Departamento de Inteligencia Artificial
Escuela Técnica Superior de Ingenieros Informáticos

PhD Thesis

Ontology Evaluation: a pitfall-based approach to ontology diagnosis

Author: María Poveda Villalón

Supervisors:

Prof. Dr. Asunción Gómez Pérez
Dr. María del Carmen Suárez de Figueroa Baonza

February, 2016

Tribunal nombrado por el Sr. Rector Magfco. de la Universidad Politécnica de Madrid, el día 18 de enero de 2016.

Presidente: Dr. Oscar Corcho

Vocal: Dra. Valentina Tamma

Vocal: Dr. Jesús Bermúdez de Andrés

Vocal: Dra. María Jesús Taboada Iglesias

Secretario: Dr. Mariano Fernández López

Suplente: Dra. Belén Díaz Agudo

Suplente: Dr. Dirk Walther

Realizado el acto de defensa y lectura de la Tesis el día 8 de febrero de 2016 en la Escuela Técnica Superior de Ingenieros Informáticos

Calificación: _____

EL PRESIDENTE VOCAL 1 VOCAL 2

VOCAL 3 EL SECRETARIO

A mis padres y mi hermano.

A Rober.

Agradecimientos

Ahora que por fin escribo la última página de uno de los capítulos más largos e intensos de mi vida, sólo queda recordar a todas esas personas que han hecho posible que llegue hasta aquí.

En primer lugar quiero darle las gracias a Asun por todo su apoyo y dedicación a la tesis y por confiar en mí. Gracias por todo lo que he aprendido en estos años llenos de oportunidades. Y también gracias por tantos empujoncitos para superarme una y otra vez.

Igualmente quiero agradecerle a Mari Carmen haberme llevado de la mano tantos años en los que me ha enseñado mucho más que a hacer una tesis. Gracias sobre todo por no haberme dejado tirar la toalla en ningún momento.

Quiero agradecer al gran equipo de lingüistas las largas horas de ayuda desinteresada y por animarme tanto. Gracias Julia, Elena y Lupe.

Quiero también darle las gracias a los profesores e investigadores que de una u otra manera me han ayudado a desarrollar este tesis. Gracias Oscar y Mariano F. por todos los consejos y conversaciones sobre modelado, semántica, etc. A Miriam Fernández y Vadim Ermolayev por sus revisiones y valiosos comentarios que han contribuido a mejorar esta tesis. Gracias también a Antonio Jiménez por su ayuda con las revisiones de estadística.

Quiero agradecerle a Florence Amardeilh, Bernard Vatant y Chris Bizer haberme dado la oportunidad de visitar sus centros para realizar estancias. Especialmente quiero agradecerle su hospitalidad a Valentina Tamma y haber sido en gran parte artífice de lo que tesis representa. También quiero agradecerle los buenos momentos vividos a toda la gente que he conocido durante las estancias: Ed Beamer, Jim, Maya, Chrysa, Sam, Anne, Hannes, Anja, Robert, Rafael, Laurent, Jean-Charles, Benoît...

Y ya ha llegado el momento de darle las gracias a mis compañeros de laboratorio, de desayuno, de comida, de “consejos de sabios”, de penas y alegrías,

de cervezazas y bailes: Andrés, Esther, Olga, Dani V, Filip, Raúl(es), Victor, Pablo, Carlos, Mariano, Freddy, Nandana, María Pérez, Angelito, Rafa, Jorge, Boris, Alejandros(s), JARG, Almudena. En especial quiero darle las gracias a Idafen (sí, por el logo, pero no sólo) y Dani, por aguantarme y ayudarme tanto. Habéis hecho seamos un gran equipo tanto dentro como fuera del labo. También a Miguel, por todos los buenos momentos y por supuesto, por soportarme (en los dos sentidos).

No puedo dejar de agradecer el ánimo recibido por mis amigas y amigos fuera de la universidad: Pili, Mari, Encar, Gema, Sandra, Cris, Patri, Tamara, Maribel y Rafi.

De manera muy especial quiero darle las gracias a mi familia. Sobre todo agradecerle a mis padres, Victorio y Mari Tere, que me hayan apoyado en esto, como en todo, su cariño y paciencia inagotable y su incalculable ayuda. Sencillamente, sin vosotros nada hubiera sido posible, no sólo la tesis. También a mi hermano, Javi, por todo el ánimo que me das y por seguir cuidando de mí como el primer día.

Por último quiero agradecerle a Rober que siempre haya estado a mi lado de forma incondicional, en los malos momentos y sobre todo en los peores. Gracias Rober por haber recorrido conmigo este camino, por creer en mí y apoyarme, por tu paciencia y comprensión, por no dejarme desistir en los momentos más difíciles. Gracias por ser como eres.

Abstract

Ontology evaluation, which includes ontology diagnosis and repair, is a complex activity that should be carried out in every ontology development project, because it checks for the technical quality of the ontology. However, there is an important gap between the methodological work about ontology evaluation and the tools that support such an activity. More precisely, not many approaches provide clear guidance about how to diagnose ontologies and how to repair them accordingly.

This thesis aims to advance the current state of the art of ontology evaluation, specifically in the ontology diagnosis activity. The main goals of this thesis are (a) to help ontology engineers to diagnose their ontologies in order to find common pitfalls and (b) to lessen the effort required from them by providing the suitable technological support. This thesis presents the following main contributions:

- A catalogue that describes 41 pitfalls that ontology developers might include in their ontologies.
- A quality model for ontology diagnose that aligns the pitfall catalogue to existing quality models for semantic technologies.
- The design and implementation of 48 methods for detecting 33 out of the 41 pitfalls defined in the catalogue.
- A system called OOPS! (OntOlogy Pitfall Scanner!) that allows ontology engineers to (semi)automatically diagnose their ontologies.

According to the feedback gathered and satisfaction tests carried out, the approach developed and presented in this thesis effectively helps users to increase the quality of their ontologies. At the time of writing this thesis, OOPS! has been broadly accepted by a high number of users worldwide and has been used around 3000 times from 60 different countries. OOPS! is integrated with third-party software and is locally installed in private enterprises being used both for ontology development activities and training courses.

Resumen

La evaluación de ontologías, incluyendo diagnóstico y reparación de las mismas, es una compleja actividad que debe llevarse a cabo en cualquier proyecto de desarrollo ontológico para comprobar la calidad técnica de las ontologías. Sin embargo, existe una gran brecha entre los enfoques metodológicos sobre la evaluación de ontologías y las herramientas que le dan soporte. En particular, no existen enfoques que proporcionen guías concretas sobre cómo diagnosticar y, en consecuencia, reparar ontologías.

Esta tesis pretende avanzar en el área de la evaluación de ontologías, concretamente en la actividad de diagnóstico. Los principales objetivos de esta tesis son (a) ayudar a los desarrolladores en el diagnóstico de ontologías para encontrar errores comunes y (b) facilitar dicho diagnóstico reduciendo el esfuerzo empleado proporcionando el soporte tecnológico adecuado. Esta tesis presenta las siguientes contribuciones:

- Catálogo de 41 errores comunes que los ingenieros ontológicos pueden cometer durante el desarrollo de ontologías.
- Modelo de calidad para el diagnóstico de ontologías alineando el catálogo de errores comunes con modelos de calidad existentes.
- Diseño e implementación de 48 métodos para detectar 33 de los 41 errores comunes en el catálogo.
- Soporte tecnológico OOPS!, que permite el diagnśtico de ontologías de forma (semi)automática.

De acuerdo con los comentarios recibidos y los resultados de los test de satisfacción realizados, se puede afirmar que el enfoque desarrollado y presentado en esta tesis ayuda de forma efectiva a los usuarios a mejorar la calidad de sus ontologías. OOPS! ha sido ampliamente aceptado por un gran número de usuarios de formal global y ha sido utilizado alrededor de 3000 veces desde 60 países diferentes. OOPS! se ha integrado en software desarrollado por terceros y ha sido instalado en empresas para ser utilizado tanto durante el desarrollo de ontologías como en actividades de formación.

Contents

List of Figures

List of Tables

Acronyms

API: Application Programming Interface

CQ: Competency Question

DAML+OIL: DARPA Agent Markup Language + Ontology Inference Layer

DARPA: Defense Advanced Research Projects Agency

GUI: Graphical User Interface

IDE: Integrated Development Environment

LD: Linked Data

LOD: Linked Open Data

NTK: NeOn Toolkit

OOPS!: OntOlogy Pitfall Scanner!

ORDS: Ontology Requirement Document Specification

OWL: Web Ontology Language

RDF: Resource Description Framework

RDF/XML: Resource Description Framework / eXtensible Markup Language (XML syntax for RDF)

RDFS: RDF Schema

SPARQL: SPARQL Protocol and RDF Query Language

UML: Unified Modeling Language

URI: Uniform Resource Identifier

URL: Uniform Resource Locator

WUI: Web User Interface

Chapter 1

Introduction

In computer science, the term *Ontology* refers to a "formal, explicit specification of a shared conceptualization" (Studer et al., 1998). *Conceptualization* refers to an abstract model that allows describing something relevant in the world, for which concepts, properties and constraints are normally used. All the entities in the abstract model should be explicitly described, so that the ontology covers as much as possible the world phenomenon to be represented. Being *formal* refers to the ontology being machine-readable, that is, available in some language such as the Resource Description Framework Schema (RDFS[1]) or the Web Ontology Language (OWL[2]) that can be easily processed. And finally, and most importantly, *shared* reflects the notion that an ontology captures consensual knowledge; that is, it is not private to some individuals, but accepted by a group.

The Ontology Engineering discipline (Gómez-Pérez et al., 2004) can be broadly understood as the one that works on methods, tools and techniques to facilitate the development of ontologies. This discipline has been active for more than two decades and has witnessed an important evolution during its lifetime, much of it related to the development of ontology languages, ontology engineering tools and ontology development methodologies.

The correct application of ontology development methodologies (for example Grüninger & Fox (Grüninger and Fox, 1995), METHONTOLOGY (Fernández-López et al., 1997; Fernández-López et al., 1999), On-To-Knowledge (Staab et al., 2001), DILIGENT (Pinto

[1]https://www.w3.org/TR/rdf-schema/ (last visited on the 15th January, 2016)
[2]https://www.w3.org/TR/owl-ref/ (last visited on the 15th January, 2016)

et al., 2004), or the NeOn Methodology (Suárez-Figueroa et al., 2015)) eases the ontology development process and benefits the quality of the ontology being built. The ontology quality is checked in the ontology evaluation activity, which should be carried out in any ontology development project.

It is worth remembering Guarino's reflections on ontology evaluation and its differences with to software components (literally taken from (Guarino, 2004), page 78): *Like every software product, ontologies need proper quality control to be effectively deployed in practical applications. Unfortunately, adapting them to the evaluation metrics and quality enforcement procedures developed for software engineering doesn't work. An ontology's nature is very different from that of a piece of code.(...) Indeed, ontologies are not software processes - rather, they belong to the class of data models.*

Therefore, we should address the ontology evaluation problem taking into account the particular characteristics of ontologies as models instead of treating them just as any other type of software.

The 1990s have witnessed the growing interest in the ontology evaluation activity by numerous researches who proposed prime contributions, including: (a) seminal works dealing with definitions about ontology evaluation appeared in 1995 (Gómez-Pérez et al., 1995); (b) popular methods for evaluating ontologies against a set of requirements called Competency Questions (Grüninger and Fox, 1995); and (c) approaches to common error checking in taxonomies (Gómez-Pérez, 1996, 1999).

Since then, numerous approaches in ontology evaluation have emerged. Some of these attempts define a generic quality evaluation framework (Gangemi et al., 2006; Gómez-Pérez, 2004; Vrandečić, 2010); others, such as OntoClean, (Guarino and Welty, 2009; Welty and Guarino, 2001) provide specific guidelines to assess the ontological adequacy and logical consistency of taxonomies; others propose evaluating an ontology in terms of its final reuse and use (Suárez-Figueroa, 2010; Suárez-Figueroa et al., 2013a); some others propose quality models based on features, criteria, and metrics (Burton-Jones et al., 2005); whereas others present methods for pattern-based evaluation (Djedidi and Aufaure, 2010; Presutti et al., 2008).

As a consequence of the emergence of new methods and techniques, tools for ontology evaluation have also been proposed. The first steps towards the automation of ontology evaluation started with the development of ODEClean by Fernández-López and Gómez-Pérez in 2002 (Fernández-López and Gómez-Pérez, 2002). ODEClean, a plug-in for

the WebODE ontology editor (Arpírez et al., 2003), provided technological support for OntoClean (Welty and Guarino, 2001). Later in 2004, ODEval (Corcho et al., 2004) was developed to ease the evaluation of RDFS and DAML+OIL taxonomies. Several years later, automation in ontology evaluation re-emerged with the appearance of systems such as AEON (Völker et al., 2008), XD-Analyzer (Daga et al., 2010), among others.

Despite the vast amounts of frameworks, criteria, and methods, ontology evaluation is still largely neglected by developers and practitioners. Consequently, many ontologies are evaluated using only few evaluation techniques, as for instance syntax checking or simple reasoning tests. This situation might stem from a combination between (a) the time-consuming and tedious nature of evaluating the quality and (re)usability of an ontology, and (b) the lack of awareness of the need for ontology evaluation.

Therefore, ontology developers do not need just another method, framework or set of criteria to evaluate ontologies, but rather a simple and lightweight approach supported by an engaging application that lightens the tedious task of ontology evaluation. This thesis does not aim at reinventing the wheel. Instead, it aims to combine existing methods and criteria, develop new ones when needed, and apply them in a systematic way. Outcomes from this thesis will fill gaps in aspects that the state of the art does not cover. In this context, **the main goal of this thesis is to advance in the ontology evaluation field, more precisely, in the ontology diagnosis activity**. Such general goal can be divided into the following subgoals: (a) to help ontology engineers to diagnose their ontologies, and consequently, (b) to ease the ontology diagnosis activity, lessening thus the effort required from ontology engineers. To do so, the **main contributions are: (a) a catalogue of pitfalls, or common errors, that will help developers diagnose their ontologies, improving therefore their quality, and (b) the design and implementation of detection methods to (semi)automatically diagnose ontologies, together with the corresponding technological support.** It is worth noting that the catalogue does not intend to be exhaustive and more pitfalls could be added in the future. The evaluation of the pitfall catalogue, the pitfall detection methods, and the technological support that encapsulates them, is mainly based on qualitative assessments at this point of the development and quantitative assessments are planned for future releases.

Before going further, it should be noted that the term "**pitfall**" instead of "error" is used throughout this thesis, as there might be discrepancies about whether the situations

considered in the catalogue represent actual errors or not. In this sense, a situation might be an error or not depending on: (a) modelling decisions taken by the team of developers; (b) ontology requirements previously set; (c) application context of the ontology; or (d) the scope or domain of the ontology. If a situation is not considered an error by a potential user[3] according to their own well-founded rationales, the corresponding pitfall could be omitted from the checking list, or omitted from the results when using the technological support presented also in this thesis.

The rest of the chapter is organized as follows: Section 1.1 presents the organization of this document. Section 1.2 provides an overview of the publications derived from the thesis work. Section 1.3 presents the research stays carried out along the development of the thesis. Section 1.4 lists the research and innovation projects in which the author of this thesis has participated.

1.1 Structure of the document

The thesis is structured as follows:

- Chapter 2 (State of the art) provides a general vision of the existing work in the areas related to this thesis and the terminology involved. We first frame the ontology evaluation activity within existing ontology development methodologies. Then, a review of methods and frameworks for ontology evaluation is presented. We end with a review of tools.
- Chapter 3 (Goals and contributions) defines the objectives, contributions, hypothesis and restrictions for this thesis. The chapter includes the research methodology followed.
- Chapter 4 (Catalogue for ontology diagnosis) presents the catalogue of 41 pitfalls including their descriptions, examples and ontology repair suggestions. This chapter also presents the quality model for ontology diagnose. Such quality model could be used as reference for researchers and practitioners for producing consistent, easily integrated and comparable ontology evaluations.

[3]Even though OOPS!'s main target are not ontology experts with high knowledge of Ontological Engineering, OOPS!'s users might range from ontology experts or ontology engineers to newcomers, including also ontology developers or practitioners.

- Chapter 5 (Methods and technological support: OntOlogy Pitfall Scanner!) describes the 48 methods defined for pitfall detection as well as the OOPS! architecture and main characteristics.
- Chapter 6 (Evaluation) presents different experiments and the validation of the method and technological support proposed in this thesis. This includes qualitative assessments of the pitfalls, their identification methods, and the technological support that encapsulates them, while quantitative assessment of the time saved by ontology engineers is planned for future research.
- Chapter 7 (Conclusions and future work) offers the main conclusions of this work, emphasizing its main contributions. This chapter also lists future lines of work that might be performed in the field of ontology evaluation.

