An Ontology Debugger for the Semantic Wiki KnowWE
(Tool Presentation)

Sebastian Furth[1] and Joachim Baumeister[1,2]

[1] denkbares GmbH, Friedrich-Bergius-Ring 15, 97076 Würzburg, Germany
{firstname.lastname}@denkbares.com
[2] University of Würzburg, Institute of Computer Science, Am Hubland, 97074 Würzburg, Germany

Abstract. KnowWE is a semantic wiki that provides the possibility to define and maintain ontologies and strong problem-solving knowledge. Recently, the ontology engineering capabilities of KnowWE were significantly extended. As with other ontology engineering tools, the support of ontology debugging during the development of ontologies is the deficient. We present an Ontology Debugger for KnowWE that is based on the delta debugging approach known from Software Engineering. KnowWE already provides possibilities to define test cases to be used with various knowledge representations. While reusing the existing testing capabilities we implemented a debugger that is able to identify failure-inducing statements between two revisions of an ontology.

1 Introduction

In software engineering changing requirements and evolving solutions are well-known challenges during the software development process. Agile software development [3] became popular as it tackles these challenges by supporting software engineers with methods based on iterative and incremental development.

In the field of ontology engineering the challenges are similar. It is extremely rare that the development of an ontology is a one-time task. In most cases an ontology is developed continuously and collaboratively. Even though this insight is not new and tool support has improved in recent years, a mature method for agile ontology development still is a vision.

The idea of continuous integration (CI) has been adapted for the development of knowledge systems (cf. Baumeister et al. [1] or Skaf-Molli et al. [14]). CI for knowledge system uses automated integration tests in order to validate a set of modifications. In software engineering the continuous integration is applied by unit and integration tests to mostly manageable sets of changes, which often is sufficient to isolate bugs. In accordance to Vrandečić et al. [16], who adapted the idea of unit testing to ontologies, we consider unit tests for ontologies to be difficult to realize. Additionally in ontology engineering changes can be rather complex, e.g. when large amounts of new instance data is extracted from texts and added automatically to an ontology.

As abandoning a complete change set because of an error is as unrealistic as tracing down the failure cause manually, a method for isolating the fault automatically is necessary. We developed a debugging plugin for the semantic wiki KnowWE that is able to find the failure-inducing parts in a change set. The debugger is based on the Delta Debugging idea for software development proposed by Zeller [18].

The remainder of this paper is structured as follows: Section 2 describes the delta debugging approach for ontologies; in Section 3 we present the developed plugin for KnowWE in detail. Section 4 contains a short case study while Section 5 concludes with a discussion and the description of future work.

2 Delta Debugging for Ontologies

2.1 Prerequisites

The proposed Delta Debugging approach assumes that we are able to access different revisions of an ontology. Additionally, we assume that a mechanism for the detection of changes exists. We call the difference between two revisions the set of changes C. The detection can be realized for example by utilizing revision control systems like SVN or Git or by manually calculating the difference between two snapshots of an ontology. Even more sophisticated change detection approaches are possible, like the one proposed by Goncalves et al. [5] for the detection of changes in OWL ontologies on a semantic level.

Definition 1 (Changes). *Let $C = \{c_1, c_2, \ldots, c_n\}$ be the set of changes c_i provided by a change detection mechanism.*

Definition 2 (Revision). *Let O_1 be the base revision of an ontology and $O_2 = O_1 \cup C$ a revision of the ontology, where the set of changes C have been applied.*

With respect to a given test one of the revisions has to pass while the other one has to fail a specific test (Axiom 1).

