Integrating Ontology Debugging and Matching into the eXtreme Design Methodology

Zlatan Dragisic, Patrick Lambrix and Eva Blomqvist

Conference Publication

N.B.: When citing this work, cite the original article.

Original Publication:

Zlatan Dragisic, Patrick Lambrix and Eva Blomqvist, Integrating Ontology Debugging and Matching into the eXtreme Design Methodology, Proceedings of the 6th Workshop on Ontology and Semantic Web Patterns (WOP 2015), 2015
Copyright: The authors
http://ceur-ws.org/

Postprint available at: Linköping University Electronic Press

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-121794

Integrating Ontology Debugging and Matching into the eXtreme Design Methodology

Zlatan Dragisic[1,2], Patrick Lambrix[1,2], Eva Blomqvist[1]

(1) Department of Computer and Information Science,
(2) Swedish e-Science Research Centre
Linköping University, 581 83 Linköping, Sweden

Abstract. Ontology design patterns (ODPs) and related ontology development methodologies were designed as ways of sharing and reusing best practices in ontology engineering. However, while the use of these reduces the number of issues in the resulting ontologies defects can still be introduced into the ontology due to improper use or misinterpretation of the patterns. Thus, the quality of the developed ontologies is still a major concern. In this paper we address this issue by describing how ontology debugging and matching can be integrated in a state-of-the-art ontology development methodology based on ontology design patterns - the eXtreme Design methodology, and show the advantages in a case study based on a real world ontology.

1 Introduction

Ontology engineering deals with the ontology development process, the ontology life cycle, and the methodologies, tools and languages for building ontologies. While most of the ontology engineering methodologies define a step that concerns quality assurance of the developed ontologies, the description of how to do this is usually in the form of high-level tasks that need to be executed, or checklists that have to be manually ticked off. The tasks usually lack specific details on how to execute them, and require considerable user intervention, i.e., there is usually only limited tools support.

Ontology Design Patterns (ODPs) have emerged as a way to share and reuse best practices in ontology engineering, hence, ODPs could be viewed as one way of reducing quality issues in ontologies already at the design stage. However, previous studies, e.g. [1], have shown that although the use of ODPs (and related methodologies, such as eXtreme Design(XD)) significantly reduces the number of issues in the resulting ontologies, alone they cannot solve all the problems. ODPs can be misinterpreted, applied in an inappropriate way, or integrated into the overall ontology in an inappropriate manner, or the ontology engineer may simply make mistakes in modelling the parts not covered by an ODP.

During the recent years the detection and repair of defects in ontologies has been an active research area and a number of methods and tools were developed for dealing with different types of defects. We argue that these methods and tools should be a part of ontology engineering methodologies, thus aiding the

ontology quality assurance process, which in turn may result in major efficiency and effectiveness gains. Therefore the aim of this paper is to show how such methods can improve upon the results of a pattern-based methodology, such as XD, although the results are also transferable to other methodologies.

The paper is organized as follows: Section 2 gives an overview of state-of-the-art in ontology engineering and ontology debugging and matching. In Section 3 we describe the XD methodology while Section 4 discusses how ontology debugging and matching can be integrated into the methodology. In Section 5 we show the advantages of this approach in a case study based on a real world example. Finally, Section 6 gives concluding remarks and some ideas for future work.

2 Related work

Ontology engineering methodologies have been developed based on experiences in different areas, and adopt different strategies for addressing the ontology engineering process [3, 29]. Some of the more recent methodologies have focused particularly on iterative and collaborative development, e.g., DILIGENT [23], or unifying several different potential development strategies, such as the NeOn ontology engineering framework [30]. There are also methodologies that specifically target modular ontology development (e.g. [22], and XD [25]).