Finally, the thesis includes the bibliographic references used for its elaboration and two annexes. The first annex includes information about OOPS!'s user interface and web service and the second annex provides the questionnaire used for evaluating OOPS! from an user-based perspective.

1.2 Derived publications

During the development of this thesis the resulting achievements were published in journals and international conferences with peer evaluation. Such publications, and other outcomes, are gathered in this section.

- **Journal articles**
  - **Poveda-Villalón, M.**, Gómez-Pérez, A., and Suárez-Figueroa, M.C. OOPS! (OntOlogy Pitfall Scanner!): An On-line Tool for Ontology Evaluation. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 10(2). Pages 7-34, ISSN: 1552-6283. Editorial IGI Publishing. October 2014. This publication presents the work described in Chapter 4 and Chapter 5.
- **Conference papers**
  - **Poveda-Villalón, M.**, Suárez-Figueroa, M.C., and Gómez-Pérez, A. Malas prácticas en ontologías. *Current Topics in Artficial Intelligence, CAEPIA*

2009. In Proceedings of the CAEPIA-TTIA 2009. ISBN: 978-84-692-6424-9. Pages 307-316. Sevilla, Spain. 9th - 13th November, 2009. This publication represents the bases for the work described in Chapter 4.

- **Poveda-Villalón, M.**, Suárez-Figueroa, M.C., and Gómez-Pérez, A. Validating ontologies with OOPS!. *Knowledge Engineering and Knowledge Management - 18th International Conference, EKAW 2012*. Galway City, Ireland, October 8th - 12th, 2012. This publication presents the first approach towards the work described in Chapter 4 and Chapter 5.
- Keet, C.M., Suárez-Figueroa, M.C., and **Poveda-Villalón, M**. The Current Landscape of Pitfalls in Ontologies. *Proceedings of the International Conference on Knowledge Engineering and Ontology Development (KEOD2013)*. Pages 132-139. Algarve, Portugal. 19th - 22nd September, 2013. This publication presents complementary information for Chapter 6.
- Suárez-Figueroa, M.C., Kamel, M., and **Poveda-Villalón, M**. Benefits of Natural Language Techniques in Ontology Evaluation: the OOPS! Case. *Proceedings of the 10th International Conference on Terminology and Artificial Intelligence (TIA 2013)*. Pages 107-110. 28th - 30th October, 2013. Paris, France. This work proposes improvements and future lines of work for the detection methods proposed in Chapter 5.

- **Book chapters**
  - **Poveda-Villalón, M.**, Gómez-Pérez, A., and Suárez-Figueroa, M.C. Common Pitfalls in Ontology Development. *Current Topics in Artficial Intelligence, CAEPIA 2009 Selected Papers*. LNAI 5988/2010. Springer-Verlag Berlin Heidelberg, 2010. This is a publication derived from the CAEPTIA-TTIA 2009 conference paper and also represents the bases for the work described in Chapter 4.
  - Keet, C M., Suárez-Figueroa, M.C., and **Poveda-Villalón, M.** Pitfalls in Ontologies and TIPS to Prevent Them. *Knowledge Discovery, Knowledge Engineering and Knowledge Management*. Springer-Verlag Berlin Heidelberg, 2015. A. Fred et al. (Eds.): IC3K 2013, CCIS 454. Pages 1-17, 2015. DOI: 10.1007/978-3-662-46549-3 8. This is a publication derived from

the KEOD2013 conference paper and also presents the work described in Chapter 4. More precisely, this work has contributed to the ontology repair suggestions.

- **Workshop papers and presentations**

  - **Poveda-Villalón, M.**, Suárez-Figueroa, M.C., Gómez-Pérez, A. A Double Classification of Common Pitfalls in Ontologies. *Workshop on Ontology Quality (OntoQual 2010) at the 17th International Conference on Knowledge Engineering and Knowledge Management. Lecture Notes in Computer Science*, pages 1-12. Springer. 15$^{\text{th}}$ October, 2010, Lisbon, Portugal.

  - **Poveda-Villalón, M.**, Vatant, B., Suárez-Figueroa, M.C., Gómez-Pérez, A. Detecting Good Practices and Pitfalls when Publishing Vocabularies on the Web. *Proceedings of the 4th Workshop on Ontology and Semantic Web Patterns at the 12th International Semantic Web Conference (ISWC2013).* 21$^{\text{st}}$ October 2013. Sydney, Australia.

  - **Poveda-Villalón, M.**, Suárez-Figueroa, M.C., Gómez-Pérez, A. A pitfall catalogue and OOPS!: an approach to ontology validation. Knowledge, Reasoning and Computation Workshop. 23$^{\text{rd}}$-24$^{\text{th}}$ April 2014. Madrid, Spain. Only presentation, no publication associated.

  - **Poveda-Villalón, M.**, Suárez-Figueroa, M.C., Gómez-Pérez, A. OOPS! (OntOlogy Pitfall Scanner!): a web-based tool for ontology evaluation. EUON (European Ontology Network) Workshop, 25$^{\text{th}}$ September 2014. Amsterdam, The Netherlands. Only presentation, no publication associated.

  - **Poveda-Villalón, M.**, Suárez-Figueroa, M.C., Gómez-Pérez, A. OOPS! (OntOlogy Pitfall Scanner!): evaluating ontologies online. DanTermBank Workshop, 9$^{\text{th}}$ January 2015. Copenhagen, Denmark. Only presentation, no publication associated.

- **Invited talks**

  - **Poveda-Villalón, M.**, Suárez-Figueroa, M.C., Gómez-Pérez, A. A pitfall catalogue and OOPS!: an approach to ontology validation. OntologySum-

mit2013, Summit Theme: Ontology Evaluation Across the Ontology Lifecycle, 31st January, 2013. Virtual presentation.[4]

- **Poveda-Villalón, M.** OOPS! - OntOlogy Pitfall Scanner!. An online system for ontology evaluation. 2nd UPM INNOVATECH International Workshop: Investing in High Technology for Success. 3rd December, 2014. Madrid, Spain.

- **PhD symposium**

  - **Poveda-Villalón, M.** A Reuse-based Lightweight Method for Developing Linked Data Ontologies and Vocabularies. *The Semantic Web: Research and Applications*. PhD symposium at 9th Extended Semantic Web Conference (ESWC2012). 27th - 31st May, 2012. Heraklion, Greece.

- **Demos**

  - **Poveda-Villalón, M.**, Suárez-Figueroa, M.C., Gómez-Pérez, A. Did you validate your ontology? OOPS!. Poster and demo session at 9th Extended Semantic Web Conference (ESWC2012). 27th - 31st May, 2012. Heraklion, Greece. In proceedings "The Semantic Web: ESWC 2012 Satellite Events". Pages 402-407. Springer-Verlag Berlin Heidelberg. 2015.
  - **Poveda-Villalón, M**, Suárez-Figueroa, M.C., and Gómez-Pérez, A. Did you validate your ontology? OOPS! (OntOlogy Piftall Scanner!). Poster and Demo session at 18th International Conference on Knowledge Engineering and Knowledge Management. Galway, Ireland. 8th - 12th October, 2012. No publication associated.

- **Posters**

  - **Poveda-Villalón, M.**, Suárez-Figueroa, M.C., Gómez-Pérez, A. Ontology Analysis Based on Ontology Design Patterns. WOP 2009 - Workshop on Ontology Patterns at the 8th International Semantic Web Conference (ISWC 2009). ISBN: ISSN 1613-0073. CEUR Workshop Proceedings. Pages 155-162. 25th October, 2009. Washington, DC.

[4] http://ontolog.cim3.net/cgi-bin/wiki.pl?ConferenceCall_2013_01_31 (last visited on the 12th October, 2015)

- **Poveda-Villalón, M.**, Suárez-Figueroa, M.C., Gómez-Pérez, A. OOPS! (OntOlogy Pitfall Scanner!): a web-based tool for ontology evaluation. 1st European Ontology Network (EUON) Workshop. 25th September 2014. Amsterdam, The Netherlands. No publication associated.

- **Recognitions**

  - Diploma to one of the three best technologies presented during the *"III Curso de Comercialización de Tecnologías UPM"* in November 2014 which led to a invited talk at the "2nd UPM INNOVATECH International Workshop: Investing in High Technology for Success" on the 3rd December, 2014. Madrid, Spain.
  - IAOA prize of OntologySummit2013 hackathon for the project *Clinics Ontohub OOR OOPS Integration*[5] on May, 2013.

- **Software registries**

  - OOPS! software has been registered by the Universidad Politénica de Madrid at the registry of *Comunidad de Madrid*, Spain. Registration number M009445/2012.
  - OOPS! software has been registered by Universidad Politénica de Madrid as national brand at the registry *Oficina Españold de Patentes y Marcas*. Registration number M3534342 and M3534343.

1.3 Research stays

During this thesis, research visits took place in the following research institutions:

- **University of Liverpool** at Liverpool, United Kingdom, from the 23rd of March to the 24th of June of 2011 supervised by Dr. Valentina Tamma. During this stay, the author analysed different dimensions of ontology evaluation and started the programming of the detection methods used in the technological support presented in this thesis.

[5] http://ontolog.cim3.net/cgi-bin/wiki.pl?OntologySummit2013_Hackathon_Clinics (last visited on the 12th October, 2015)

- **Free University of Berlin** at Berlin, Germany, from the 12th of March to the 1st of June of 2012. During this time the author worked within the "Web Based System Group" research group directed by Dr. Chris Bizer learning about Linked Data principles, tools, techniques and applications. This stay was funded by the Spanish *Ministerio de Educación mediante el programa de movilidad de profesores visitantes y de estudiantes, en el marco de implantación de estrategias de formación doctoral e impulso de la excelencia e internacionalización de los programas de doctorado de las universidades (Orden EDU/2719/2011)* grant.
- **Mondeca** at Paris, France, from the 1st of March to the 30th of June of 2013. During this period the author worked on the evaluation of ontologies regarding their publication process within the research department led by Dr. Florence Amardeilh. This stay was funded by the *XI Convocatoria de Ayudas del Consejo Social para la Formación y la Internacionalizaciión de Doctorandos de la Universidad Politiécnica de Madrid* program. This research visit led to a continuos collaboration and to the following publication related to the topic of this thesis:

  - **Poveda-Villalón, M.**, Vatant, B., Suárez-Figueroa, M.C., Gómez-Pérez, A. Detecting Good Practices and Pitfalls when Publishing Vocabularies on the Web. *Proceedings of the 4th Workshop on Ontology and Semantic Web Patterns at the 12th International Semantic Web Conference (ISWC2013).* 21st October 2013. Sydney, Australia.

1.4 Work in research and innovation projects

Prior and during this thesis, the author has acquired practical experience in Ontological Engineering and Linked Data generation through the participation in the following research and innovation projects:

- NeOn project[6] (FP6-027595) [1st March, 2006 to 28th February, 2010]: during the author's collaboration as undergraduate student in this European project she acquired knowledge about methodologies for building ontologies and experience with Ontology Design Patterns. This project finished before the author enrolled in the doctoral program.

[6] http://www.neon-project.org/ (last visited on the 12th October, 2015)

- GeoBuddies (TSI2007-65677-C02) [1st October, 2007 to 1st September, 2010]: in this Spanish project the author gained experience in ontology development and methodologies as well as in the manual diagnose of ontologies to find common errors. This project finished before the author enrolled in the doctoral program.
- mIO![7] (CENIT-2008-1019) [1st September, 2008 to 31st December, 2011]: this Spanish project provided experience in Ontological Engineering in a collaborative and distributed context. In addition, it provided the environment for experimentation with the proposed systematic approach for ontology diagnosis. In this sense, the project benefited from the pitfall catalogue and the first version of OOPS! during the ontology diagnosis activity.
- Development of the Ontology Network Specification Requirements and Conceptualization [15th June, 2009 to 31st January, 2010]: this international project carried out in collaboration with the World Health Organization provided experience in Ontological Engineering in a distributed environment and in collaboration with domain experts. This project finished before the author enrolled in the doctoral program.
- BuscaMedia[8] (CENIT-2009-1026) [1st October, 2009 to 31st December, 2012]: this Spanish project provided experience in Ontological Engineering and a real use case for experimentation with the technological support proposed in this thesis. This project also benefited form the pitfall catalogue and first versions of OOPS! during the ontology diagnosis activity.
- Ready4SmartCities[9] (FP7-608691/608711) [1st October, 2013 to 30th September, 2015]: this project was coincident with the last years of the thesis development. It benefited from the technology developed in the thesis, as OOPS! functionalities have been integrated within the catalogue of ontologies for smart cities developed as part of the project. In addition, the outcomes of the thesis have been useful for training sessions and tutorials carried out in the context of the project.

[7] http://www.cenitmio.es/ (last visited on the 24th November, 2015)
[8] http://www.cenitbuscamedia.es/ (last visited on the 24th November, 2015)
[9] http://www.ready4smartcities.eu (last visited on the 12th October, 2015)

Chapter 2

State of the art

2.1 Introduction

This chapter presents the state of the art of the topics of interest related to this thesis. First of all, significant definitions about ontology evaluation and related terms are provided in Section 2.2.

Because ontology evaluation is part of the ontology development process, it should be considered within methodologies for building ontologies. In this sense, Section 2.3 dwells on how the most well-known methodologies in the Ontology Engineering field and lightweight methodological approaches address the ontology evaluation activity. This review frames the main purpose of this thesis within the Ontological Engineering field.

Next, Section 2.4 offers the insights of some seminal works on ontology evaluation, which is the main issue addressed in this thesis. This section reviews the methodological works and the frameworks to address this activity. After that, an overview of the main tools for ontology evaluation from a practical point of view is provided in Section 2.5.

Finally, Section 2.6 concludes with a summary of the main gaps in the current state of the art for ontology evaluation. In a nutshell, we will show that precise and detailed methods for ontology evaluation are needed in order to cover all the different dimensions and aspects that might be evaluated in an ontology. We also point out the lack of automated user friendly tools for supporting this activity.

2.2 Terminology

This section gathers definitions about ontology evaluation following a chronological order. Standard definitions of evaluation and related topics are extracted from the ISO standard (ISO, 2011b) and presented here in a broader way for the sake of the understandability of the reader.

Regarding the evaluation definitions in the Ontological Engineering field, we mention the following ones as the most relevant and related to the work presented in this thesis:

- First definitions of ontology evaluation and related terms were provided in (Gómez-Pérez et al., 1995). These definitions have been largely reused along the ontology development process and evaluation literature. These definitions from (Gómez-Pérez et al., 1995) read as follows:

  - **Ontology evaluation**: *evaluation means to make a technical judgment of the ontologies, their associated software environments, and documentation with respect to a frame of reference*[10] *during each phase and between phases of their lifecycle. The term "Evaluation" subsumes the terms "Verification" and "Validation". This activity is divided into evaluation of*:

    * *Each individual definition and axiom.*
    * *Collections of definitions and axioms that are stated explicitly in the definitions in the ontology.*
    * *Definitions that are imported from other logical theories.*
    * *Axioms that can be inferred using collections of definitions and axioms.*

  - **Ontology Verification** *refers to building the ontology correctly, that is, ensuring that its definitions correctly satisfy its requirements, its competency questions or performs correctly in the real world. To guarantee that an ontology is well-verified, we have to verify its architecture, its lexis and syntax, and its content.*

  - **Ontology Validation** *refers to whether the meaning of the ontology definitions really represent the real world for which it was created.*

[10] A frame of reference can be: requirements specifications, competency questions, and the real-world.