Definition 3 (Test Function). *A function $test : O \rightarrow BOOLEAN$ determines for a given ontology whether it passes ($TRUE$) a specified test procedure or not ($FALSE$).*

Axiom 1 *For a given test function test and the revisions O_1 and O_2 of the ontology $test(O_1) = TRUE$ and $test(O_2) = FALSE$ holds.*

2.2 Tests for Ontologies

Test outcome The Delta Debugging approach we propose is not limited to a certain test procedure as long as it can assert a boolean value as defined in Definition 3 and Axiom 1. In software engineering the outcome of a test can also be undefined. Zeller [18] pointed out three reasons why this can happen:

Failure 1 *Integration: When a change relies on earlier changes that are not included in the currently focused change set, the change may not be applied.*

Failure 2 *Construction: When applying all changes a program may have syntactical or semantical errors which avoids the construction of the program.*

Failure 3 *Execution: A program can not be executed.*

As ontology engineering is basically about adding/removing/changing triples, these failures can hardly occur—at least on the syntactical level. Incorrect statements can usually not be added to a repository and therefore should not detected as a valid/applicable change by the change detection mechanism. Additionally triples do syntactically not depend on other triples and therefore can be added to and removed from a repository independently. Finally ontologies are not executed in the way a program is, what relaxes the execution failure. On the semantical level, however, integration and construction failures are very likely to occur but they do not result in an undefined test outcome, but a failing test—which is the desired behavior.

Example tests A test could consider for example the result of a SPARQL query. A concrete implementation could compare the actual query result with an explicitly defined (expected) result. Another realization could use SPARQL's ASK form.

When dealing with an ontology that makes heavy use of semantics like OWL, a reasoner like Pellet [13] could be utilized in a test to check whether an ontology is consistent and/or satisfiable.

A test does not even have to test the ontology itself, as in task-based ontology evaluation [8] the outcome of the ontology's target application could be considered. Testing with sample queries for semantic search applications is an example, where a given ontology is expected to provide certain results in an otherwise unchanged semantic search engine.

Regardless of the actual implementation the definition of test cases should be a substantial and integral part of the underlying ontology engineering methodology. We described the TELESUP project [4] that aims for a methodology and tool for ontology development in a self-improving manner and emphasizes on the early formulation of test cases.

2.3 The Delta Debugging Algorithm

We propose a delta debugging algorithm (Algorithm 1) for ontologies that is basically a divide-and-conquer algorithm recursively tracing down the faulty parts of an ontology.

The input of the recursive algorithm is the base revision of the ontology O_1 that is known to pass the specified test procedure *test*. Additionally the set of changes C between this base revision and the failing revision O_2 is provided.

Algorithm 1 The delta debugging algorithm for ontologies.

```
function DeltaDebug(O_1, C, test)
    if C.length is 1 then
        return C
    end if
    r ← {}
    for all c_i in Divide(C) do
        O_t ← O_1 ∪ c_i
        if test(O_t) is FALSE then
            r ← r + DeltaDebug(O_1, c_i, test)
        end if
    end for
    return r
end function
```

If the considered change set only contains one change then this is the failure-inducing change by definition. Otherwise the helper function DIVIDE slices the set of changes in i new change sets. The function may use heuristics or exploit the semantics of the ontology to divide the initial change set. In the following each change set c_i proposed by the DIVIDE function is applied to the base revision O_1 of the ontology. If the resulting revision of the ontology O_t does not pass the specified test procedure, then the change set is recursively examined in order to find the triple responsible for the failure. As more than one recursive call of the DELTADEBUG algorithm can return a non empty set of failure inducing changes, the final result may contain more than one triple.

The shown version of the algorithm returns all changes that applied to the base revision O_1 cause the failure. It additionally assumes monotonicity, i.e. a failure occurs as long as the responsible changes are contained in the change set. A more sophisticated handling of interferences will be subject of future work.

3 Implementation

3.1 KnowWE

We have implemented the delta debugging algorithm as an extension of the semantic wiki KnowWE [2]. KnowWE provides the possibility to define and maintain ontologies together with strong problem-solving knowledge. Ontologies can be formulated using the RDF(S) or OWL languages. KnowWE provides different markups for including RDF(S) and OWL: proprietary markups, turtle syntax, and the Manchester syntax. KnowWE compiles ontologies incrementally, i.e. only those parts of an ontology get updated that are affected by a specific change. This is possible as KnowWE's incremental parsing and compiling mechanism is able to keep track of which markup is responsible for the inclusion of a specific statement. Thus statements can easily be added to or removed from the repository when a specific markup has been changed.