Semantically-enabled applications need high quality ontologies and one of the basic requirements for high quality is the lack of defects. During the recent years, in the ontology debugging and matching areas, the detection and repairing of defects has received a lot of attention. Systems and approaches such as the ones in [4, 14, 24] allow for detecting defects using different kinds of patterns. Defects in ontologies can be divided into syntactic, semantic and modeling defects [13]. Syntactic defects represent syntactic errors such as incorrect format or missing tags, semantic defects represent problems withing the logic in the ontology while modelling defects are defects which are a result of modelling erros, such as wrong and missing relations. There is a lot of work on resolving semantic defects, where the focus has been on coherence and consistency of ontologies (e.g., [13, 26–28]) and ontology networks (e.g., [8, 10, 12, 21, 31]). Recent work on repairing modelling defects has addressed the completion of the is-a structure of [15–17, 32] and the mappings between ontologies (e.g. overviews in [5]).

However, the areas mentioned above, i.e., ontology engineering methodologies and ontology debugging and matching, have so far not been very well integrated, although debugging could be viewed as an obvious activity in more or less any of the above mentioned methodologies. For instance, the study in [29] found that almost all the larger projects used a methodology, but that there was a lack of tools support for many of the steps in the methodologies. Further, despite the fact that quality of the developed ontologies is a major concern, most projects only did minor testing, and tools for aiding in the ontology quality assurance were commonly lacking. Most of the earlier ontology engineering methodologies only describe quality assurance and evaluation of the resulting ontology as an important aspect, without going into details on how this should be done. When

testing and quality assurance is mentioned it is often restricted to formal consistency and coherence checking, using a reasoner, or manually going through checklists for different aspects of the ontologies. In more recent methodologies, such as XD [25], testing and debugging is discussed in more detail, however, still only a few guidelines have been given [2], with basic tool support, and the repair process is left completely up to the ontology engineer.

In summary, the main novelty and contribution of this paper compared to related work is to showcase the integration of the ontology engineering and the ontology debugging and matching fields by integrating debugging and matching into a modern agile ontology engineering methodology. Although these may have been implicitly used in many ontology engineering projects, to the best of our knowledge this is the first attempt to explicitly specify how to perform ontology debugging and matching as an integrated part of an ontology engineering methodology.

3 The XD Methodology

In order to specifically support the use of ODPs in ontology engineering, pattern-based methodologies have emerged. One such methodology is eXtreme Design (XD) [25]. XD builds on the idea that ODPs, and in particular so-called Content ODPs, represent generalised use cases, which can be matched against the concrete use cases that represent the requirements of the ontology to be built. Content ODPs (CPs) [7] focus on concrete modelling issues and can be represented as small ontologies to be reused by importing them in the ontology under development. Once matching CPs have been found, they can be specialised and tailored for the specific case at hand. This CP specialisation process (including also potential composition of several CPs, and extensions) results in a small part of the ontology, i.e., what we here call an *ontology module*. A typical case of CP specialisation would start by creating a new (empty) ontology, representing the module to be built, import the reusable OWL building block that represents the CP to be reused, then add subclasses and subproperties of the imported classes and properties, which represent the specific pattern realisation for this particular use case. The module may also include additional classes, properties and other axioms, i.e. extensions that go beyond the CP, since CP coverage is not complete for all domains, but rather CPs only focus on the difficult modelling issues. The result is a module, i.e., a small ontology, with a locally defined part and one or more imported CPs. Such modules are then integrated to form the overall ontology, hence, XD inherently produces modular ontologies.

The ontology engineering process suggested by the XD methodology as well as the extensions proposed in this paper are depicted in Figure 1. The original process consists of a project initiation phase, whereafter the ontology is built incrementally, by adding one module at a time, each solving a small set of the overall requirements. Verification and validation of the requirements through testing is an important activity [2], but despite the fact that XD stresses testing and evaluation as a core activity, this part of the methodology has not received much

Fig. 1. Illustration of the integration of the detailed debugging and matching methods into XD. The rectangles represent activities already specified in the original version of XD (darker) and detailed activities that are proposed in this paper (lighter).

attention previously. In [2] some support for formalising requirements into unit tests, i.e., SPARQL queries, was discussed. Although such testing is important, it only determines that it is actually possible to write a query corresponding to the requirements, i.e., it does not, for instance, test whether this requires extensive knowledge of the implicit assumptions behind the model, or if all relations and axioms related to the requirements are actually made explicit in the model. In this paper the focus is on precisely these aspects, i.e. to further debug the ontology module, and the overall ontology, to find missing elements, elements that are not made explicit in the model, and also further support for analysing the axiomatic structure of the ontology.