  - Finally, **Ontology Assessment** *refers to the understanding, usability, generality, granularity, quality, portability, incrementalism, maintainability and uniformity of the definitions and axioms given by an ontology.*

- METHONTOLOGY (Fernández-López et al., 1997) states that **to evaluate an ontology** is to *make a technical judgment with respect to a frame of reference.*

- Denny Vrandečić's PhD dissertation (Vrandečić, 2010) provides a definition for ontology evaluation focused on the Web's ontologies: ***Ontology evaluation*** *is the task of measuring the quality of an ontology. It enables us to answer the following main question: How to assess the quality of an ontology for the Web?*

- The glossary of terms in the NeOn Methodology (Suárez-Figueroa et al., 2013a) provides the following definitions:

  - ***Ontology Assessment****. It refers to the activity of checking an ontology against the user's requirements, such as usability, usefulness, abstraction, quality.*
  - ***Ontology Evaluation****. It refers to the activity of checking the technical quality of an ontology against a frame of reference.*
  - ***Ontology Diagnosis****. It refers to the activity of identifying parts of the ontology directly responsible for incorrectness and incompleteness. Ontology diagnosis is triggered by ontology validation.*
  - ***Ontology Repair****. It refers to the activity of solving errors (incompleteness, incorrectness) in the ontology. This activity is triggered by ontology diagnosis.*
  - ***Ontology Validation****. It is the ontology evaluation that compares the meaning of the ontology definitions against the intended model of the world aiming to conceptualize. It answers the question "Are you producing the right ontology?"*
  - ***Ontology Verification****. It is the ontology evaluation that compares the ontology against the ontology requirement specification document (ontology requirements and competency questions), thus ensuring that the ontology is built correctly (in compliance with the ontology requirements specification). It answers the question "Are you producing the ontology right?"*

From "ISO/IEC 25040:2011 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Evaluation process" (ISO, 2011b), we extract the following definitions[11] related to the evaluation of software systems:

- ***Evaluation****: systematic determination of the extent to which an entity meets its specified criteria [SOURCE: ISO/IEC 12207:2008].*
- ***Evaluation method****: procedure describing actions to be performed by the evaluator in order to obtain results for the specified measurement applied to the specified product components or on the product as a whole.*
- ***Evaluation tool****: instrument that can be used during evaluation to collect data, to perform interpretation of data or to automate part of the evaluation. Note 1 to entry: Examples of such tools are source code analysers to compute code metrics, CASE tools to produce formalized models, test environments to run the executable programs, checklists to collect inspection data or spreadsheets to produce syntheses of measures.*

In the course of this thesis, we rely on NeOn definitions of ontology evaluation, validation, diagnosis and repair, as such definitions are the product of the evolution of seminal works throughout a community-based consensus process (Suárez-Figueroa, 2010) and widely known and used. To define evaluation method and tool we turn to the definitions provided in the ISO standard, as these terms are more general and in this thesis are applied to the specific case of ontology evaluation.

According to the NeOn glossary, ontology evaluation, diagnosis and repair are classified as activities, while ontology validation is classified as a process. Throughout this thesis such classification will be followed.

2.3 Ontology evaluation in methodologies for ontology development

This section summarizes the main methodologies and approaches within the Ontology Engineering field. In the following, brief descriptions of several methodologies and

[11] https://www.iso.org/obp/ui/#iso:std:iso-iec:25040:ed-1:v1:en (last visited on the 28th October, 2015)

recent works in Ontological Engineering are provided. This analysis is divided into methodologies (Section 2.3.1) and lightweight approaches (Section 2.3.2).

2.3.1 Methodologies

This section includes a brief description of the five main ontology development methodologies (Grüninger & Fox, METHONTOLOGY, On-To-Knowledge, DILIGENT and NeOn), listed chronologically. Special attention deserves the way such methodologies deal with ontology evaluation activities, to what extent this activity is considered and whether detailed guidelines for addressing it are provided.

2.3.1.1 Grüninger & Fox

The methodology proposed by Grüninger & Fox (Grüninger and Fox, 1995) is based on the idea of common sense models that have the ability to deduce answers to queries that require relatively shallow knowledge of the domain. The procedure for engineering such models is depicted in Figure 2.1 and is detailed along the following lines.

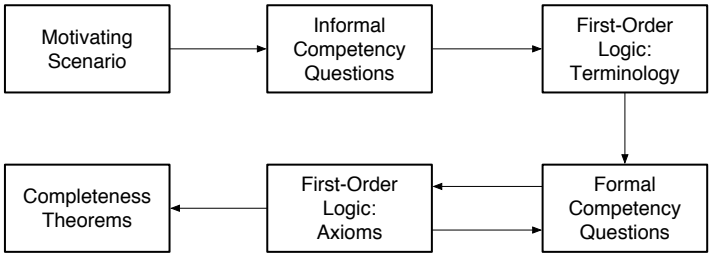


Figure 2.1: Procedure for ontology design and evaluation. Image taken from (Grüninger and Fox, 1995).

First of all, the ontology development or its extension should start with the presentation of some motivating scenario(s) that often have the form of story problems. Given the motivating scenario, a set of queries that an ontology must be able to answer will arise. These sets of queries are considered requirements of the ontology, more precisely, informal competency questions, since they are not yet expressed in a formal language.

The next step is to specify the terminology of the ontology based on the competency questions.

In the proposed methodology, such terminology should be formally specified using first-order-logic. Once the competency questions have been posed informally and the

terminology of the ontology has been defined, the competency questions are defined formally as an entailment or consistency problem with respect to the axioms in the ontology.

After that, axioms expressed in first-order-logic are included in order to specify the definitions of the terms and constrains. The development of axioms with respect to the competency questions is an iterative process. In this sense, if the proposed axioms are insufficient to represent the competency questions, additional objects or axioms should be added to the ontology as needed. This is a key step because a set of objects or ground terms does not constitute an ontology for the authors. Axioms must be provided to define the semantics of these terms.

Finally, the competency questions are tested by proving completeness theorems.

It is worth noting that this work introduced the Competency Questions notion that is widely used for extracting ontology requirements and for testing the ontology. Throughout the rest of the thesis we will therefore refer to this section when talking about CQs as method for evaluating ontologies.

2.3.1.2 METHONTOLOGY

METHONTOLOGY (Fernández-López et al., 1997; Fernández-López et al., 1999) is a methodology created to support the ontology development at the knowledge level. The methodology proposes a life cycle for ontology development based on evolving prototypes so that it is possible to add, change and remove terms in each iteration. This work also identifies the set of activities to be carried out during the ontology development process. This set of activities is based on the main activities identified for the software development process (IEEE, 1996) and on those used within knowledge engineering methodologies. Figure 2.2 shows the ontology life cycle model proposed in the methodology and the activities identified, grouped by management activities, technical activities and support activities. It should be mentioned that the management and support activities should be carried out simultaneously with the technical activities. As it is also shown in the figure, the ontology evaluation activity should receive more attention during the conceptualization activity to prevent error propagation in further stages of the ontology life cycle.

Regarding ontology evaluation, METHONTOLOGY not only provided novel terminologies and their definitions but also identified and classified different kinds of errors

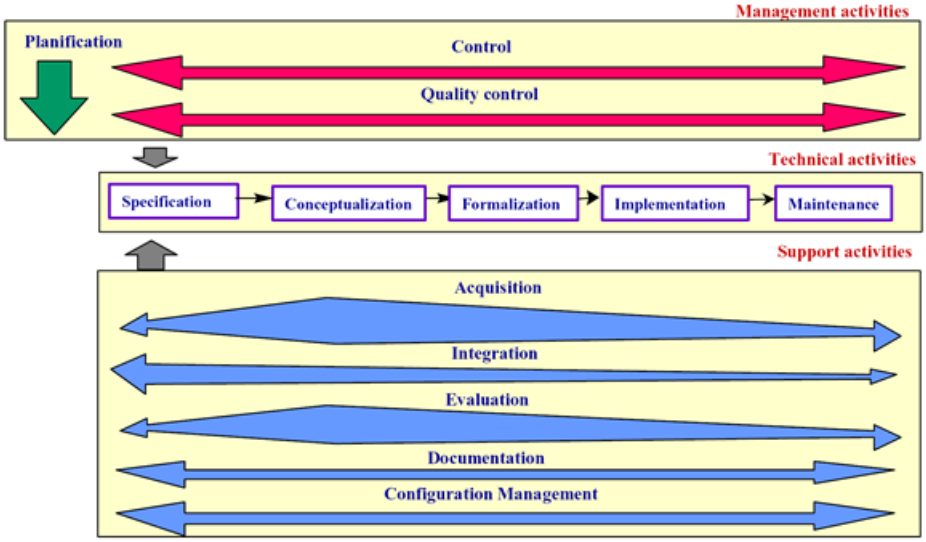


Figure 2.2: Development process and life cycle of METHONTOLOGY. Image taken from (Gómez-Pérez et al., 2004).

in taxonomies. These errors can be used as a check list for evaluating taxonomies. The list of errors that ontologists made when modelling taxonomic knowledge is subdivided in inconsistency, incompleteness, and redundancy errors is Figure 2.3.

It should be mentioned that this list of errors was defined for taxonomies developed assuming frames as a modelling paradigm.

2.3.1.3 ON-TO-KNOWLEDGE

The On-To-Knowledge methodology (Staab et al., 2001) focuses on the application-driven development of ontologies. The life cycle proposed consists of five phases, namely "Feasibility study", "Ontology kickoff", "Refinement", "Evaluation" and "Maintenance" as shown in Figure 2.4. It can also be observed that two loops are defined, one between the "Refinement" and "Evaluation" phases and the other one between the "Refinement" and "Maintenance" phases, being therefore an iterative development oriented methodology.

The evaluation phase aims at proving the usefulness of the ontology under development together with its associated software environment. During this phase two main steps are carried out:

- Step 1: the ontology engineer checks whether the target ontology conforms with

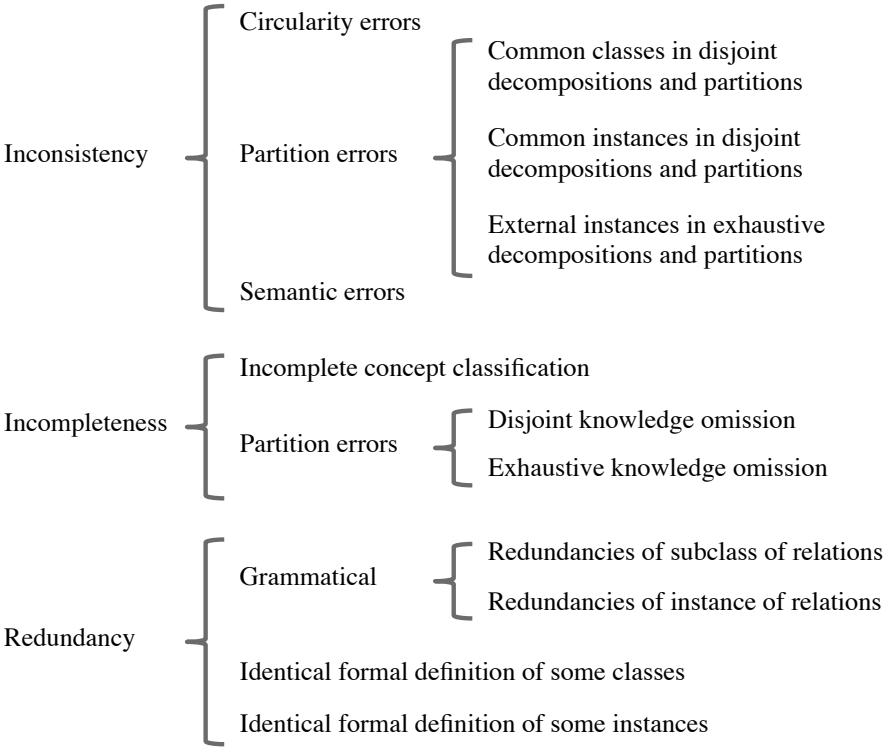


Figure 2.3: Taxonomy errors defined in METHONTOLOGY. Image taken from (Gómez-Pérez et al., 2004).

the ontology requirements specification document and whether the ontology supports or answers the competency questions stated in the kickoff phase.

- Step 2: the ontology is tested in the target application environment in order to get feedback from beta users that would be a valuable input for further refinement of the ontology, as well as usage patterns of the ontology.

As already mentioned, this evaluation phase is closely linked to the refinement phase. An ontology engineer might need to perform several cycles.

2.3.1.4 DILIGENT

The DILIGENT (Pinto et al., 2004) methodology was proposed to support ontology development in a distributed environment, in a loosely-controlled way and considering the evolution of ontologies. In this scenario, the actors involved in the collaborative development of the same ontology are experts with different and complementary skills,

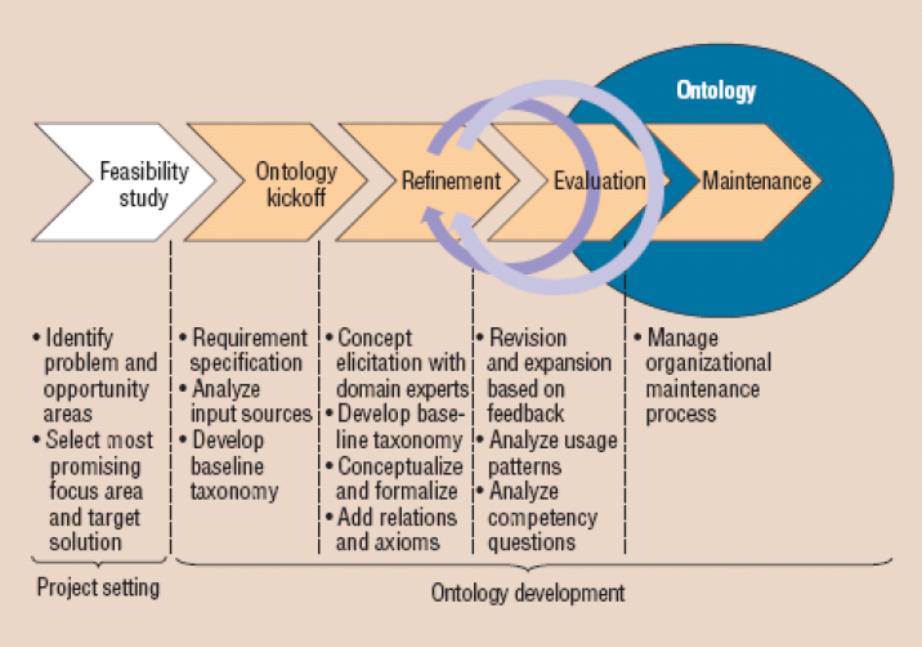


Figure 2.4: Development process and life cycle of On-To-Knowledge. Image taken from (Staab et al., 2001)

ontology users and ontology developers. The general process proposed comprises five main activities: (1) build, (2) local adaptation, (3) analysis, (4) revision, and (5) local update, as shown in Figure 2.5.

At the beginning of the process, domain experts, users, knowledge engineers and ontology engineers develop an initial ontology. Once the product is available, users can start using it and locally adapting it for their own purposes. In their local environment they are free to change the reused shared ontology. However, they are not allowed to directly change the ontology shared by all users. The control board collects change request to the shared ontology and analyses them in order to decide which changes will be introduced in the next version of the shared ontology. The board should regularly revise the shared ontology, so that local ontologies do not diverge too far from the shared ontology. Users can update their own local ontologies to better use the knowledge represented in the new version.

This methodology does not address the ontology evaluation activity. The "revision" activity refers to the control of differences among local and shared ontologies instead of to the quality assurance of the ontology.

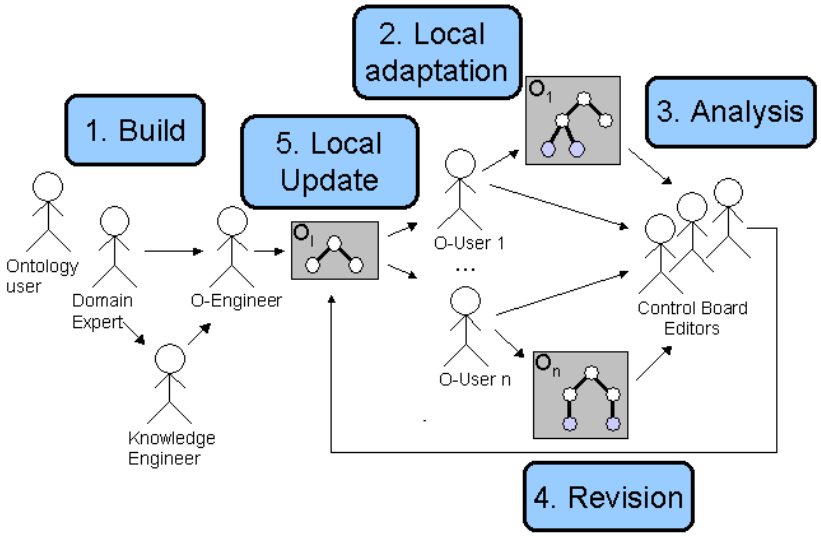


Figure 2.5: DILIGENT ontology life cycle. Image taken from (Pinto et al., 2004).