3.2 Change Detection Mechanism

We use a dedicated change log to keep track of all changes applied to an ontology in KnowWE. Each time a change is applied to the repository KnowWE's event mechanism is used to fire events that inform about the statements that have been added to and removed from the repository. For every change a log entry is created. Listing 1.1 shows an example of log entries, that indicate the removal (line 1) and addition (line 2) of statements at the specified timestamps.

Listing 1.1. Example change log

```
1  -;1401619927398;si:abraham;rdfs:label;Abraham Simpson
2  +;1401619927401;si:abraham;rdfs:label;Abraham Simson
```

The change detection mechanism can now be realized by accessing the log file and asking for the changes between two points in time. The ontology revisions O_1 and O_2[3] can be constructed by reverting all changes between a specified start point and the currently running revision of the ontology (HEAD). The set of changes C between these two revisions can be extracted directly from the log file.

3.3 Tests in KnowWE

In order to realize the *test* function, we have introduced a Java interface called `OntologyDeltaDebuggerTest` which requires implementors to realize the method `boolean execute(Collection<Statement> statements)`.

We have implemented a sample test that checks whether a revision of an ontology is able to provide specified results for a SPARQL query. We exploit the already existing possibilities of KnowWE to formulate and execute labeled SPARQL queries. In the following, we use an exemplary ontology inspired by the comic characters "The Simpsons"[4].

Listing 1.2. Example for an expected SPARQL result.

```
1  %%SPARQL
2  SELECT ?s
3  WHERE {
4    ?s rdf:type si:Human;
5       si:gender si:male;
6       rdfs:label ?name .
7    FILTER regex(str(?name), "Simpson")
8  }
9  @name: maleSimpsons
10 %

12 %%ExpectedSparqlResult
13 |si:abraham
14 |si:homer
15 |si:bart
16 @sparql: maleSimpsons
17 @name: maleSimpsonsExpected
18 %
```

[3] The revision O_2 is constructed to check whether Axiom 1 holds.

[4] http://en.wikipedia.org/wiki/The_Simpsons

Additionally we use KnowWE's feature to define expected results for a specified query. This can be done by adding the expected results to a table and referencing a labeled SPARQL query. For the convenient formulation a special markup has been introduced. Listing 1.2 shows an example where `si:homer` and `si:bart` are the expected results of the SPARQL query with the label "maleSimpsons". In order to access the formulated expected results, the markup also gets a label ("maleSimpsonsExpected"). The actual test is instantiated using this label, which allows accessing the expected results as well as the underlying SPARQL query.

3.4 Ontology Debugger

The Delta Debugger for Ontologies is realized by the markup `OntologyDebugger` that allows for the convenient configuration of the debugger. The configuration is done by specifying the base revision O_1 using the `start` annotation, optionally the revision O_2 can be specified using the `end` annotation. If not specified the current revision of the ontology (HEAD) is considered as O_2. Using the annotation `expected` the label of the expected SPARQL result is defined.

Listing 1.3. Example for the definition of an Ontology Debugger.

```
1  %%OntologyDebugger
2    @expected: maleSimpsonsExpected
3    @start: 1401267947599
4  %
```

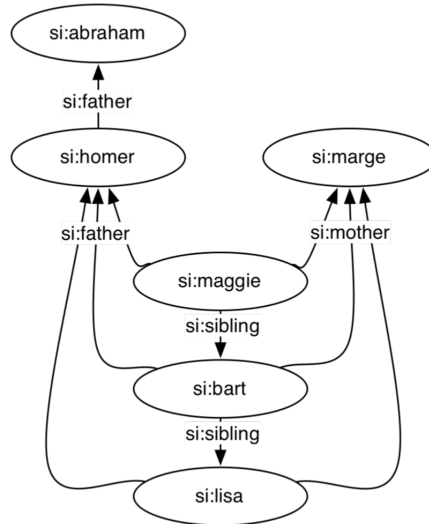
The so defined ontology debugger instance is rendered like depicted in Figure 1. A tool menu allows the execution of the debugger, a progress bar is used to visualize the running process. The actual implementation of the delta debugging algorithm for ontologies has been realized as `LongOperation` that is a feature of KnowWE's framework architecture, which allows for executing long operations in background without having the user to wait for the result. When the long operation has finished, then the failure-inducing changes are returned and displayed. An error message is rendered instead, if a failure occurs during the execution of the debugger, e.g. because the test is undefined or Axiom 1 does not hold for the specified revisions.