The motivation behind focusing on the XD methodology for our study reported in this paper is that it has an obvious point of integration, since it already clearly specifies testing and evaluation activities.

4 Integrating Ontology Debugging and Matching into XD

Ontology debugging and matching techniques can be used to facilitate the process of constructing and integrating modules using the XD methodology. We integrate the debugging process in the “Test and revise” and the “Evaluate and revise” steps and the matching process in the “Integrate” step (Figure 1).

Modifying the “Test and revise” activity. The first step after developing a module and applying any unit tests, i.e., SPARQL queries over example data (c.f. [2]), is to deal with **syntactic defects**. These are the simplest type of defects and are easy to detect and resolve. They can usually be avoided by using ontology modelling tools. However, manual interventions can introduce syntactic errors into the modules. Repairing can be done using validators or spell checkers.

As identified in [20] the next step is to consider **semantic defects**. In this case, the aim is to acquire a consistent and a coherent module. These kinds of defects can be detected using reasoners. One possible reason for inconsistency is the instantiation of an unsatisfiable class. In the case of incoherence, we need to deal with unsatisfiable classes which requires removal or redefining of some of the axioms in the module. There exist a number of tools for dealing with semantic defects, such as RaDON [10], Swoop [13] and MUPSter [27].

The third phase deals with **modelling defects**. These kinds of defects require domain knowledge to detect and resolve. Two examples of modelling defects are wrong and missing relations. Wrong relations are relations which are not correct according to the domain. One way of detecting these relations is by detecting semantic defects. Other ways include using external knowledge sources or through manual inspection. In order to deal with wrong relations, we need to remove parts of the justifications of the relation so that it is no longer possible to derive the relation. Missing relations represent knowledge that is correct according to the domain but is not derivable in the module. Repairing missing relations requires adding or modifying axioms so that the detected relation becomes derivable in the module. The simplest approach is just to add the detected relation to the module, however, in [15–17, 32] it was shown that the repairing process for completing ontologies can add new knowledge which was not detected before. One of the tools for dealing with modelling defects is RepOSE (e.g. [8, 15, 19]).

Another tool for detecting semantic and modelling defects is OOPS! [24] which contains a catalogue of patterns which correspond to common pitfalls in ontology development. The tool is semi-automatic and for each detected pitfall the tool gives an explanation and a suggestion how to resolve it.

After dealing with modelling defects it is necessary to run the unit tests once more, and then check again for syntactic, semantic and modelling defects. This iterative approach is necessary as relations added in the repairing process could have introduced new defects. Therefore, the debugging process in “Test and revise” is iterative and should be repeated until no new defects are identified.

Modifying the “Integrate” activity. After developing and releasing modules, the next step in XD is to integrate the newly developed partial solution into the overall ontology. This step can be supported by ontology matching techniques (debugging through completing). Since the overall ontology is inherently modular, we should consider which modules to actually integrate, in order to achieve an appropriate level of coupling. Hence, we run ontology matching systems on pairs of modules. This can either be done on all or on selected pairs, depending on the desired coupling. After collecting the result from the ontology matching system(s), the suggestions need to be validated to see if they are correct according to the domain and if they match the intended semantics of the integrated ontology. After validating the output from the ontology matching process the ontology integration team should consider which of these potential integration points to include in the integrated ontology. There is a trade-off between keeping a loose coupling of modules, but still supporting the requirements of the overall

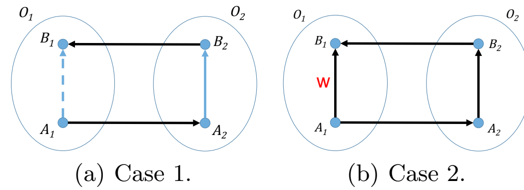
ontology, e.g., reasoning requirements, and supporting certain queries. Hence, not all of the found mappings, although found to be correct, may be chosen for inclusion in the overall integrated ontology. The integration team can also extend the set with additional relations which were identified by other means. The next step in the integration process is to create an integrated ontology containing all the modules, as well as the set of validated integrating relations.