2.3.1.5 NeOn Methodology

The NeOn Methodology (Suárez-Figueroa, 2010; Suárez-Figueroa et al., 2015) for ontology networks development is one of the outcomes of the NeOn European project.[12] Its main goal is to provide support for the drawbacks identified in the previous methodologies, namely, to address the collaborative dimension of the development and to provide concrete guidelines for reuse and reengineering of knowledge sources. To do so, this methodology identifies nine flexible scenarios for building ontology networks, as it has been realized that there are many alternatives or ways to build ontologies and ontology networks. Such scenarios, shown in Figure 2.6, are those identified as the most common scenarios during ontology network development, even though the list should not be considered exhaustive:

- Scenario 1: From specification to implementation.
- Scenario 2: Reusing and reengineering non-ontological resources.
- Scenario 3: Reusing ontological resources.
- Scenario 4: Reusing and reengineering ontological resources.
- Scenario 5: Reusing and merging ontological resources.

[12] http://www.neon-project.org/ Last visit 28th October, 2015

- Scenario 6: Reusing, merging and reengineering ontological resources.
- Scenario 7: Reusing Ontology Design Patterns (ODPs).
- Scenario 8: Restructuring ontological resources.
- Scenario 9: Localizing ontological resources.

The ontology support activities shown at the bottom of Figure 2.6, namely, knowledge acquisition, documentation, configuration management, evaluation and assessment, should be carried out during the whole development process in combination with any scenario selected for developing the ontology network. It is worth noting than Scenario 1 is considered the most common and basic scenario when developing ontologies, therefore it is mandatory to follow it in every ontology development project combined, when needed, with any number of the other scenarios.

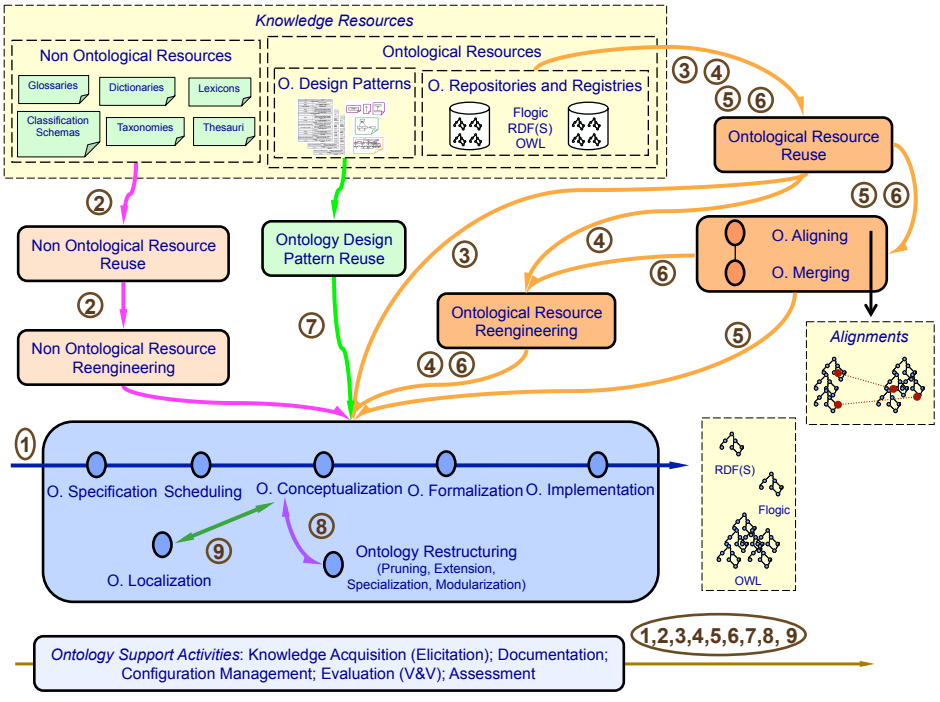


Figure 2.6: Scenarios for building ontology networks proposed in NeOn. Image taken from (Suárez-Figueroa et al., 2015)

This methodology considers ontology evaluation a support activity to be carried out in the course of the whole development. In the context of the NeOn Methodology, a compilation of methods and techniques proposed by other authors is provided

in (Sabou and Fernández, 2012). This work presents a structured summary of the ontology (network) evaluation activity according to the templates to describe activities and processes defined in the NeOn Methodology. The authors also present a workflow for ontology (network) evaluation based on the selection of existing approaches according to the evaluation goals chosen by the user. The list of possible goals is as follows: (a) domain coverage; (b) quality of modelling; (c) suitability for an application/task; and (d) adoption and use.

Following, the authors propose a set of evaluation metrics according to a frame of reference to be selected by the user. The frames of reference proposed are: (a) gold standard; (b) application based; (c) data driven; (d) assessment by humans; (e) topology based; (f) language based; and (g) methodology based.

2.3.2 Lightweight approaches

This section includes a brief chronological description of four lightweight approaches to ontology development (Ontology Development 101 guide, EXtreme Ontology, RapidOWL and XD Methodology), focusing on the way such methodologies deal with the ontology evaluation activity.

2.3.2.1 Ontology Development 101 guide

The "Ontology Development 101" guide (Noy and McGuinness, 2001) represents one of the basic and most popular tutorials for building ontologies. The guide does not intend to cover all the issues that an ontology developer may address. Instead, it tries to provide a starting point; an initial guide to help newcomers to develop ontologies. This guide proposed seven steps for creating the ontology and also provides recommendations and guidelines for each step and for general topics. These steps are the following:

1. Determine the domain and scope of the ontology
2. Consider reusing existing ontologies
3. Enumerate important terms in the ontology
4. Define the classes and the class hierarchy
5. Define the properties of classes-slots
6. Define the facets of the slots

7. Create instances

Even though its main goal is to guide developers to build ontologies, this guide lists common errors related to the following topics and prevents developers from making them:

- Creating both a singular and a plural version of the same concept
- Using synonyms of the same concept as different classes
- Including cycles in the class hierarchy
- Defining domain and ranges in a too general or too narrow way
- Using ontology elements incorrectly
- Deciding whether a particular concept is a class in the ontology or an individual
- Not following a naming convention

It should be mentioned that even though this guide was developed for frame-based systems, most of the steps and recommendations could also be applied to description logic ontologies.

2.3.2.2 EXtreme Ontology

The EXtreme Ontology (Hristozova and Sterling, 2002) method[13] aims at developing ontologies applying the principles and techniques of EXtreme Programming (XP). The process proposed consists of the following steps:

1. Fetching the requirements of the system. The collected data will be used to create a baseline ontology consisting of a small number of concepts unavoidable for a particular domain.
2. Defining competency questions. The competency questions are provided by the user. Competency questions serve both as a base for extracting the ontology concepts and as test cases for completeness and validity.
3. Running validation tests. Ontology engineers check whether the baseline ontology can answer the competency questions.

[13] Authors also refer to this method as EXPLODE in subsequent publications.

4. Running redundancy tests. When ontology engineers include new classes or relationships, they should first check for existing ontology entities and then add the proposed ones only if they are not redundant.
5. Planning. The planning estimates the cost and time for building and managing the ontology. The depth and width of the ontology are analysed. Modelling decisions are made during this step.
6. Integrating. Ontology developers should ensure that the ontology works properly with the rest of the components of the system.
7. Running acceptance tests. Tests should be created for each iteration step. For each question and validation test there is a level of acceptance of the results.

This methodology takes into account the application based ontology evaluation and domain coverage (based on data and competency questions). However, no specific techniques or methods for carrying out this activity are provided.

2.3.2.3 RapidOWL

RapidOWL (Auer, 2006) is an adaptive, light-weight methodology for collaborative Knowledge Engineering inspired by the wiki approach to collaboration. This methodology is based on the idea of iterative refinement, annotation and structuring of a knowledge base.

Instead of distinguishing different phases within the life cycle as conventional methodologies do, agile methodologies give the importance of applying a change a much higher value than being located in a certain stage of the life cycle. Therefore, based on agile methodologies ideas, RapidOWL does not provide a phase model but values from which principles are derived for the engineering process in general, as well as practices for establishing those principles. Figure 2.7 shows such values, principles and practices proposed by the methodology.

As it can be observed, regarding the ontology evaluation activity, only consistency checking is considered. This task is suggested to be driven mainly by users. Due to the fact that this methodology is oriented to large knowledge base development and not only ontologies, it suggests the extraction of distinct parts or "slices" of the knowledge base adhering to a given OWL profile or a rule language to perform constraint and/or consistency checks using reasoners.

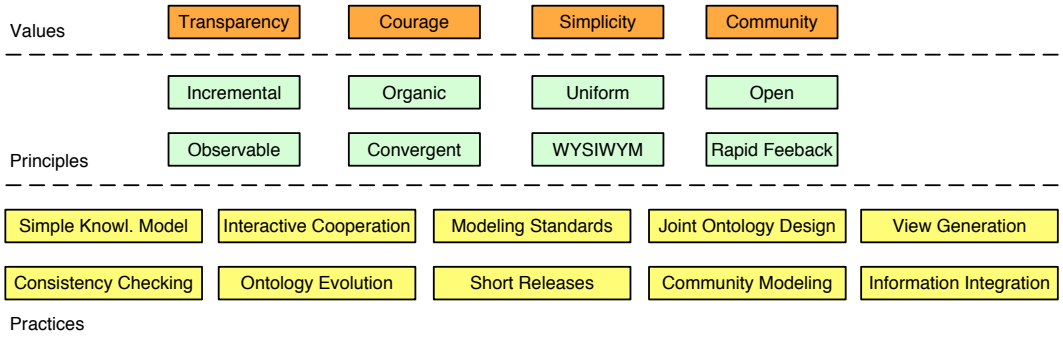


Figure 2.7: The building blocks of RapidOWL: Values, Principles, Practices. Image taken from (Auer, 2006).

2.3.2.4 XD Methodology

The XD method, first introduced in (Suárez-Figueroa et al., 2009) and revisited in (Presutti et al., 2012), is inspired by the agile software methodology called eXtreme Programming (XP) (Shore and Warden, 2007) in many ways. However, its focus is different: where XP diminishes the value of careful design, this is exactly where XD has its main focus. XD is test-driven, and applies the divide-and-conquer approach as XP does. Also, XD adopts pair design, as opposed to pair programming. The main principles of the XD method can be summarized as follows: (a) customer involvement and feedback; (b) customer stories, Competency Questions, and contextual statements; (c) iterative development; (d) test-driven design; (e) modular design; (f) collaboration and integration; (g) task-oriented design; (h) pair-design. The overall process proposed by this methodology is depicted in Figure 2.8. It should be mentioned that the process shown in this picture is usually not a sequential one. In most cases the arrows indicate an input/output dependency rather than a sequence of actions. For example, the integration and release steps could be performed in parallel with Steps 4 and 5, where new modules are produced.

This methodology deals with ontology evaluation in a test-driven way. More precisely, in the case of XD, the testing is used as an integrated means for completing the developed modules. Stories, CQs, reasoning requirements, and contextual statements are used to develop unit tests, for example, transforming CQs into SPARQL queries.

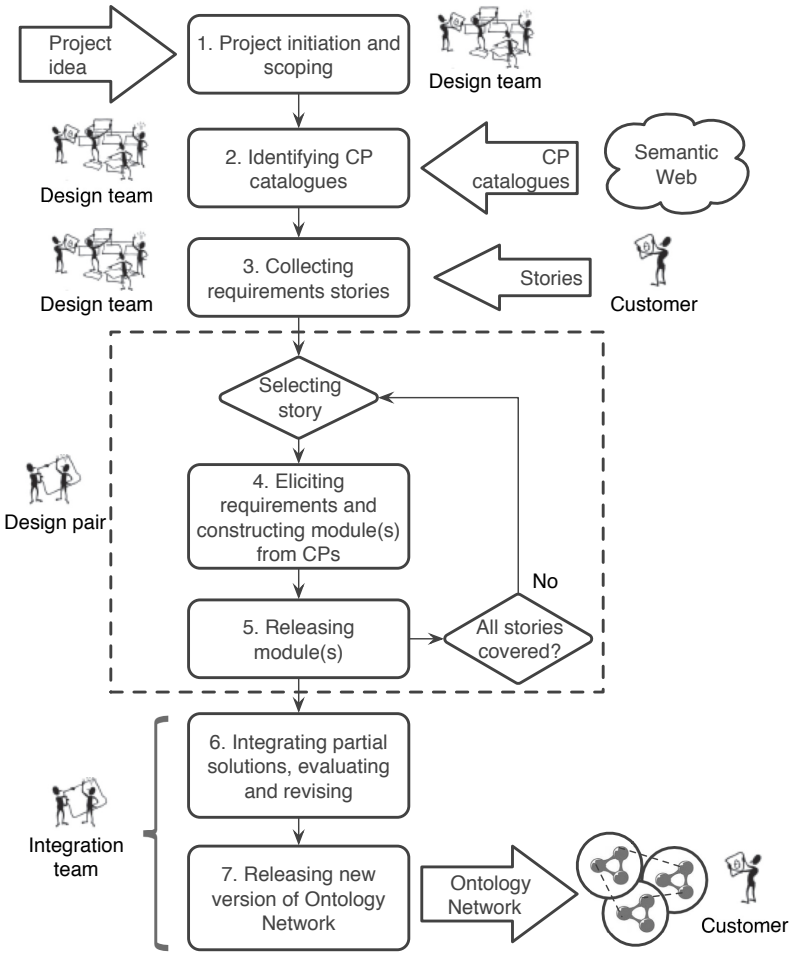


Figure 2.8: The overall XD process, for Content Pattern reuse. Image taken from (Presutti et al., 2012).

2.4 Frameworks and methods for ontology evaluation

Ontology evaluation is a complex ontology engineering process, mainly due to two reasons. The first one is its applicability in different ontology engineering scenarios, such as development and reuse, and the second one is the abundant number of approaches and metrics. A broad selection of these approaches are gathered in (Sabou and Fernández, 2012) and (Brank et al., 2005). In the following sections we describe the main approaches, methods and frameworks for ontology evaluation.

2.4.1 Gómez-Pérez's framework

Gómez-Pérez (Gómez-Pérez, 1995) proposed a set of initial and general ideas to guide the evaluation of ontologies. This work states that the development team must perform a global technical evaluation that ensures well-defined properties in (1) the definitions of the ontology, (2) the software environment used to build, reuse and share definitions, and (3) the documentation. As software evaluation can be performed by software engineering evaluation techniques, her work focuses on the ontology side, that is, points (1) and (3). The evaluation of definitions of the ontology aims at detecting the absence of some well-defined properties. The following steps are proposed: (a) check the **structure or architecture** of the ontology; (b) check the **syntax** of the definitions; (c) check the **content** of the definitions. This first step tries to identify both the lack of knowledge as well as mistakes in the definitions dealing with the problem of the three Cs:

- **Consistency** refers to the incapability of getting contradictory conclusions simultaneously from valid input data.
- **Completeness** refers to the extension, degree, amount or coverage to which the information in a user-independent ontology covers the information of the real world.
- **Conciseness** refers to the question whether all the information gathered in the ontology is useful and precise. Conciseness does not imply the absence of redundancies.

The evaluation of the documentation in knowledge sharing technologies aims at guaranteeing that certain documents are developed and that they evolve in step with the definitions and software.

2.4.2 OntoClean

The OntoClean methodology (Guarino and Welty, 2009; Welty and Guarino, 2001) was developed to validate the ontological adequacy and logical consistency of taxonomic relationships. This work is based on highly general notions drawn from philosophy, such as *essence*, *rigidity*, *identity*, *unity* and *dependence*. Based on these notions, a set of meta-properties are described, which impose several constraints on the taxonomic relationships between concepts in the hierarchy. The methodology consists of four

layers such that the notions and techniques within each layer are based on the notions and techniques in the lower layers. These layers are (from lowest to top): (1) formal ontological properties, including the meta-properties; (2) useful property kinds that can be seen as a library of reference cases useful to characterize the meta-properties of a given property, and to check for constraint violations; (3) ontology-driven modelling principles, from which a backbone ontology consisting of rigid properties and a stratified ontology are identified; (4) the top level ontology.

The OntoClean methodology allows the user to analyse taxonomies focusing on the nature of the properties involved in subsumption relationships. In addition, the methodology does not focus on structural similarities between property descriptions, but on the validation of single subsumption relationships based on the intended meaning of their arguments in terms of the meta-properties defined.