Fig. 1. The ontology debugger in KnowWE.

4 Case Study

4.1 The Simpsons Ontology

Fig. 2. An excerpt of the ontology showing the relationships of the Simpson family.



In the following we describe a small case study that illustrates the functionality of the presented ontology debugging extension for KnowWE. Therefore we use an example ontology that has been developed for tutorial purposes and covers various facts of the popular comic television series "The Simpsons". The ontology contains several classes like `Human` or `Building`, as well as properties like `parent` or `owns`. Additionally, some instances of the defined classes are included. We do not present the entire ontology but concentrate on some relevant parts.

Figure 2 shows relationships of the Simpsons family, e.g. that Homer (`si:homer`) is the father (`si:father`) of Bart (`si:bart`), Lisa (`si:lisa`) and Maggie (`si:maggie`), who are also marked as siblings (`si:sibling`). The sibling property was initially defined as `owl:TransitiveProperty`, i.e. a triple that explicitly states that Lisa is sibling of Maggie is not necessary. We have also defined that `si:father` is a sub-property of `si:parent`, which has an inverse property `si:child`. Listing 1.4 describes a SPARQL query for all children of Homer (`si:homer`) and Marge (`si:marge`).

Listing 1.4. SPARQL query for the children of Homer and Marge.

```
1  %%SPARQL
2  SELECT ?kid
3  WHERE {
4    ?kid rdf:type si:Human .
5    si:homer si:child ?kid .
6    si:marge si:child ?kid .
7  }
8  @name: simpsonsKids
9  %
```

An expert on the Simpson family knows that Bart, Lisa and Maggie are the expected result of this query. So this knowledge can be defined in KnowWE as an expected SPARQL result (Listing 1.5), which than can be used as a test case for the ontology.

Listing 1.5. Expected results of the query for the children of Homer and Marge.

```
1  %%ExpectedSparqlResult
2  |si:maggie
3  |si:bart
4  |si:lisa
5  @name: simpsonsKidsExpected
6  @sparql: simpsonsKids
7  %
```

Listing 1.6 is another example containing a SPARQL query for all siblings of Maggie (`si:maggie`) and the definition of the expected result (`si:bart` and `si:lisa`).

Listing 1.6. A test case for Maggie's siblings.

```
1  %%SPARQL
2  SELECT ?sibling
3  WHERE {
4    BIND (si:maggie as ?kid) .
5    ?kid si:sibling ?sibling .
6    FILTER (?kid != ?sibling) .
7  }
8  @name: maggiesSiblings
9  %

11 %%ExpectedSparqlResult
12 |si:bart
13 |si:lisa
14 @name: maggiesSiblingsExcpected
15 @sparql: maggiesSiblings
16 %
```

For this case study various changes have been applied to the ontology (see Listing 1.7) and broke it finally, i.e. the SPARQL results do not return the expected results: Bart can not be retrieved as sibling of Maggie, and apparently Homer and Marge do not have any children.

Listing 1.7. Changes applied to the Simpsons ontology.

```
1  -;1401267947600;si:snowball;rdfs:label;Snowball
2  +;1401267947605;si:santas_little_helper;rdfs:label;Santa's little helper@en
3  +;1401267947605;si:snowball;rdfs:label;Snowball II
4  -;1401268045755;si:child;owl:inverseOf;si:parent
5  +;1401283675264;si:sibling;rdf:type;owl:IrreflexiveProperty
6  -;1401283841549;si:relatedWith;rdf:type;owl:ReflexiveProperty
7  +;1401283841552;si:relatedWith;rdf:type;owl:SymmetricProperty
8  -;1401283907308;si:sibling;rdf:type;owl:TransitiveProperty
9  -;1401287487640;si:Powerplant;rdfs:subClassOf;si:Building
```

In order to find the failure-inducing changes, we have defined two ontology debugger instances that utilize the test cases defined above. Listing 1.8 shows their definitions. Revision 1401267947599 is the base revision O_1 for both instances as we know that the queries had been working before we started changing the ontology.