Modifying the "Evaluate and revise" activity. The debugging process that follows integration is very similar to the one for individual modules. First, the integrated ontology needs to be tested for **syntactic defects** which in this phase can be a result of the ontology matching process. Then, the integrated ontology needs to be tested for consistency and coherence, given that some new **semantic defects** could have been introduced by integrating different modules. If inconsistencies are detected we either need to deal with unsatisfiable classes or class/property instantiations. In the case of unsatisfiable classes, axioms in certain modules should be modified or removed.

Finally, we need to deal with **modelling defects** as the integrated ontology might pinpoint some additional modelling defects, e.g. missing relations which could not be detected when considering modules on their own. We note that by repairing modelling defects only in the integrated ontology it could happen that defects are left in individual modules, which limits their re-usability. For example, let us consider Case 1 in Figure 2(a). In this case, we have two modules, O_1 and O_2, containing concepts A_1, B_1 and A_2, B_2, respectively. The integration of these modules is done through relations $B_2 \sqsubseteq B_1$ and $A_1 \sqsubseteq A_2$. The module O_1 lacks the missing relation $A_1 \sqsubseteq B_1$. Now, if we were to repair the integrated ontology first, the missing is-a relation could be made derivable if we add relation $A_2 \sqsubseteq B_2$. However, this would also mean that the missing is-a relation is not made derivable in the O_1 module and therefore is still missing in the module itself. If we repair the module first, then relation $A_2 \sqsubseteq B_2$ could be identified in the integration phase as another way of deriving $A_1 \sqsubseteq B_1$.

In addition, repairing defects in individual modules can also contribute to the integration process, hence, the two activities (i.e., "integrate" and "evaluate and revise") are inherently coupled and may be run partly in parallel. For example, let us consider Case 2 in Figure 2(b). In this case $A_1 \sqsubseteq B_1$ is present in module O_1 but is wrong according to the domain and should be removed. This knowledge could be used in the integration phase which in this case means that $B_2 \sqsubseteq B_1$ and $A_1 \sqsubseteq A_2$ should not both be added in the module integration, or this could also imply that $A_2 \sqsubseteq B_2$ is wrong. This is especially valuable in cases where the module development and the module integration are done by different teams. By documenting these kinds of cases, the module developers would help the integration team and lower the necessity for their involvement in the integration. Moreover, debugging individual modules might lower the effort needed for debugging the integrated ontology as the modules are smaller and therefore defects related to the modules are easier to detect and resolve.

As in the module development phase, the debugging process should be rerun iteratively, as some of the repairs might have introduced new defects.



(a) Case 1. (b) Case 2.

Fig. 2. Cases exemplifying the need for debugging both individual modules as well as the integrated ontology.

5 Case study

The XD methodology has been used in several real-world ontology engineering efforts, as well as research projects. For experimenting with our proposed approach we reused a set of ontology modules and their integration, which were produced by applying the XD methodology in a previous research project.

The ontologies were created in the context of the IKS project[1], for supporting one of the project use cases, which was a showcase of an intelligent bathroom environment [9]. The ontologies were intended to support seamless information integration and access to diverse information sources, for providing the bathroom environment with personalised and contextualised information served to the end user through various devices, such as screens integrated in mirrors and shower walls, audio interfaces etc. The specific showcase scenario involved a person entering the bathroom in the morning, receiving site-specific weather information, and event suggestions based on his calendar and personal preferences. The use case contains 51 modules as well as the resulting integrated ontology.[2]