2.4.3 Rector et al.

The work presented by Rector et al. in (Rector et al., 2004) provides a list of the most common difficulties encountered by newcomers when modelling using description logics, including OWL-DL. This work focuses on the use of OWL-DL so as to make effective use of the classifiers or reasoners. This research only concerns issues in defining OWL classes. The most common problems reported in this work are:

- Failure to make all information explicit
- Mistaken use of universal rather than existential restrictions as the default
- Open world reasoning common problems
- The effect of range and domain constraints as axioms
- Trivial satisfiability of universal restrictions
- Difficulty to differentiate between defined and primitive classes
- Errors in understanding common logical constructs
- Expecting classes to be disjoint by default
- Difficulty in understanding subclass axioms used for implication

2.4.4 Semiotic metrics suite

The approach proposed in (Burton-Jones et al., 2005) lists a number of attributes related to ontology quality such, as lawfulness, richness, interoperatilbity, consistency, etc. These attributes are grouped in the following four metric suites: syntactic, semantic, pragmatic, and social. This work follows the measurement tradition in software engineering, and authors adapt it to the Ontological Engineering field. They suggest to identify the internal attributes of ontologies that give rise to external quality attributes. It should be noted that only relationships between these internal attributes are considered in their work, and therefore external attributes of ontologies as maintainability or understandability are out of the scope. Their work also provides an implementation of the attributes for DAML ontologies. For each method, a value between 0 and 1 is given in a way such that a combination of weight average of the attributes for each suite can be calculated. Then, the overall ontology quality can also be calculated in a similar way, obtaining therefore a value between 0 and 1 that indicates the overall quality of the ontology, whereby values closer to 1 denote better ontologies.

2.4.5 OntoQA

The OntoQA approach presented in (Tartir et al., 2005) analyses ontology schemas (T-box) and their instances (A-box) in order to describe them through a set of metrics. The metrics are divided into schema metrics and instance metrics.

Schema metrics aim at evaluating ontology design and its potential for rich knowledge representation by means of the following metrics: relationship richness, attribute richness and inheritance richness (of the schema).

Instance metrics evaluate the placements of instance data within the ontology and the effective use of the ontology by means of the following metrics: class richness, average population, cohesion, instance distribution, fullness, inheritance richness (of classes per schema level), relationship richness, connectivity and readability.

2.4.6 O^2 integrated model

In order to integrate methods for ontology evaluation into a one single framework by means of a formal model, (Gangemi et al., 2006) propose a multi-layered approach consisting of: (1) O^2, a meta-ontology that allows to treat an ontology as a semiotic object;

(2) oQual (for Ontology Quality), an ontology for ontology validation that models ontology evaluation as a diagnostic task; and (3) qood (for Quality-Oriented Ontology Description), the component of oQual which describes the desired evaluation criteria, that is an ontology of ontology validation. The three main types of dimensions for evaluation are identified in their work are:

- **Structural dimension**: which focuses on the syntax and formal semantics. In this sense, the topological, logical and meta-logical properties of an ontology can be measured by means of a context-free metric.
- **Functional dimension**: which is related to the intended use of a given ontology and of its components, that is, their function in a context. It focuses on the conceptualization specified by an ontology.
- **Usability-profiling dimension**: which focuses on the ontology profile (annotations), which typically addresses the communication context of an ontology.

2.4.7 Unit tests

The idea of adapting the notion of unit tests commonly used within software engineering into the ontology evaluation field was presented in 2006 (Vrandečić and Gangemi, 2006) and refined in (Vrandečić, 2010). This work focuses on web ontologies as defined by the OWL DL standard and addresses both the T-box and the A-box. In his work, unit tests are not meant to be complete formalizations of the ontology requirements but rather indicators of potential errors or omissions. The different ideas, inspired by the notion of unit testing, and explored and presented in such approach are, among others:

- Affirming derived knowledge: this consists in creating two ontologies, the positive test ontology and the negative test ontology. Ontology developers should check that for each axiom in the positive test ontology, such axiom is being inferred by the tested ontology. There might be an error in the tested ontology for each axiom that is not being inferred. Ontology developers should check that for each axiom in the negative test ontology, such axiom is not being inferred by the tested ontology. There might be an error in the tested ontology for each axiom that is being inferred.

- Formalized competency questions: this consists in (a) formalizing the competency questions in a query language; (b) writing down the expected answers for each question; (c) executing such queries; and (d) comparing the obtained results with the expected results. Note that answering all competency questions does not mean that the ontology is complete.
- Expressive consistency checks: this consists in introducing a test ontology T for an ontology O that includes the high axiomatization of the terms included in O and check the satisfiability of the merged ontology $T \cup O$

2.4.8 Pattern-based debugging guidelines

The approach presented in (Corcho et al., 2009) provides both a list of common antipatterns and a debugging strategy. The antipatterns are divided into the following categories:[14] (a) Detectable Logical AntiPatterns that represent errors that DL reasoners and debugging tools are able to detect (5 antipatterns); (b) Cognitive Logical AntiPatterns that represent modelling mistakes possibly derived from a misunderstanding of the logical implications of an expression (1 antipattern); and (c) Guidelines which are correct expressions used in ontology elements but that could be replaced by simpler alternatives or more accurate expressions (4 guidelines).

The lifecycle proposed for debugging ontologies starts with the identification of unsatisfiable classes using a reasoner or debugging tool. After that, antipatterns for the selected class must be identified. The knowledge engineer then proposes recommendations for corrections that might not be automated. Once the changes are made, new unsatisfiability checks should be conducted. This iterative process runs until there is not any unsatisfiable class left in the ontology.

Authors also propose a more detailed strategy, based on the catalogue of antipatterns described. This strategy suggests to follow a specific order when checking the antipatterns, based on authors' experience.

2.4.9 Vrandečić's framework

Later in 2010, Vrandečić proposed a framework for ontology evaluation expressed by an ontology (Vrandečić, 2010). This new framework was inspired by the semiotic meta-

[14]The number of antipatterns proposed for each category are represented between brackets.

ontology O^2 and oQual (see Section 2.4.6). The framework proposed by Vrandečić is depicted in Figure 2.9 where it can be observed that a given ontology specifies a conceptualization and constrains the construction of models that satisfy the ontology. In the figure, the slashed arrows represent the "expresses" relation. According to their structural definition, ontologies are considered a set of axioms and their serialization or expression is given in the form of an ontology document. Since methods can not asses ontologies directly, evaluation methods evaluate the quality of an ontology document. The figure also shows the different levels that an ontology document might express as an XML info set, RDF graph or the ontology itself. In such framework, it is also proposed that ontology evaluation might be expressed by an ontology, which enables the integration of the results of several different methods to build a complex evaluation out of a number of simple evaluations.

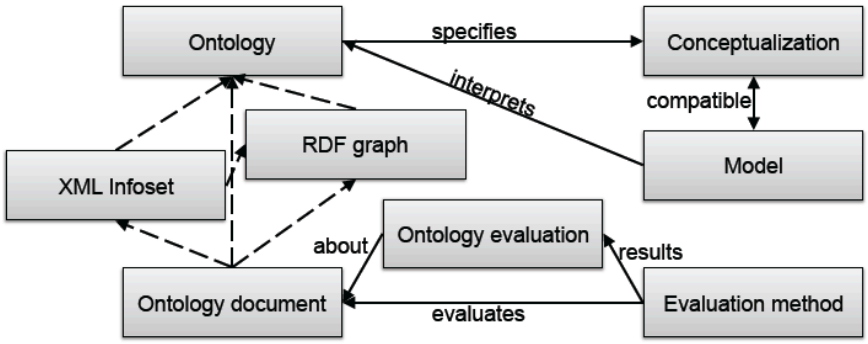


Figure 2.9: Framework for ontology evaluation. Image taken from (Vrandečić, 2010).

In this work, six aspects that are amenable to the automatic, domain- and task- independent verification of an ontology are identified, and a number of evaluation methods are proposed in relation to each aspect:[15] (a) vocabulary (10 methods); (b) syntax (1 method); (c) structure (3 methods); (d) semantics (2 methods); (e) representation (2 methods); and (f) context (5 methods).

2.4.10 OQuaRE framework

The OQuaRE framework (Duque-Ramos et al., 2011) proposes evaluating ontology quality by means of adapting the SQuaRE standard (ISO, 2011a) for software quality eval-

[15]The number of methods proposed for each aspect are represented between brackets

uation. The method requires the definition of a quality model and quality metrics for evaluating the ontology quality. The quality model consists of the following quality dimensions or characteristics: (a) structural; (b) functional adequacy; (c) reliability; (d) operability; and (e) manteinability. These characteristics are divided into sub characteristics, which are evaluated by applying a series of quality metrics. The proposed metrics for the sub characteristics are mainly topology-based metrics, for example, the mean number of properties and relationships per class. For each metric, a mapping between the range of values of the metric and the range 1 to 5 is provided, taking into account that high values in the metrics might not correspond to a high quality score. Once the conversion to the 1 to 5 scale is made, 1 represents lower quality and 5 the best possible value.

In summary, this approach gives an idea of the structure and topology of the ontology by adapting a model for software quality evaluation. However, it does not address specific ontology quality issues.

2.5 Tools for ontology evaluation

In order to ease and support the methodological guidelines and frameworks proposed for ontology evaluation, tools that assist ontology developers have been created.

While in previous sections a number of methods and techniques for ontology evaluation have been reviewed, in this section we focus on tools. More precisely, we review tools that evaluate ontologies by focusing on the explicit content of the ontology as classes, properties, and the explicit and formal relations between them, and metadata.

Other basic systems, such as syntax validators (for example, RDF Validation Service[16] or Manchester OWL Validator[17]), are out of scope, as they only check whether the ontology is compliant with the given ontology implementation language, which is no longer a challenge within ontology evaluation.

In addition, systems focused on the data level, for example OntologyTest (García-Ramos et al., 2009), or in particular instantiations of models (for example implementation of SKOS models evaluators as qSKOS (Mader et al., 2012) and Skosify (Suominen and Hyvönen, 2012)) are not considered part of the state of the art directly related to

[16] http://www.w3.org/RDF/Validator/
[17] http://owl.cs.manchester.ac.uk/validator/

this thesis. Systems aimed at debugging ontologies based on reasoning power, as for example the plug-in for the NeOn Toolkit called Radon (Ji et al., 2009), are also out of the scope of this thesis.

2.5.1 ODEClean

In the following, a description of ODEClean (Fernández-López and Gómez-Pérez, 2002), a plug-in for WebODE ontology editor (Arpírez et al., 2003), is provided. Even though ODEClean does not support OWL ontologies, we consider mentioning it because it is a pioneer implementation of the OntoClean seminal work (see Section 2.4.2).

First, it should be explained that WebODE was an ontology editor that allowed the collaborative construction of ontologies at the knowledge level, giving technological support to METHONTOLOGY. ODEClean allowed cleaning taxonomies following the OntoClean method and it was integrated into WebODE. When ontologists built an ontology in WebODE, they could select whether they wanted to build the taxonomy taking into account the OntoClean principles. Picking up an ontology from WebODE ontology library and cleaning its taxonomy just by assigning values of the meta-properties of each concept were also possible options.

The main functions provided by ODEClean were:

1. Establishing the evaluation mode: on demand or whenever an error is detected.
2. Assigning meta-properties to concepts: the user can set up meta-properties concerning identity, unity, dependency and rigidity.
3. Focus on the rigid properties: to show or not the non-rigid properties.
4. Evaluation according to taxonomic constraints: showing the error when the user runs the evaluation of the ontology.

The steps for developing ODEClean and its main components are depicted in Figure 2.10. ODEClean's ontology was built containing OntoClean knowledge useful for taxonomy cleaning. Then, the ontology was translated into Prolog in order to generate a code with an inference engine available at that moment. Finally, taking the Prolog ontology as input, the rest of the modules of ODEClean were built, namely, the user interface and the communication with the rest of WebODE environment.

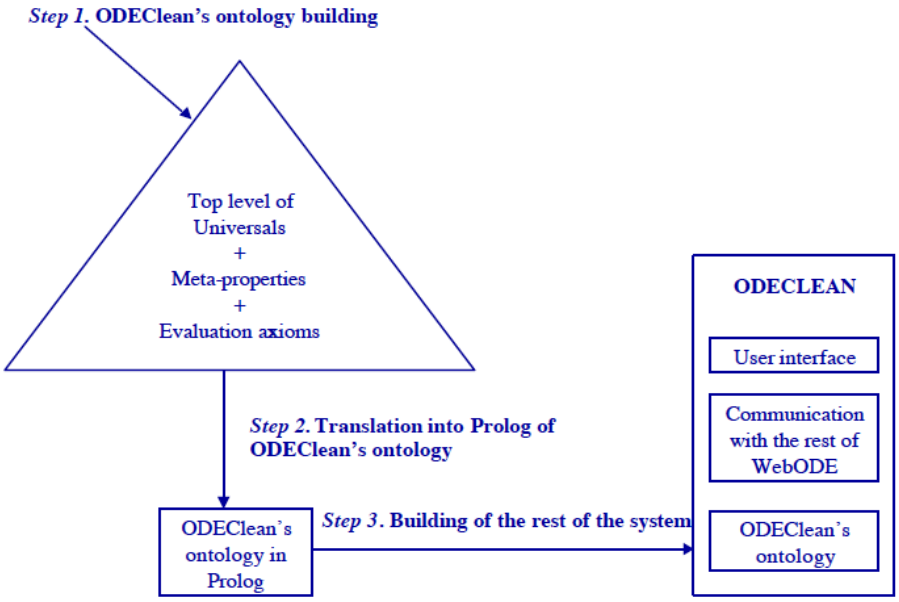


Figure 2.10: The ODEClean development process and its main components. Image taken from (Fernández-López and Gómez-Pérez, 2002).

2.5.2 ODEval

ODEval (Corcho et al., 2004) was developed in order to ease the evaluation of RDF(S) and DAML+OIL and OWL concept taxonomies in order to support the errors in taxonomies proposed in (Gómez-Pérez et al., 2004) (see Section 2.3.1.2). From the list of errors that arise when modelling taxonomical knowledge (See Figure 2.3), the system focuses on the automatic detection of inconsistencies (circularity issues and partition errors) and redundancy grammatical problems. This system is no longer available nor maintained. In addition, no architectural description has been found.

2.5.3 AEON

Authors in (Völker et al., 2008) present an approach for supporting ontology evaluation according to the OntoClean methodology (see Section 2.4.2). The main motivation behind this work is that even though OntoClean is well documented in numerous publications, its use is rather infrequent due to the high cost of its application. More precisely, authors pointed to the need of highly experienced ontology engineers in order to tag concepts with the correct meta-properties defined in the methodology. They pro-

posed a tool, AEON, which automatically tags concepts with the appropriate OntoClean meta-properties and performs the constraint checking.

The presented system matches lexico-syntactic patterns on the Web to obtain positive and negative evidence for rigidity, unity, dependence and identity of concepts in RDFS or OWL ontologies. Figure 2.11 provides an overview of the AEON architecture. The system consists of an evaluation component in charge of training and evaluation, a classifier for mapping given sets of evidence to meta-properties such as +R or -U, a pattern library, and a search engine wrapper.

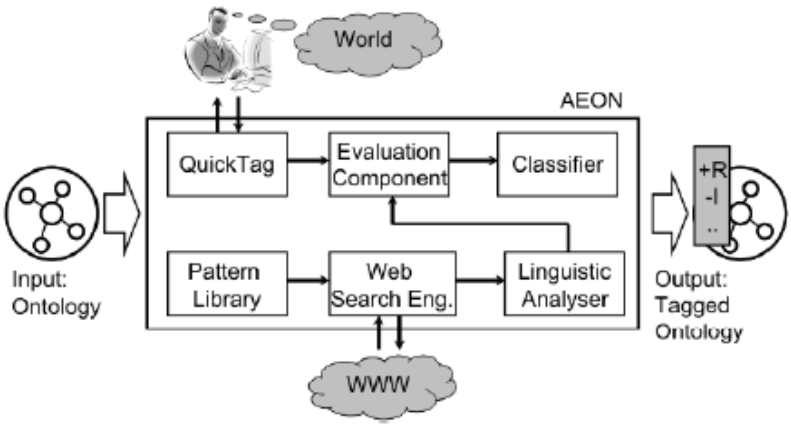


Figure 2.11: AEON system architecture. Image taken from (Völker et al., 2008)

The pattern library contains a set of abstract patterns for each meta-property, including a specification of the type of evidence it produces, for example, negative evidence for rigidity. Given a set of instantiated patterns, the search engine wrapper uses the Google™API to retrieve web pages or snippets.[18]

Finally, for each pattern contained in the above mentioned pattern library, the positive or negative evidence for a concept with a certain meta-property is calculated. Given a concept and the evidence values obtained for all patterns, a classifier makes the decision whether or not a specific meta-property applies to the concept. Authors developed a classifier for each meta-property and trained them on a small number of examples provided by human annotators. The manual effort rests with the creation of a gold standard ontology and classifiers to be trained on this ontology.

[18]Parts of web pages containing the regarding search string.