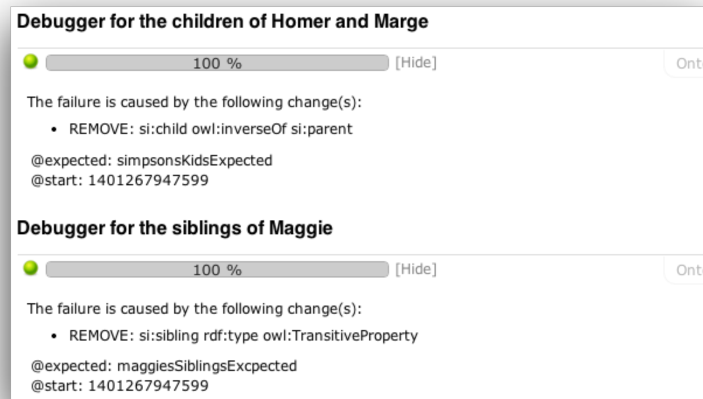
Listing 1.8. Changes applied to the Simpsons ontology.

```
1  %%OntologyDebugger
2    @expected: simpsonsKidsExpected
3    @start: 1401267947599
4  %

6  %%OntologyDebugger
7    @expected: maggiesSiblingsExcpected
8    @start: 1401267947599
9  %
```

After manually triggering the debugging process, the ontology debugger instances return the correct results. As depicted in Figure 3 the first ontology debugger instance identified that removing the statement declaring `si:child` as inverse property of `si:parent` has caused the failure that the children of Homer and Marge could not be retrieved. The second instance reports that Bart is not identified as sibling of Maggie because the transitivity (`owl:TransitiveProperty`) has been removed from the `si:sibling` property.

Fig. 3. The result of running the ontology debugger.



We ran the case study on an Apple MacBook with 3 GHz Intel Core i7 processor and 8 GB RAM; the example ontology contained 2,518 triples. The ontology debugger returned each result after about 1.2 seconds on.

4.2 Ontology of an Industrial Information System

The ontology debugger was also utilized to trace down a failure-inducing change in an ontology for an industrial information system. The ontology comprises more than 700,000 triples and makes heavy use of OWL2-RL semantics. In this scenario the debugger returned the correct hint for the failure-inducing change after 5 minutes. The manual tracing of the error would have costed many times over the presented automated approach.

5 Conclusion

In this paper we presented an extension for KnowWE that adds support for ontology debugging by using a divide-and-conquer algorithm to find failure-inducing changes. Our current implementation is working on the syntactical level of an ontology and uses a provided test case in combination with a change detection mechanism and a heuristic for dividing the change set. However, the design of the algorithm and the software allows for the incorporation of more sophisticated methods that may consider semantics to leverage the debugging to a semantic level.

There has already been work on considering semantics for the debugging of ontologies. Schlobach et al. [10] coined the term *pinpointing* which means reducing a logically incorrect ontology, s.t. a modeling error could be more easily detected by a human expert. They also proposed algorithms that use pinpointing to support the debugging task [11]. Ribeiro et al. [9] proposed the usage of Belief Revision to identify axioms in OWL ontologies that are responsible for inconsistencies. Wang et al. [17] proposed a heuristic approach that considers OWL semantics in order to explain why classes are unsatisfiable. Shchekotykhin et al. [12] proposed an interactive debugging approach to address the problem that in OWL ontologies more than one explaination for an error can exist and additional information is necessary to narrow down the problem. As debugging OWL ontologies is closely related to the justification of entailments the work in this field must also be considered. See for example Horridge et al. [6] or Kalyanpur et al. [7]. However, Stuckenschmidt [15] questions the practical applicability of several debugging approaches of OWL ontologies with respect to scalability and correctness.