Debugging individual modules. In the first phase of the debugging process we consider individual modules. The first step in debugging individual modules is to consider **syntactic defects**. We have identified in total 4 such defects in 3 modules (DeviceContentCapabilities, mpeg7, and SPDO). The detection was done by loading the ontologies using OWL API. If the API failed to load an ontology, defects were returned in the exception description. In our case, these defects were incorrect datatype definitions, i.e. the defined datatypes were not a part of OWL 2 built-in datatypes nor were they custom defined. The next step in the debugging process is to deal with **semantic defects** in modules. To identify such defects, we used the Hermit reasoner[3] to detect incoherent and inconsistent modules. In the use case, 9 modules where incoherent, while one module was inconsistent. Four out of 9 incoherent module classes contained the same

[1] EU FP-7 integrating project, targeting the integration of semantic technologies into content management systems. http://www.iks-project.eu/

[2] The original modules developed within the project use case are available from the following repository: http://www.ontologydesignpatterns.org/iks/ami/2011/02/ while the debugged modules and integrated ontology can be found at http://www.ida.liu.se/∼zladr41/AMI.zip.

[3] http://hermit-reasoner.com/

error, corresponding to the following case: $A \equiv \forall r.B$, $A \sqsubseteq C$, $Domain\ r\ is\ C$, $B \sqcap C \sqsubseteq \bot$. From the first three axioms, it follows that C is equivalent to the top class. Given this and the fourth axiom, B is unsatisfiable. These four situations were repaired by replacing the corresponding equivalence axioms with subsumption axioms. Repairing the unsatisfiable classes in these 4 modules also repaired additional 2 modules which imported the incoherent modules. Finally, the remaining 3 incoherent modules were repaired by removing a single subsumption in one of the modules. The inconsistent module (Users) was caused by three data property assertions, which specified literals with a datatype outside of the range of the data property. This was repaired by specifying a correct datatype. Further, we used OOPS! on the modules. The tool did not detect any pitfalls which could lead to inconsistencies.

The next step is to consider **modelling defects**. In this study, we only considered one type of modelling defects, i.e., missing is-a relations between named classes. In [19] it was shown that RepOSE can be used both for the detection and repairing of missing is-a structure. RepOSE accepts as input a set of existing or missing is-a relations and suggests (new) ways of deriving these is-a relations and thus possible new knowledge to add to an ontology. A domain expert needs to validate the suggestions. In this study, we have run RepOSE on every individual module where all is-a relations between named classes in a module were used as an input set. This identified additional ways of deriving existing is-a relations in 5 modules. In total, 8 new is-a relations were added. For example, in the mpeg7 module, we have added 3 new relations. Adding TemporalLocator $\sqsubseteq$ Locator and RegionLocator $\sqsubseteq$ Locator would make it possible to derive existing is-a relations TemporalLocator $\sqsubseteq$ Resource and RegionLocator $\sqsubseteq$ Resource, respectively, given that the module already contains the is-a relation Locator $\sqsubseteq$ Resource. These relations were not derivable and represent new knowledge. Further, OOPS! detected issues related to missing annotations, missing domain and range for properties as well as recursive definitions. Missing domain and range issues were related to properties in design patterns and therefore were not added to avoid overcommitment. Further inspection of recursive definitions has shown that these definitions are necessary. For example, property overlappingTopic is recursive as concept Topic is both domain and range. However, this is correct according to the domain. Another iteration of the debugging process did not detect additional defects.

In summary, our added debugging steps were able to detect and in some cases suggest repair actions for several defects in investigated modules. This shows two important things; (i) since ontologies, and even small ontology modules, are complex artefacts they usually contain defects, even if they were carefully designed by expert ontology engineers, and using ODPs, hence, performing this kind of debugging is essential for producing high-quality ontologies, and (ii) the suggested debugging methods are able to detect a wide variety of defects, ranging from incorrect datatypes to missing is-a relations, hence, by using the spectrum of existing methods we most likely cover a large part of the potential defect types.

Integration. After debugging all modules we proceeded to integrate the modules. The integration is done by running ontology matching on each pair of modules. The mappings from each of the ontology pairs are then collected and presented to the user for validation. In our experiment we have used 3 different ontology matching systems, i.e., AML [6], LogMap [11] and SAMBO [18].