2.5.4 Eyeball

Eyeball[19] is a library and command-line tool part of the Jena[20] family of RDF/OWL tools. A graphical user interface in the form of a desktop application is also provided; however, the interface is still in an experimental phase. This system allows to check for various common problems that often result in technically correct but implausible RDF. These checks are performed by an inspector plug-in which can be customized by the user. The list of inspectors provided in Eyeball is the following:

- unknown (with respect to the schemas) properties and classes
- bad prefix namespaces
- ill-formed URIs, with user-specifiable constraints
- ill-formed language tags on literals
- datatyped literals with illegal lexical forms
- unexpected local names in schema namespaces
- untyped resources and literals
- individuals having consistent types, assuming complete typing
- likely cardinality violations
- broken RDF list structures
- suspected broken use of the typed list idiom
- broken OWL restrictions
- user-specified constraints written in SPARQL

2.5.5 Moki

The wiki-based ontology editor[21] MoKi (Pammer, 2010) incorporates ontology evaluation functionalities by means of the MokiValidaton plug-in. Even though MoKi is a web-based application, it involves an installation process, as it is built upon a wiki platform that should be setup.

[19] http://jena.apache.org/documentation/tools/eyeball-getting-started.html (Last visit 09/09/2015)
[20] http://jena.apache.org/ (Last visit 09/09/2015)
[21] https://moki.fbk.eu/website/index.php (last visited on the 28th October, 2015)

The MokiValidation plug-in contains four validation modules, namely, the ontology questionnaire, assertional effects, the models' checklist and the quality indicator. These modules aim to give feedback on how the models can be improved rather than rate the quality of the models on a numerical scale.

The ontology questionnaire displays inferences from the domain model by supporting the process of reviewing inferences for ontology evaluation purposes. The assertional effects module displays assertional effects after modifications of a knowledge base. These two modules (ontology questionnaire and assertional effects) support ontology evaluation based on the formal interpretation of a model, and use reasoning services.

The other two validation modules, the models' checklist and the quality indicator, might be considered as implementations of modelling guidelines based on heuristics. Given general guidelines on good practices in modelling, such as, for example, "model elements should be well documented", the models' checklist lists elements that do not comply to the modelling guidelines, and the quality indicator module visualizes how well a given element complies with the modelling guidelines. These two modules can be seen as two views (local and global) of the same indicators.

The models' checklist contains a set of checks concerning characteristics that typically, but not always, point to oversights and modelling mistakes. The list of checks done by the system is:

- Concepts without verbal description
- Orphaned concepts
- Concepts without individuals
- Non-shared concepts
- Individuals with no type defined
- Properties without verbal description
- Properties with no domain defined
- Properties with no range defined
- Non-shared properties

2.5.6 XD-Analyzer

The XD-Tools (Daga et al., 2010) is a plug-in[22] for the ontology editor NeOn Toolkit[23] for supporting the XD method (Presutti et al., 2012) (see Section 2.3.2.4). One of the functionalities of this plugin is called XD-Analyzer, which provides end-user support for good practices in pattern based ontology design. XD-Analyzer provides the following types of checks from the most to the least important in terms of ontology quality: "error", "warning" and "suggestion". This system works for OWL ontologies and the list of checks that it provides is as follows:

- Domain or range intersection (error). The domain or range of a property contains an intersection of classes. Sometimes the intended axiom is the union of classes.
- Missing type (error). Each entity (a class, a property, an individual) must be the instance of something. This is valid for entities in the T-Box as well as in the A-Box.
- Missing comment (warning). All entities should have at least one *rdfs:comment*.
- Missing label (warning). All entities should have at least one *rdfs:label*.
- Missing inverse (warning). Each object property should have an inverse property (except for symmetric properties).
- Unused imported ontology (warning). All imported ontologies should have at least one locally referenced entity,
- Isolated entity (warning). Each entity must be related at least to another one through some ontology axiom.
- Architectural import notice (suggestion). Most of the locally defined entities do not specialise imported entities.
- Missing domains or ranges (suggestion). Each property should have its domain and range properly defined.

No architectural description of the system has been found apart from the fact that it uses the OWL API (SVN version), it is implemented as Eclipse Plug-in, and that

[22] http://neon-toolkit.org/wiki/XDTools (last visited on the 28th October, 2015)
[23] http://neon-toolkit.org/wiki/Main_Page (last visited on the 28th October, 2015)

it follows good practices such as modularization and externalization of dependency libraries.

2.5.7 OQuaRE

OQuaRE (Duque-Ramos et al., 2011) is a web application[24] that extracts quality measurements from the ontology structure and compares them to certain predefined values in order to support the OQuaRE approach (see Section 2.4.10). The measurements used by the method are: lack of cohesion in methods, weighted method count, depth of subsumption hierarchy, number of ancestor classes, number of children, coupling between objects, response for a class, number of properties, properties richness, attribute richness, relationships per class, class richness, annotation richness, and tangledness.

While this system provides numerous metrics for ontology characteristics, it does not directly identify specific ontology quality issues.

2.5.8 OntoCheck

OntoCheck (Schober et al., 2012), a plug-in[25] for the Protégé editor that helps to clean up an ontology with regard to its lexical heterogeneity. This is done by enforcing naming conventions and metadata completeness.

For the metadata completeness check, the system provides the "Check panel", which allows the user to specify which annotation properties to check. It also allows to verify whether a particular orthographic or morpho-syntactic naming convention is fulfilled in a selected subtree by letting the user choose: a) word case; b) word separator; c) the presence or absence of regular expressions; and d) minimum and maximum character and word count. The naming patterns can be stored in an external file, so that developers can check them repeatedly and share the list among each other.

Finally, OntoCheck provides a "compare panel", which allows to compare the values for the specified labelling and metadata entities, and a "statistics panel", which detects and quantifies otology measures useful for complexity analysis, ontology evaluation and process monitoring. This latter panel detects, for instance, isolated entities.

[24]The OQuaRE web application is available at http://miuras.inf.um.es:9080/oqmodelsliteclient/ (last visited on the 7th September, 2015).

[25]http://protegewiki.stanford.edu/wiki/OntoCheck (last visited on the 28th October, 2015)

The OntoCheck Java plug-in was implemented for the Protégé 4.1 ontology editor using the Protégé OWL API under Eclipse. The download link from OntoCheck's website[26] does not currently work.

2.6 Conclusions

A graphical summary of the reviewed works of this chapter is provided in Figure 2.12, where (a) ontology development methodologies and lightweight approaches (top of the figure), (b) methodological works about ontology evaluation (center of the figure) and (c) existing tools (bottom of the figure) are shown in chronological order. Relations between methodologies, methods and tools for ontology evaluation are explicitly displayed in the figure. In addition, for each ontology development methodology or lightweight approach, a dashed box attached to it summarizes how the given item addresses the ontology evaluation activity.

Conclusion 1: The analysis of the state of the art allows us to conclude that **methodologies for building ontologies** that address ontology evaluation have traditionally relied on the competency questions technique to address such activity, since Grüninger and Fox proposed them in 1995. Some methodologies consider other techniques as well. For example, in METHONTOLOGY, the first definitions for ontology evaluation and related terms were provided as well as a list of common errors in taxonomies; ON-TO-KNOWLEDGE considered application based evaluation in combination with the competency questions techniques even though no specific guidelines were provided; and RapidOWL only focuses on consistency checking. Finally, the NeOn Methodology compiles existing approaches and techniques in the state of the art.

Conclusion 2: Most of the methodologies for building ontologies pay little attention to the ontology evaluation activity in terms of detailed guidelines for supporting it. These methodologies do not cover more than two dimensions, for example, domain coverage or syntax validation. The most complete methodology regarding ontology evaluation is the NeOn Methodology, which compiles a number of approaches and relies on the original work for specific guidelines.

Conclusion 3: Regarding the **methods and frameworks for ontology evaluation**, we can observe that two of the presented works, namely Gómez-Pérez (Gómez-

[26] http://www2.imbi.uni-freiburg.de/ontology/OntoCheck/

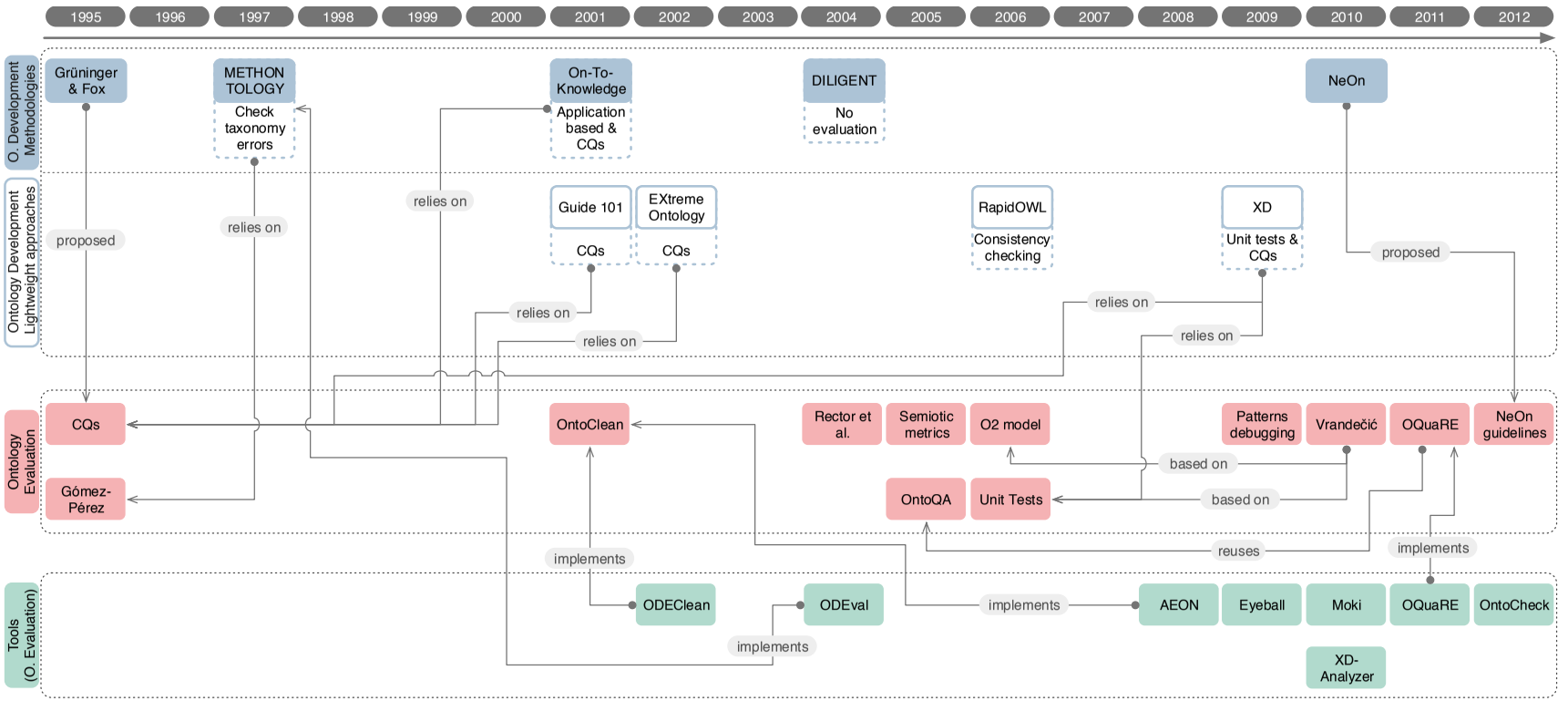


Figure 2.12: Chronological summary of ontology development methodologies, ontology evaluation frameworks, methods and tools.

Pérez, 1995) and the OntoClean methodology (Guarino and Welty, 2009; Welty and Guarino, 2001), only deal with the taxonomical knowledge of the ontology. These works do not address either any other quality problem related to other types of knowledge represented in the ontology or any other dimensions, as for example, the application context.

Conclusion 4: Some approaches provide a wide range of dimensions, criteria and metrics but no specific guidelines or methods. For those cases where topological measurements about the structure of the ontology are given, for example in (Duque-Ramos et al., 2011), no concrete ontology diagnosis output nor pointers to specific errors are offered.

Conclusion 5: A few methods are more oriented towards the identification of patterns, or antipatterns, as is the case of this thesis. These similar approaches are, for example, the work proposed by in (Rector et al., 2004), (Corcho et al., 2009) and (Vrandečić, 2010). The antipatterns and common problems proposed in these works can be still extended and complemented by means of analysing different ontology quality dimensions.

Conclusion 6: Regarding the **tools for supporting ontology evaluation** for OWL ontologies, we can say that existing tools suffer from one or more of the following weaknesses:

1. They are developed as plug-ins for desktop applications (for example, XD-Analyzer and OntoCheck), which implies that the user must install both the tool and the plug-in for ontology evaluation. In this case, it happens that the plug-in might be outdated, and sometimes incompatible, as long the core desktop applications evolve to new versions. This is the case of XD-Tools, whose last developed version is compatible from NTK 2.3.2 to NTK2.4.2, while the last version of NeOn Toolkit is NTK2.5.

2. Tools that are based on wiki technologies (such as MoKi) force an installation process to set up the wiki system and extensions.

3. Some tools limit their activity to finding a number of metrics but they do not detect specific modelling issues to be solved, as it is the case of OQuaRE.

4. Users need a high technological background to install and use them, as they are designed to be included in third party software, as it is the case of Eyeball.

Conclusion 7: In most of the analysed evaluation tools the number of automatically detected problems is considerably limited and oriented to one or few ontology evaluation dimensions, for example, oriented only to ontology metadata, syntax validation, logical consistency or taxonomical knowledge.

In summary, we can state that there is an important gap between the number of theoretical and methodological works about ontology evaluation and the number of tools supporting them. To the best of the authors' knowledge, the contribution of this thesis provides (a) a wider range of identified common problems for diagnosing ontologies (b) tips for repairing them and (c) detection methods together with (d) the technological support for diagnosing OWL ontologies.

Chapter 3

Goals and contributions

The goal of this thesis is to investigate methods and tools for ontology evaluation. With this thesis we have contributed to the state of the art in such field, more precisely in the ontology diagnosis activity. This chapter presents the objectives pursued along this thesis and the resulting contributions. It includes also the assumptions, hypothesis and restrictions that define and delimit the scope of the thesis. The research methodology followed during the development of this thesis is also described.

3.1 Goals

The overall goal of this thesis is **to advance the current state of the art in the ontology evaluation field, specifically in the ontology diagnosis activity**. For doing so, we pretend to assist ontology developers when evaluating ontologies lessening their effort during this activity by means of extending existing approaches. In order to attain this general goal, the following subgoals should be pursued:

O1: To help ontology engineers to diagnose their ontologies in order to find common pitfalls.

O2: To ease the ontology diagnosis activity by means of providing suitable technological support, lessening thus the effort required from ontology engineers.

3.2 Contributions

Along this thesis, we intend to provide solutions to the conceptual and technological open problems identified in Chapter 2.

With regard to the conceptual objective the contributions are:

C1: Catalogue of common pitfalls for ontology diagnosis.

C2: Quality model of ontology diagnosis.

Regarding the technological objective, the contributions are:

C3: Design and implementation of detection methods for the pitfalls defined in the catalogue whenever it is possible.

C4: OOPS! (OntOlogy Pitfall Scanner!): a tool for ontology diagnose that integrates the methods for pitfall detections providing human and machine oriented interfaces.

3.3 Assumptions

The work presented in this thesis is based on the following set of assumptions:

A1: The catalogue of pitfalls proposed in this thesis is not exhaustive.

A2: New pitfalls could appear and could be added to the catalogue in the future.

A3: During the ontology diagnosis activity, an ontology network or an ontology that reuses parts from other ontologies, is considered as a unique ontology.

A4: Two or more anonymous[27] ontologies executed with OOPS! are considered the same ontology if their evaluation results are equal. This is taken into account when analysing OOPS!'s statistics in order to avoid duplicates.

[27] An ontology is considered to be anonymous if its URI is not defined.

3.4 Hypothesis

After having identified the assumptions in which this thesis is based on, the hypothesis are:

H1: Systematic approaches to evaluate ontologies based on a list of common pitfalls improve the quality of ontologies.

H2: A collection of methods for detecting pitfalls in a (semi)automatic way facilitates the ontology diagnosis activity.

3.5 Restrictions

The following restrictions define the limits of our contributions and may establish future lines of research:

R1: The catalogue of pitfalls proposed in this thesis addresses OWL DL ontologies.

Note: for some pitfall detection methods OWL 2 features have been included. These cases are included due to users' requests.