While already functional we consider the current implementation of our ontology debugging plugin for KnowWE as early work. For the future we plan several enhancements to the debugger, like the replacement of the change log by a change detection mechanism that is based on standard wiki functionality providing direct access to different revisions of an ontology. As mentioned above we want to improve the handling of interferences, check the monotonicity assumption for different ontology languages and plan to examine the applicability of OWL debugging approaches. As KnowWE also allows for the definition of strong problem-solving knowledge the generalization of the debugger to other knowledge representations will be subject of future work.

Acknowledgments

The work described in this paper is supported by the Bundesministerium für Wirtschaft und Energie (BMWi) under the grant ZIM KF2959902BZ4 "SELE-SUP – SElf-LEarning SUPport Systems".

References

1. Baumeister, J., Reutelshoefer, J.: Developing knowledge systems with continuous integration. In: Proceedings of the 11th International Conference on Knowledge Management and Knowledge Technologies. p. 33. ACM (2011)
2. Baumeister, J., Reutelshoefer, J., Puppe, F.: KnowWE: a Semantic Wiki for knowledge engineering. Applied Intelligence 35(3), 323–344 (2011)
3. Cockburn, A.: Agile Software Development (2002)
4. Furth, S., Baumeister, J.: TELESUP Textual Self-Learning Support Systems. In: under review (2014)
5. Gonçalves, R.S., Parsia, B., Sattler, U.: Analysing the evolution of the NCI thesaurus. In: Computer-Based Medical Systems (CBMS), 2011 24th International Symposium on. pp. 1–6. IEEE (2011)
6. Horridge, M., Parsia, B., Sattler, U.: Laconic and precise justifications in OWL. In: The Semantic Web-ISWC 2008, pp. 323–338. Springer (2008)
7. Kalyanpur, A., Parsia, B., Horridge, M., Sirin, E.: Finding all justifications of OWL DL entailments. In: The Semantic Web, pp. 267–280. Springer (2007)
8. Porzel, R., Malaka, R.: A task-based approach for ontology evaluation. In: ECAI Workshop on Ontology Learning and Population, Valencia, Spain. Citeseer (2004)
9. Ribeiro, M.M., Wassermann, R.: Base revision for ontology debugging. Journal of Logic and Computation 19(5), 721–743 (2009)
10. Schlobach, S., Cornet, R.: Non-standard reasoning services for the debugging of description logic terminologies. In: IJCAI. vol. 3, pp. 355–362 (2003)
11. Schlobach, S., Huang, Z., Cornet, R., Van Harmelen, F.: Debugging incoherent terminologies. Journal of Automated Reasoning 39(3), 317–349 (2007)
12. Shchekotykhin, K., Friedrich, G., Fleiss, P., Rodler, P.: Interactive ontology debugging: two query strategies for efficient fault localization. Web Semantics: Science, Services and Agents on the World Wide Web 12, 88–103 (2012)
13. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semantics: science, services and agents on the World Wide Web 5(2), 51–53 (2007)
14. Skaf-Molli, H., Desmontils, E., Nauer, E., Canals, G., Cordier, A., Lefevre, M., Molli, P., Toussaint, Y.: Knowledge continuous integration process (k-cip). In: Proceedings of the 21st international conference companion on World Wide Web. pp. 1075–1082. ACM (2012)
15. Stuckenschmidt, H.: Debugging OWL Ontologies-A Reality Check. In: EON (2008)
16. Vrandečić, D., Gangemi, A.: Unit tests for ontologies. In: On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops. pp. 1012–1020. Springer (2006)
17. Wang, H., Horridge, M., Rector, A., Drummond, N., Seidenberg, J.: Debugging OWL-DL ontologies: A heuristic approach. In: The Semantic Web–ISWC 2005, pp. 745–757. Springer (2005)
18. Zeller, A.: Yesterday, my program worked. Today, it does not. Why? In: Software EngineeringESEC/FSE99. pp. 253–267. Springer (1999)