 2: The next step for OWL," *J. Web Semant.*, vol. 6, no. 4, pp. 309–322, 2008.

[10] M. Horridge, B. Parsia, and U. Sattler, "Laconic and Precise Justifications in OWL," *ISWC*, pp. 323–338, 2008.

[11] P. N. Johnson-Laird, "Deductive reasoning," *Annu. Rev. Psychol*, vol. 50, pp. 109–135, 1999.

[12] N. Matentzoglu, M. Vigo, C. Jay, and R. Stevens, "Making entailment set changes explicit improves the understanding of consequences of ontology authoring actions," in *EKAW*, 2016, pp. 432–446.

[13] B. Motik, R. Shearer, and I. Horrocks, "Hypertableau Reasoning for Description Logics," *JAIR*, vol. 36, pp. 165–228, 2009.

[14] A. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, and C. Wroe, "OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns," in *EKAW*, 2004, pp. 63–81.

[15] A. Rector, L. Iannone, and R. Stevens, "Quality assurance of the content of a large dl-based terminology using mixed lexical and semantic criteria: experience with snomed ct," in *K-CAP*, 2011, pp. 57–64.

[16] A. L. Rector, S. Brandt, and T. Schneider, "Getting the foot out of the pelvis: modeling problems affecting use of SNOMED CT hierarchies in practical applications," *JAMIA*, vol. 18, no. 4, pp. 432–440, 2011.

[17] P. Rodler, "Interactive Debugging of Knowledge Bases," Ph.D. dissertation, Alpen-Adria Universität Klagenfurt, 2015. [Online]. Available: http://arxiv.org/pdf/1605.05950v1.pdf

[18] ——, "On active learning strategies for sequential diagnosis," in *DX Workshop*, vol. 4, 2018, pp. 264–283.

[19] P. Rodler and W. Schmid, "On the impact and proper use of heuristics in test-driven ontology debugging," in *RuleML+RR*, 2018, pp. 164–184.

[20] P. Rodler, W. Schmid, and K. Schekotihin, "Inexpensive cost-optimized measurement proposal for sequential model-based diagnosis," in *DX Workshop*, 2018, pp. 200–218.

[21] P. Rodler, K. Shchekotykhin, P. Fleiss, and G. Friedrich, "RIO: Minimizing User Interaction in Ontology Debugging," in *RR*, 2013, pp. 153–167.

[22] C. Roussey, O. Corcho, and L. M. Vilches-Blázquez, "A catalogue of OWL ontology antipatterns," in *K-CAP*, 2009, pp. 205–206.

[23] K. Schekotihin, P. Rodler, and W. Schmid, "Ontodebug: Interactive ontology debugging plug-in for protégé," in *FoIKS*, 2018, pp. 340–359.

[24] K. Shchekotykhin, G. Friedrich, P. Fleiss, and P. Rodler, "Interactive ontology debugging: Two query strategies for efficient fault localization," *J. Web Semant.*, vol. 12, pp. 88–103, 2012.

[25] K. Shchekotykhin, G. Friedrich, P. Rodler, and P. Fleiss, "Sequential diagnosis of high cardinality faults in knowledge-bases by direct diagnosis generation," in *ECAI*, 2014, pp. 813–818.

[26] K. Shchekotykhin, D. Jannach, and T. Schmitz, "MergeXplain: Fast computation of multiple conflicts for diagnosis," in *IJCAI*, 2015, pp. 3221–3228.

[27] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical OWL-DL reasoner," *J. Web Semant.*, vol. 5, no. 2, pp. 51–53, 2007.

[28] D. Vrandecic and A. Gangemi, "Unit tests for ontologies," in *OTM Workshops (2)*, 2006, pp. 1012–1020.

[29] J. D. Warrender and P. Lord, "How, what and why to test an ontology," *CoRR*, vol. abs/1505.04112, 2015.