For evaluating the detection of mappings, we used the actual integrated ontology from the project to compare against. Out of the validated relations, 12 of them (7 equivalence and 5 is-a relations) were already present in the integrated ontology, while 14 (6 equivalence and 8 is-a relations) represent new knowledge not present in the integrated ontology we compared against. This does not necessarily mean that all 14 should be included, some may have been intentionally left out, however, at least it shows that this approach has the potential of pointing the user towards additional options that should be considered, in order not to overlook important mappings.

Debugging the integrated ontology. We proceeded with debugging **syntactic and semantic defects** in the integrated ontology. One additional semantic defect was detected concerning MusicCollection in the MusicCollection module. This defect corresponded to the described case of unsatisfiable classes when debugging individual modules, and was repaired by replacing the equivalence axiom with a subsumption axiom. Further, OOPS! did not detect any additional pitfalls which could lead to an inconsistent ontology.

Next, we proceeded with **modelling defects**. The detected integrating relations were used as input to the process of completing the is-a structure with RepOSE. In addition, the input also included other is-a relations between named classes from the integrated ontology. In the general case where the ontology debugging is integrated into the development process it may not be necessary to consider all is-a relations but only the relations which were identified by ontology matching systems and by the ontology developers in the integration step. The input to RepOSE consisted of the integrated ontology and a set of 214 is-a relations (194 from the ontology and 20 acquired from ontology matching). In the completion process, we identified additional ways of deriving is-a relations in 8 modules. In total 10 new is-a relations were added. For example, using RepOSE we identified another way of deriving Movie $\sqsubseteq$ ContentItem where classes Movie and ContentItem are from the EntertainmentContent and Content modules respectively. In this case adding Movie $\sqsubseteq$ MultimediaContent where MultimediaContent is in the mpeg7 module would provide another way of deriving the relation. This is possible due to the fact that MultimediaContent $\sqsubseteq$ ContentItem exists in the mpeg7 module. By adding Movie $\sqsubseteq$ MultimediaContent we have also integrated the mpeg7 and EntertainmentContent modules which were not related before. OOPS! did not detect any new issues. Another iteration of the debugging process did not detect additional defects.

In summary, this shows that testing and evaluating only the individual modules of a modular ontology is not enough, hence, we need to additionally implement a similar debugging step before the release of the integrated ontology. However, it should also be noted that some defects detected after integration,

may actually depend on modelling choices made already in the module development, hence, iteratively revising the modules is inevitable.
Impact of debugging We have evaluated the impact of our changes on three levels: class level, instance level and query level.

On the class level, we identified concepts which were affected by the changes in the ontology. We looked at named classes whose subclasses or superclasses changed. For each is-a relation $A \sqsubseteq B$ that was added or removed during debugging, the subclasses of A and the superclasses of B are affected. For equivalence relations added during the debugging process, the super- and subclasses of both A and B are affected. In our case, due to addition of is-a relations, out of 197 named concepts in the integrated ontology 56 got new subclasses and/or new superclasses. Further, we have removed in total 3 is-a relations which account to additional 16 affected classes. We note that for the impact we have only considered the is-a hierarchy. Therefore, the impact of our repairs can be even greater given that we deal with expressive ontologies. For example, adding a subsumption or equivalence axiom might change the range/domain of some property.

On the instance level, we were interested in identifying the affected instances, i.e., those instances that obtained new or changed class memberships. We checked the instances of subclasses of each named class involved in the is-a or equivalence relation added or removed during the debugging process. In our case study, the integrated ontology contains 88 instances. Out of these 88, 57 instances were affected by our changes. Out of the 57 affected instances, 29 are affected by removing relations when dealing with semantic defects.