R2: Only the diagnose of the conceptual level (T-box) of ontologies is addressed in this work. Techniques for diagnosing the data level (A-box) are out of scope.

Note: SKOS implementations are out of scope of this work. This is a particular case of R2. A given SKOS model is considered an instantiation (A-box) of the SKOS ontology.

R3: Techniques making use of the data level (A-box) for diagnosing the conceptual level (T-box) are out of scope.

Note: This restriction applies to methods implemented in this thesis but does not apply to third party software being reused.

R4: Technological support for the ontology repair activity is not a goal of this thesis. However, indications about potential repair actions are given but not automated.

R5: OOPS! uniquely takes as input an ontology written in OWL that are syntactically correct according to turtle or RDF/XML syntax.

R6: OOPS! does not take as input any kind of ontology associated information nor documentation, for example, the ontology requirement specification document, background knowledge, data, etc.

R7: OOPS! does not perform inference. OOPS! checks only the explicit knowledge represented in the ontology.

R8: Only English language is supported for the detection methods in which natural language processing functionalities are required.

R9: For processes that involve linguistic knowledge we rely on WordNet.

R10: For processes regarding namespace hijacking we rely on TripleChecker.

R11: For processes regarding the detection of licenses in ontologies we rely on Licensius.

Figure 3.1 summarizes the mapping between the objectives identified in Section 3.1 and the contributions developed in this thesis (Section 3.2). The figure also includes, for each contribution, relations to the associated assumptions (Section 3.3), hypothesis (Section 3.4) and restrictions (Section 3.5) of the thesis.

3.6 Research methodology

This section presents an overview of the research methodology followed during the development of this thesis. The iterative and incremental process followed is detailed in Section 3.6.1.

In order to address our general research problem of **advancing the current state of the art in the ontology evaluation field, specifically in the ontology diagnosis activity**, we have followed a "divide and conquer" strategy. In this way, the general problem is decomposed into different subproblems. Next, in order to solve each subproblem, different alternatives and solutions are provided. Finally, the solution to the general problem is drawn by combining the solutions to the different subproblems.

The subproblems identified are:

1) The need of providing a systematic approach to address the ontology diagnosis activity.

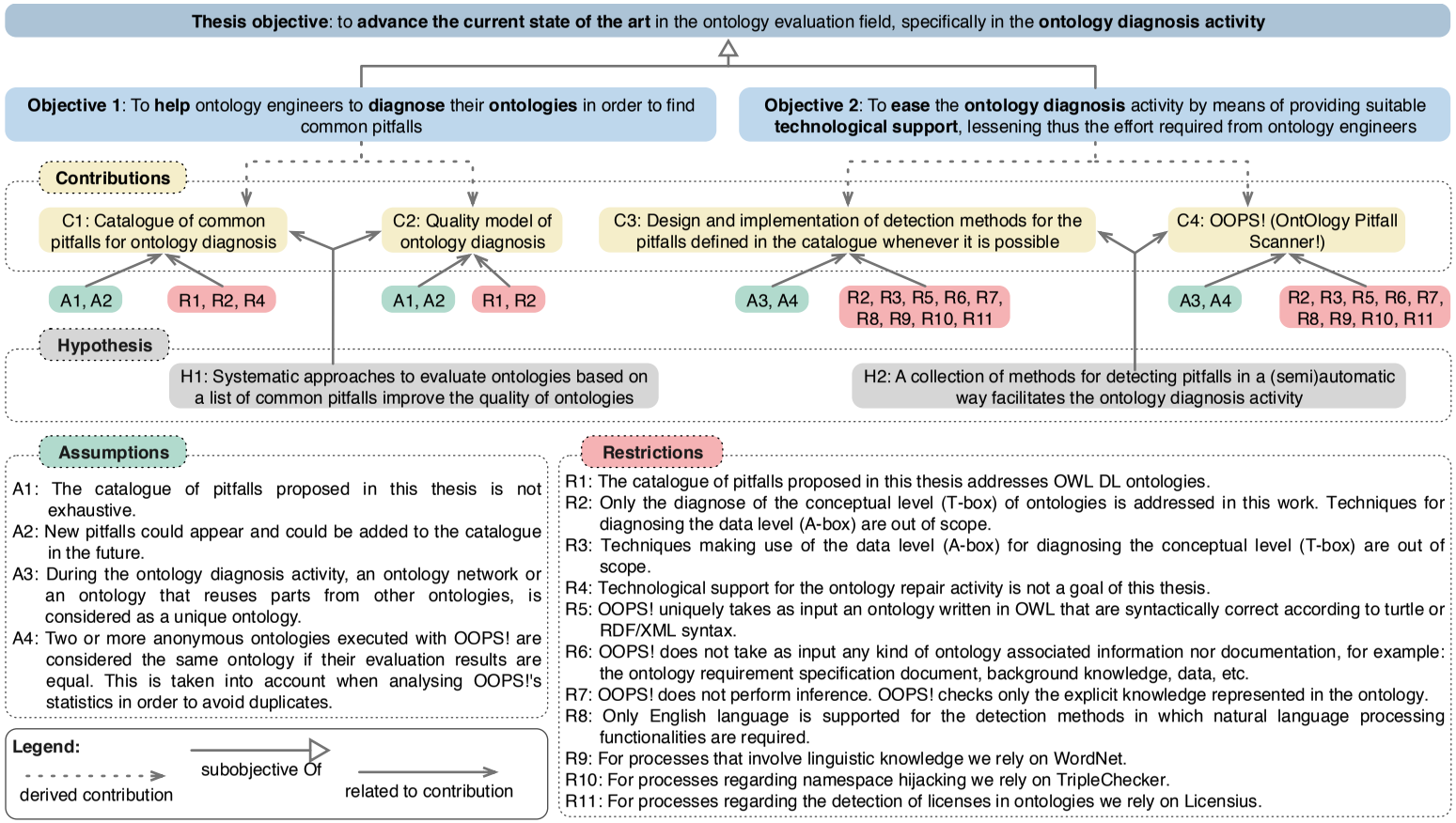


Figure 3.1: Correspondences between objectives, contributions, assumptions, hypothesis and restrictions.

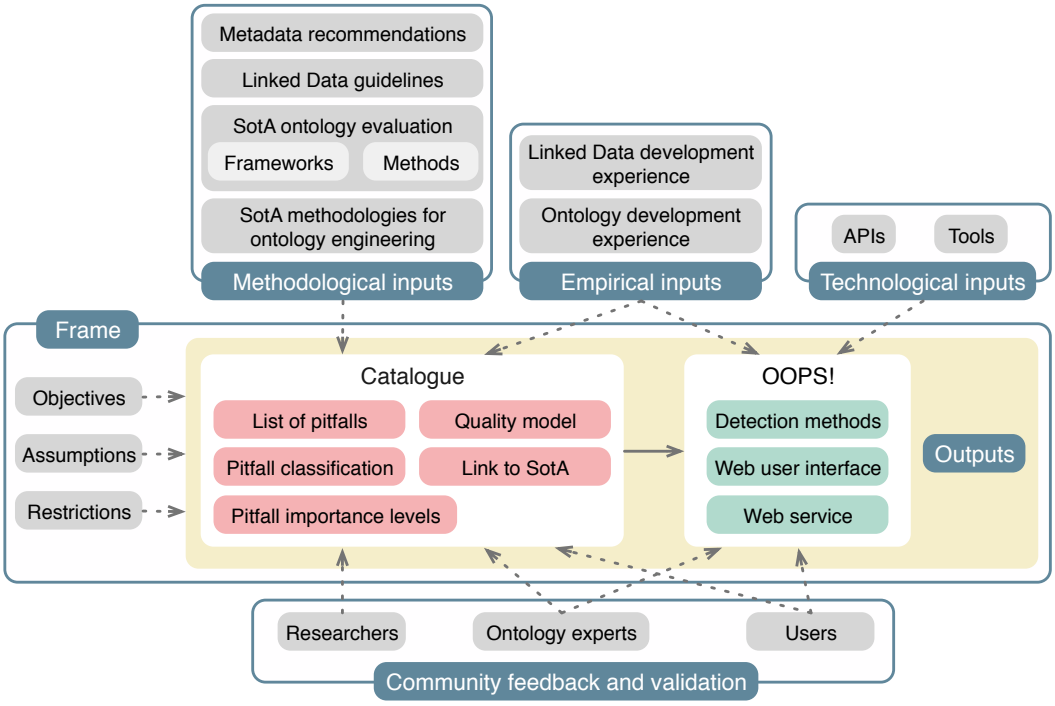


Figure 3.2: Inputs and context taken into account for creating the pitfall-based approach to ontology diagnose.

- To approach this subproblem we developed the common pitfall catalogue (C1) for addressing the ontology diagnose activity and the quality model (C2). These contributions are described in Chapter 4.

2) The need of technological support to ease the ontology diagnosis activity.

- To approach this subproblem we developed the list of pitfall detection methods (C3) and the system called OOPS! (C4). These contributions are described in Chapter 5

For obtaining the catalogue, the quality model, and the detection methods that are implemented within the system, the state of the art and empirical inputs were taken into account as shown in Figure 3.2. Such figure also represents the community based feedback and validation component that guided the thesis. In addition, the figure reminds that the work done is framed by a set of goals, restrictions and assumptions previously presented in this chapter.

For creating the pitfall catalogue and the quality model we took into account both methodological and empirical inputs, as Figure 3.2 represents. Such inputs are:

1. *Existing methodologies in Ontological Engineering.* From METHONTOLOGY (Fernández-López et al., 1997; Fernández-López et al., 1999), the NeOn Methodology (Suárez-Figueroa et al., 2015), and the "Ontology Development 101" guide (Noy and McGuinness, 2001) we have extracted and taken as reference good practices and design guidelines.

2. *Existing methods to ontology evaluation.* This thesis is grounded on the following methods and approaches to ontology evaluation:

   - *Frameworks*: In this thesis we take into account the framework and dimensions provided in (Gangemi et al., 2006) as well as the evaluation framework and ontology aspects presented in (Vrandečić, 2010).
   - *Methods*: Regarding works on ontology evaluation, we reviewed, reused, and included in the pitfall catalogue outcomes taken: (i) from (Rector et al., 2004), in which the authors describe a set of common errors made by developers during the ontology modelling activity; (ii) from (Gómez-Pérez, 2004), in which she provides a classification of errors identified during the evaluation of consistency, completeness, and conciseness of ontology taxonomies; (iii) from (Noy and McGuinness, 2001), where they present a methodology for creating ontologies and point out some common errors and how to avoid them; and (iv) from (Vrandečić, 2010) in which a set of methods for ontology evaluation are provided.

3. *Metadata recommendations*: We have also reused existing works in different and important aspects of ontology metadata. In this sense we rely on the work described in (Rodríguez-Doncel et al., 2013) for licensing ontologies; the recommendations for ontology metadata presented in (Vandenbussche and Vatant, 2012), and the guidelines provided in (Montiel-Ponsoda et al., 2011) and (Aguado-De Cea et al., 2015) for generating natural language annotations and naming.

4. *Existing guidelines for Linked Data development.* We took as reference existing guidelines about Linked Data publication and consumption (Heath and Bizer,

2011); studies about problems identified for accessing RDF on the Web (Hogan et al., 2010); works on publishing vocabularies with content negotiation (Berrueta et al., 2008); and guidelines for creating persistent URIs (Archer et al., 2012). From these works, we abstracted the issues that could be applied to ontology development and publication on the Web.

5. *Previous practices and experience.* Previous experience of the PhD candidate in ontology development within several national projects as GeoBuddies (TSI2007-65677-C02), mIO![28] (CENIT-2008-1019) and BuscaMedia[29] (CENIT-2009-1026) and international projects as the "Development of the Ontology Network Specification Requirements and Conceptualization" in collaboration with the World Health Organization provided inputs on ontology engineering methodologies, techniques and tools. Also, the participation in the NeOn project[30] (FP6-027595) and during the development and publication of the AEMET[31] linked dataset (Atemezing et al., 2013) have provided valuable experience and knowledge. We made a retrospective analysis of the processes and activities performed within such projects to get a preliminary set of informal steps, and positive and negative lessons learnt. Such lessons were refined and completed to draw the ontology evaluation approach, including a catalogue for ontology diagnosis and preliminary ideas for repairing ontologies.

For the development of the detection methods and OOPS!, we were grounded on:

1. *Previous practices and experience.* Along the experience of the PhD candidate in the development of ontologies and Linked Data publication projects, the available ontology evaluation systems and the needs regarding the ontology evaluation activity were analysed. Main problems found regarding existing tools were: (a) the low coverage of errors detections; (b) the dependence of specific ontology editors; (c) the complexity of creating ad-hoc tests; and (d) the deep knowledge about programing or query languages needed to use some of the existing systems.

[28] http://www.cenitmio.es (last visited on the 24th November, 2015)
[29] http://www.cenitbuscamedia.es (last visited on the 24th November, 2015)
[30] http://www.neon-project.org (last visited on the 12th October, 2015)
[31] http://aemet.linkeddata.es (last visited on the 12th October, 2015)

2. *Available software.* In this case, we reused available APIs for parsing RDF as JENA, web technologies and languages, linguistic resources as WordNet,[32] as well as web services available like TripleChecker[33] and Licensius.[34]

3.6.1 Description of the research process

Along this thesis an iterative and incremental process has been followed, leading to five phases, namely: compilation, extension, implementation, Linked Data extension and refinement. Figure 3.3 shows an overview of main contributions of each phase including: (a) the order in which the phases were carried out; (b) the degree of development of the pitfall catalogue; (c) the degree of the development of detection methods and their implementation within OOPS!; (d) the provenance of the pitfalls that were included in the catalogue; and (e) main publications obtained from each phase.

As Figure 3.3 pretends to be self-contained, the citations used within the boxes, represented by integers between brackets, correspond to the enumerated bibliographical resources at the bottom of the figure. The bibliographical information at the end of each reference in Figure 3.3, includes the citations being used along this thesis. As an example, in Figure 3.3, reference "[1]" corresponds to "(Poveda-Villalón et al., 2009)" in the rest of the document.

It can be observed that, due to the fact that the catalogue drives the development of OOPS!, first phases were focuses on the development and evolution of such catalogue. However, since the third phase, iterations over the pitfall catalogue and the technological support are done in parallel as both contributions are closely related.

During the **first phase (Compilation)**, we started analysing ontologies in order to provide a compilation of potential problems or errors when modelling ontologies. The result of such analysis was presented in 2009 in (Poveda-Villalón et al., 2009). At that point, 14 pitfalls were identified and classified according to the Ontology Design Patterns initiative described in (Presutti et al., 2008). Such pitfalls were identified by manual ontology inspection. It should be mentioned that this phase took place before the author enrolled in the doctoral program.