Finally, on the query level, we have used logs from the system developed during the project, which used these ontologies. The log files contain SPARQL queries that were used by the system to extract instances from the ontology during the test runs of the system in the bathroom environment. In order to evaluate the impact of our changes to the integrated ontology, we consider the number of affected queries, i.e., those queries which query for instances of affected classes. These affected queries might produce different results[4] if run over the new version of the integrated ontology, i.e., the version produced after our debugging was applied. In 1200 log files there were in total 10724 requests for data access to the integrated ontology. Out of these, 5402 are affected by our changes, i.e. they query for instances of affected classes.

6 Conclusion and Future work

In this paper we have discussed how existing approaches for debugging and matching ontologies can be used to improve the results of a pattern-based methodology. The proposed approach, specifically for integrating debugging into the XD ontology development methodology was tested in a use case study with a real world ontology. The case study has shown the usefulness of our approach, and

[4] Unfortunately, as the project ended a few years ago, we were not able to evaluate the correctness of these differences with respect to the actual results provided to the users of the system, since neither the system nor the test users are available.

pointed out a number of important benefits of performing debugging. Additionally, we have seen that there can be quite a number of defects still remaining in ontologies, even after applying both ODPs and a proper methodology when creating them, performing appropriate unit tests, such as testing SPARQL queries, and even after using the ontologies in test runs of the system itself.

There are a number of directions for future work. Rather than having to use stand-alone tools as was done in our case study, one future direction would be the integration of those tools into ontology development environments (e.g. as plug-ins). In our use case analysis we only considered relations between concepts. In the future, we will consider other existing approaches for ontology debugging and how these can be integrated into the methodology, e.g. for debugging other properties. It is also important to perform an evaluation of the added value of the changes in terms of added value for users, such as the additional retrieved (correct and relevant) data in response to a query.

Acknowledgements. We thank the Swedish e-Science Research Centre (SeRC), the Swedish National Graduate School in Computer Science (CUGS) and the EU FP7 project VALCRI (FP7-IP-608142) for financial support.

References

1. E Blomqvist, V Presutti, E Daga, and A Gangemi. Experimenting with eXtreme Design. In *Proc. of the 17th EKAW*, pages 120–134, 2010.
2. E Blomqvist, A Seil Sepour, and V Presutti. Ontology testing - methodology and tool. In *Proc. of the 18th EKAW*, pages 216–226. 2012.
3. O Corcho, M Fernandez-Lopez, and A Gomez-Perez. Methodologies, tools and languages for building ontologies. Where is their meeting point? *Data & Knowledge Engineering*, 46:41–64, 2003.
4. O Corcho, C Roussey, L M Vilches, and I Pérez. Pattern-based OWL ontology debugging guidelines. In *Proc. of the 1st WOP*, pages 68–82, 2009.
5. J Euzenat and P Shvaiko. *Ontology Matching*. Springer, 2007.
6. D Faria, C Pesquita, E Santos, M Palmonari, I F Cruz, and F M Couto. The AgreementMakerLight ontology matching system. In *On the Move to Meaningful Internet Systems: OTM 2013 Confs*, pages 527–541, 2013.
7. A Gangemi and V Presutti. Ontology Design Patterns. In *Handbook on Ontologies*. 2009.
8. V Ivanova and P Lambrix. A unified approach for aligning taxonomies and debugging taxonomies and their alignments. In *Proc. of the 10th ESWC*, pages 1–15, 2013.
9. S Janzen, E Blomqvist, A Filler, S Gönül, T Kowatsch, A Adamou, S Germesin, M Romanelli, V Presutti, C Cimen, W Maass, S Postaci, E Alpay, T Namli, and G B L Erturkmen. IKS deliverable - d4.1 report: Ami case - design and implementation (public). Technical report, 2011.
10. Q Ji, P Haase, G Qi, P Hitzler, and S Stadtmuller. RaDON - repair and diagnosis in ontology networks. In *Proc. of the 6th ESWC*, pages 863–867, 2009.
11. E Jiménez-Ruiz, B Cuenca Grau, Y Zhou, and I Horrocks. Large-scale interactive ontology matching: Algorithms and implementation. In *Proc. of the 20th ECAI*, pages 444–449, 2012.

12. E Jiménez-Ruiz, B Cuenca Grau, I Horrocks, and R Berlanga. Ontology integration using mappings: Towards getting the right logical consequences. In *Proc. of the 6th ESWC*, pages 173–187, 2009.
13. A Kalyanpur, B Parsia, E Sirin, and J Hendler. Debugging unsatisfiable classes in OWL ontologies. *Journal of Web Semantics*, 3(4):268–293, 2006.
14. M Keet. Detecting and revising flaws in OWL object property expressions. In *Proc. of the 18th EKAW*, pages 252–266, 2012.
15. P Lambrix and V Ivanova. A unified approach for debugging is-a structure and mappings in networked taxonomies. *Journal of Biomedical Semantics*, 4:10, 2013.
16. P Lambrix and Q Liu. Debugging the missing is-a structure within taxonomies networked by partial reference alignments. *Data & Knowledge Engineering*, 86:179–205, 2013.
17. P Lambrix, Q Liu, and H Tan. Repairing the Missing is-a Structure of Ontologies. In *Proc. of the 4th ASWC*, pages 76–90, 2009.
18. P Lambrix and H Tan. SAMBO - a system for aligning and merging biomedical ontologies. *Journal of Web Semantics*, 4(3):196–206, 2006.
19. P Lambrix, F Wei-Kleiner, and Z Dragisic. Completing the is-a structure in light-weight ontologies. *Journal of biomedical semantics*, 6(1):12, 2015.
20. P Lambrix, F Wei-Kleiner, Z Dragisic, and V Ivanova. Repairing missing is-a structure in ontologies is an abductive reasoning problem. In *Proc. of the 2nd WoDOOM*, pages 33–44, 2013.
21. C Meilicke, H Stuckenschmidt, and A Tamilin. Repairing ontology mappings. In *Proc. of the 20th AAAI*, pages 1408–1413, 2007.
22. T Özacar, Ö Öztürk, and MO Ünalir. ANEMONE: An environment for modular ontology development. *Data & Knowledge Engineering*, 70:504–526, 2011.
23. H S Pinto, C Tempich, and S Staab. Ontology engineering and evolution in a distributed world using diligent. In *Handbook on Ontologies*, pages 153–176. 2009.
24. M Poveda-Villalón, A Gómez-Pérez, and M C Suárez-Figueroa. Oops! (ontology pitfall scanner!): An on-line tool for ontology evaluation. *International Journal on Semantic Web & Information Systems*, 10(2):7–34, 2014.
25. V Presutti, E Bomqvist, E Daga, and A Gangemi. Pattern-based ontology design. In *Ontology Engineering in a Networked World*, pages 35–64. 2012.
26. G Qi, Q Ji, and P Haase. A Conflict-Based Operator for Mapping Revision. In *Proc. of the 8th ISWC*, pages 521–536, 2009.
27. S Schlobach and C Ronald. Non-standard reasoning services for the debugging of description logic terminologies. In *Proc. of the 18th IJCAI*, pages 355–360, 2003.
28. K Shchekotykhin, G Friedrich, Ph Fleiss, and P Rodler. Interactive ontology debugging: Two query strategies for efficient fault localization. *Journal of Web Semantics*, 12-13:88–103, 2012.
29. E Simperl, M Mochol, and T Bürger. Achieving maturity: the state of practice in ontology engineering in 2009. *Int Journal of Computer Science and Applications*, 7:45–65, 2010.
30. M C Suárez-Figueroa, A Gómez-Pérez, E Motta, and A Gangemi, editors. *Ontology Engineering in a Networked World*. Springer, 2012.
31. P Wang and B Xu. Debugging ontology mappings: a static approach. *Computing and Informatics*, 27:21–36, 2008.
32. F Wei-Kleiner, Z Dragisic, and P Lambrix. Abduction framework for repairing incomplete $\mathcal{EL}$ ontologies: Complexity results and algorithms. In *Proc. of the 28th AAAI*, pages 1120–1127, 2014.