[32] http://wordnet.princeton.edu/ (last visited on the 21st September, 2015)
[33] http://graphite.ecs.soton.ac.uk/checker/ (last visited on the 21st September, 2015)
[34] http://licensius.appspot.com/ (last visited on the 21st September, 2015)

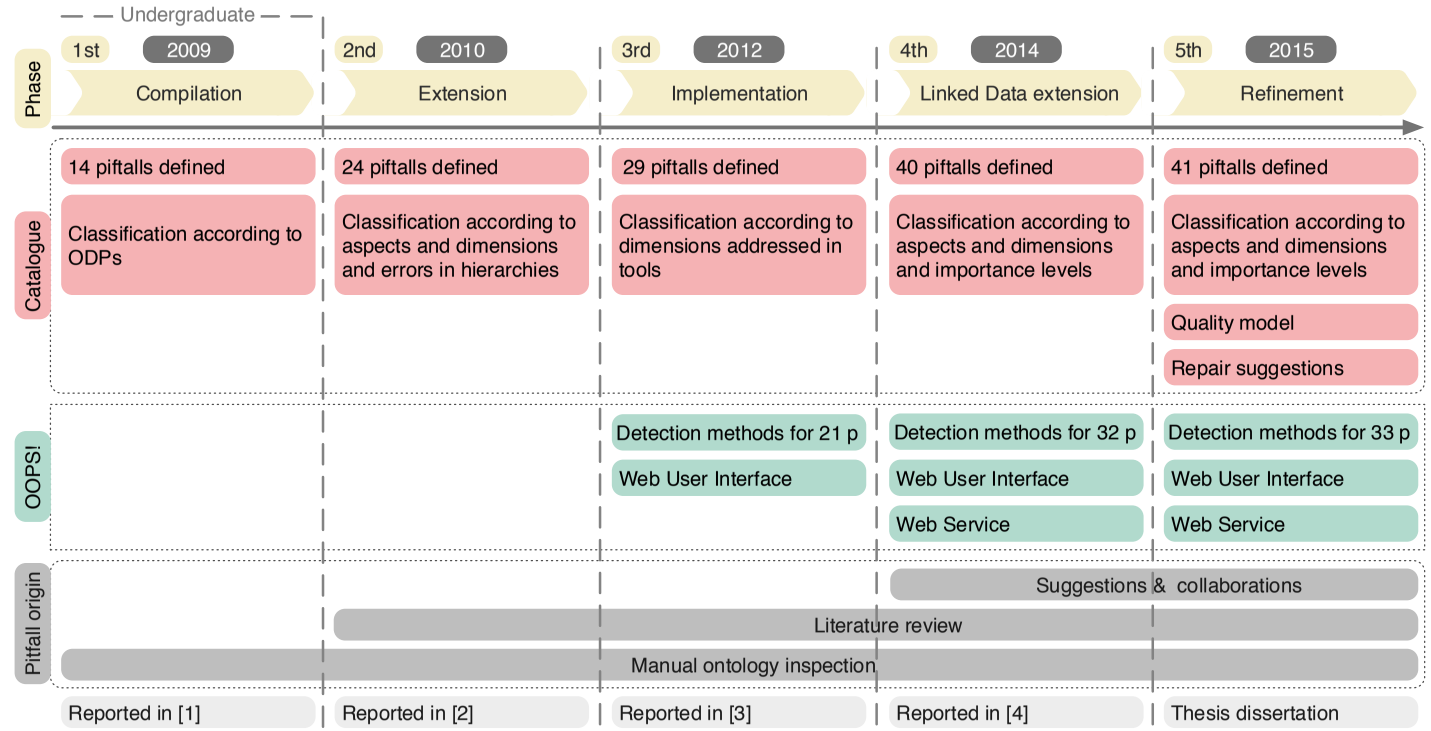


Figure 3.3: Phases of the thesis development including: catalogue evolution; OOPS! evolution; origin of the pitfalls; and references to main publications derived from the work done.

In the **second phase (Extension)**, we focused on the extension of the catalogue and the harmonization with existing works. We included also pitfalls extracted from the literature review (Gómez-Pérez, 2004; Noy and McGuinness, 2001; Rector et al., 2004). In 2010, a catalogue of 24 pitfalls were reported in (Poveda-Villalón et al., 2010) following a classification according to the dimensions proposed in (Gangemi et al., 2006) and aspects of ontology evaluation; and according to ontology evaluation criteria defined in (Gómez-Pérez, 2004).

During the **third phase (Implementation)**, the first version of the detection methods were elaborated and implemented within the first release of OOPS! in 2012 as reported in (Poveda-Villalón et al., 2012). This system addressed the detection of 21 pitfalls out of the 29 included in the catalogue at that moment.

Along the **fourth phase (Linked Data extension)** a new release of the catalogue was published in 2014 (Poveda-Villalón et al., 2014) where 40 pitfalls were defined and 32 detection methods were implemented. The main contribution of this extension of the pitfall catalogue was inspired in Linked Data principles and some development projects. This version included both the web user interface and a web service for the detection methods implemented. Apart from classifying the pitfalls according to aspects and dimensions described in (Gangemi et al., 2006), a classification according to the importance level of the pitfalls was also provided (see Section 4.3 for further details). In order to assign these importance levels to the pitfalls a survey involving the participation of ontology experts, researchers in the semantic web community and OOPS! users were carried out.

The **fifth phase (Refinement)** was devoted to defining a quality model for ontology diagnose extending existing quality models. We included suggestions for manual ontology repair. This phase also included refinement activities and continuos improvement of the pitfalls descriptions, the pitfall detection methods and OOPS!. At the end of this phase 41 pitfalls were defined and 33 of them were implemented into OOPS!.

The catalogue does not pretend to be an exhaustive, rigid and fixed checklist, as stated in assumptions **A1** and **A2** (see Section 3.3). New pitfalls might be included by ontology experts or user's suggestions or as consequence of the catalogue evolution and maintenance. The workflow for including new pitfalls in the catalogue is detailed in Section 4.5.

Chapter 4

Catalogue for ontology diagnosis

4.1 Introduction

One common approach for evaluating almost any type of software component, apart from verification tests, is to have a checklist of typical errors that other developers have made before. Thus, the developers check the product being built against such a list, detect errors, and correct them, within a diagnosis and repair loop. This thesis does not pretend to start from scratch a checklist for ontology diagnosis, but to reuse existing works where modelling problems have been already identified and to extend them by incorporating new pitfalls obtained through an empirical evaluation of existing ontologies. In addition, this thesis includes the development of technological support in order to cover the ontology diagnosis activity, based on such catalogue of pitfalls.

While the development process of the catalogue and the technological support was described in Section 3.6.1, this chapter focuses on describing the catalogue itself and its components. Section 4.2 exposes in detail the current state of the catalogue. Since not all pitfalls in the catalogue have the same importance, Section 4.3 describes how importance levels have been assigned to the pitfalls. A classification of the pitfalls according to ontology evaluation dimensions and aspects is provided in Section 4.4. Section 4.5 presents the workflow followed for including new pitfalls in the catalogue. Finally, Section 4.6 shows the quality model drawn behind the catalogue.

4.2 Ontology pitfall catalogue

This section describes the complete list of 41 pitfalls included in the catalogue. This list is not exhaustive, according to assumptions 1 and 2 (see Section 3.3). Other pitfalls might be included in the future. A workflow to include new pitfalls has been explained in Section 3.6.1.

All pitfalls included in this thesis are described according to the pitfall description template shown in Figure 4.1 that contains the following fields (mandatory fields are marked with an * in the template):

<table>
<tr><td>Title*</td><td colspan="2">Code and title of the pitfall</td><td>Importance level*</td><td>{Critical |Important |Minor}
See Section 4.3.</td></tr>
<tr><td>Aspects*</td><td colspan="2">Ontology evaluation aspects related to the pitfall. See Section 4.4 for further details and possible values.</td><td>Affects to*</td><td>{Ontology |[Classes, Object properties, Datatype properties]}</td></tr>
<tr><td colspan="5">Description*</td></tr>
<tr><td colspan="5">Detailed explanation of what the pitfall consists in. This field might contain references to other approaches.</td></tr>
<tr><td colspan="5">Examples*</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="3">Natural language description</td></tr>
<tr><td colspan="2">Graphical representation of the example following: (a) the adaptation of the UML_Ont profile defined for ontologies in the NeOn Deliverable D1.1.2 (Haase et al., 2009); or (b) source OWL encoding in functional syntax, depending on the suitability of each representation technique for each case.
In case this field includes the representation of lack of information, such information is indicated by "No explicit evidence of:" preceding the given statements.</td><td colspan="3">Natural language explanation of the graphical example or OWL code.</td></tr>
<tr><td colspan="5">How to solve it*</td></tr>
<tr><td colspan="5">Description of the actions to be taken by ontology engineers in order to solve the described pitfall.</td></tr>
<tr><td>References</td><td colspan="4">Pointers to related bibliographical references or links to URLs, if needed. This is an optional field.</td></tr>
</table>

Figure 4.1: Pitfalls description template

- **Title**: this field represents the identifier of the pitfall. It is composed by a pitfall code P, followed by a numerical identification, and a title that briefly describes what the pitfall consists in. The ordinal identification has no further meaning that identifying the pitfalls unambiguously. The code assigned to each pitfall in this section is used along this thesis.

- **Importance level**: this field provides an indicator of how crucial is the appearance of a pitfall regarding the ontology quality and its functionality. The possible values are: Critical, Important and Minor. In order to understand completely how the values for this field have been assigned, we refer the reader to Section 4.3

- **Aspects**: this field indicates the ontology evaluation aspects in which the pitfall is classified. The possible values are described in Section 4.4.

- **Affects to**: this field indicates whether the pitfall affects to specific ontology elements or whether the pitfall affects the ontology itself, as a whole. As the pitfalls described in this thesis affect uniquely to the T-Box (Restriction R2 in Chapter 3), pitfalls might affect the ontology itself or its classes, object properties or datatype properties. We do not include in this thesis pitfalls for diagnosing individuals.

- **Description**: this field provides an explanation of what the pitfall consists in. Such explanation might be completed with references to existing research work that are included in the field "References".

- **Example**: this field includes an example in which the pitfall could occur. To describe the example, we use the following two subfields:

  - **Graphical representation and/or OWL code**: depending on the suitability of each representation technique, this field includes: (a) a graphical representation following the UML_Ont profile defined for ontologies, or (b) the source OWL encoding in functional syntax. This field might also contain excerpts of OWL code for indicating the lack of information or metadata aspects. When the example involves lack of information it is indicated by "*No explicit evidence of:*". Along this field, namespaces for ontologies are only indicated if such information is relevant for the pitfall description.
  - **Natural language description**: this field provides a natural language explanation of the example shown in the "Graphical representation and/or OWL code" field.

- **How to solve it**: this field provides indications about which actions could be taken by ontology engineers in order to solve the given pitfall. It might include

several alternatives, even though they cannot be considered exhaustive. These indications might be completed with references to existing works that are included in the field "References".

- **References**: this field is optional and it provides the bibliographical references or links to URLs mentioned along the rest of the template fields. The notation used for references in the template is a cardinal number between brackets. As this is an optional field, it is shown in the table describing a given pitfall if there is a value to be included in it. Otherwise the field will not be shown in the table.

It should be mentioned that the original UML_Ont profile (Haase et al., 2009) utilizes custom stereotypes and dependencies to cover OWL 1 constructs. In this thesis, we align the stereotypes used in the profile to OWL and RDF(S) constructs as shown in Figure 4.2. Alternatives and additional notations for properties, axioms and individuals are defined in (Haase et al., 2009), and we refer the reader to such reference for additional information. In the following tables, and along the rest of this section[35] the ontology elements mainly used will be the ones listed and depicted in Figure 4.3, Figure 4.4, and Figure 4.5. For the sake of readability, specific element notations in the mentioned figures are labelled in correspondence to the enumeration items listed below:

1) **Classes**: the graphical representations for classes, class restrictions and class axioms are depicted in Figure 4.3. The constructs included in such figure are:

   1.a) **Named classes** are represented by labelled boxes.

   1.b) **Class restrictions or anonymous classes** are represented by empty boxes.

   1.c) **Universal restrictions** are represented by means of the «owl:allValuesFrom» stereotype together with the property on which the restriction is applied.

   1.d) **Existential restrictions** are represented by means of the «owl:someValuesFrom» stereotype together with the property on which the restriction is applied.

   1.e) **Intersection class** descriptions could be represented by means of:

      1.e.i) Empty circle together with the «owl:intersectionOf» stereotype.

      1.e.ii) Icon including the symbol "⊓".

[35] These notation will be reused and slightly adapted due to technical issues in Chapter 5.

UML_Ont profile	⟷	OWL primitives adaptation
ObjectAllValuesFrom	⟷	owl:allValuesFrom
ObjectSomeValuesFrom	⟷	owl:someValuesFrom
ObjectIntersectionOf	⟷	owl:intersectionOf
ObjectUnionOf	⟷	owl:unionOf
SubClassOf	⟷	rdfs:subClassOf
EquivalentClasses	⟷	owl:equivalentClass
DisjointClasses	⟷	owl:disjointWith
ObjectPropertyDomain	⟷	rdfs:domain
DataPropertyDomain	⟷	rdfs:domain
ObjectPropertyRange	⟷	rdfs:range
DataPropertyRange	⟷	rdfs:range
EquivalentObjectProperties	⟷	owl:equivalentProperty
EquivalentDataProperties	⟷	owl:equivalentProperty
InverseObjectProperties	⟷	owl:inverseOf
Transitive	⟷	owl:TransitiveProperty
Symmetric	⟷	owl:SymmetricProperty
ClassType	⟷	rdf:type
not	⟷	owl:complementOf

Figure 4.2: Correspondences between UML_Ont profile stereotypes defined in (Haase et al., 2009) and OWL and RDF(S) constructs.

1.f) **Union class** descriptions could be represented by means of:

1.f.i.) Empty circle together with the «owl:unionOf» stereotype.

1.f.ii.) Icon including the symbol "⊔".

1.g) **Subclass of** axioms could be represented by means of:

1.g.i) Generalization arrow.

1.g.ii) UML dependency arrow with the «rdfs:subClassOf» stereotype.

1.h) **Equivalent class** axioms could be represented by means of:

1.h.i) Double-sided UML dependency with the «owl:equivalentClass» stereotype.

1.h.ii) Circle including the symbol "≡".

1.i) **Disjoint class** axioms could be represented by means of:

1.i.i) Double-sided UML dependency with the «owl:disjointWith» stereotype.

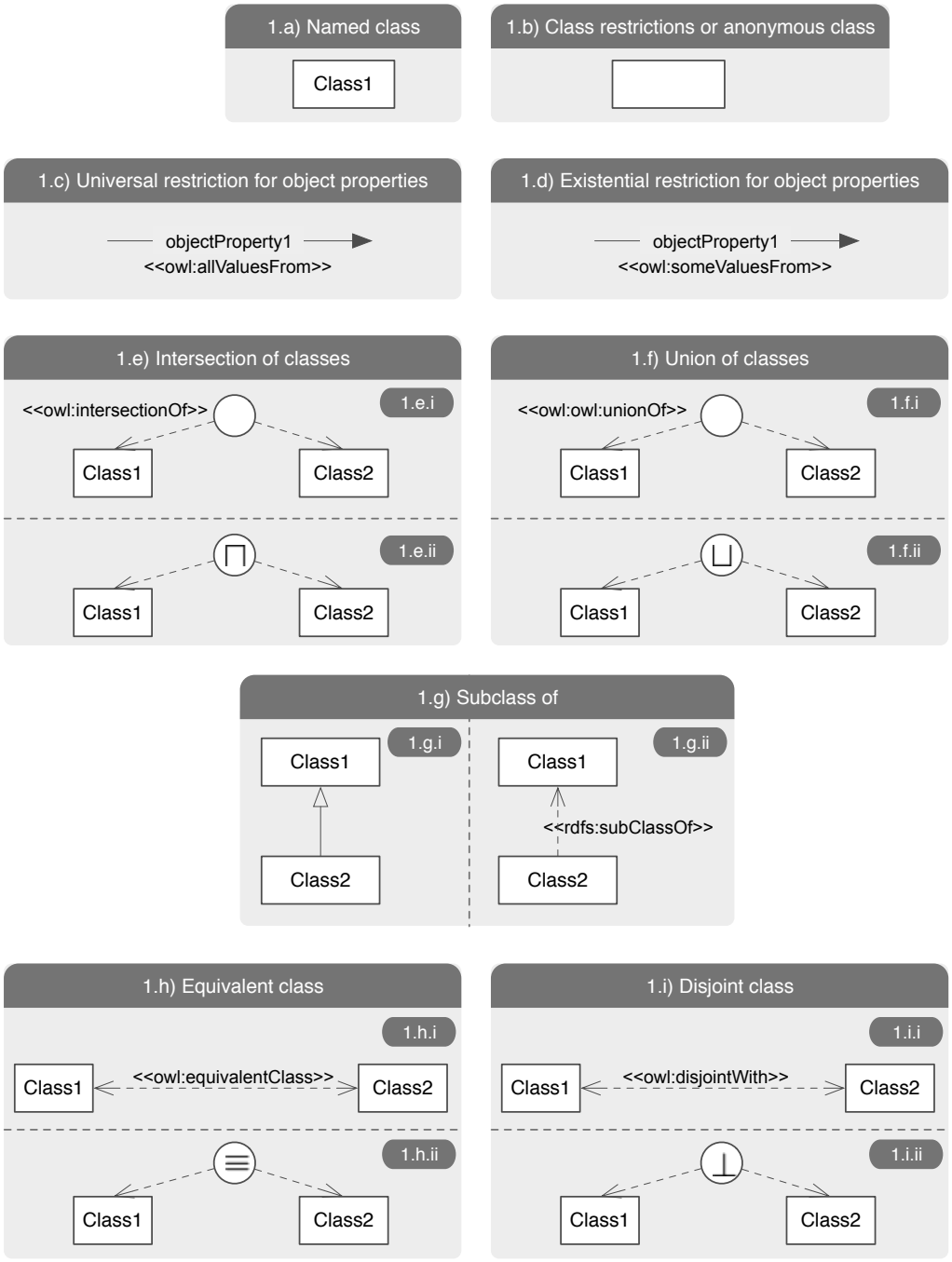


Figure 4.3: Notation for classes, class restrictions and class axioms. Adapted from (Haase et al., 2009)

1.i.ii) Circle including the symbol "⊥".

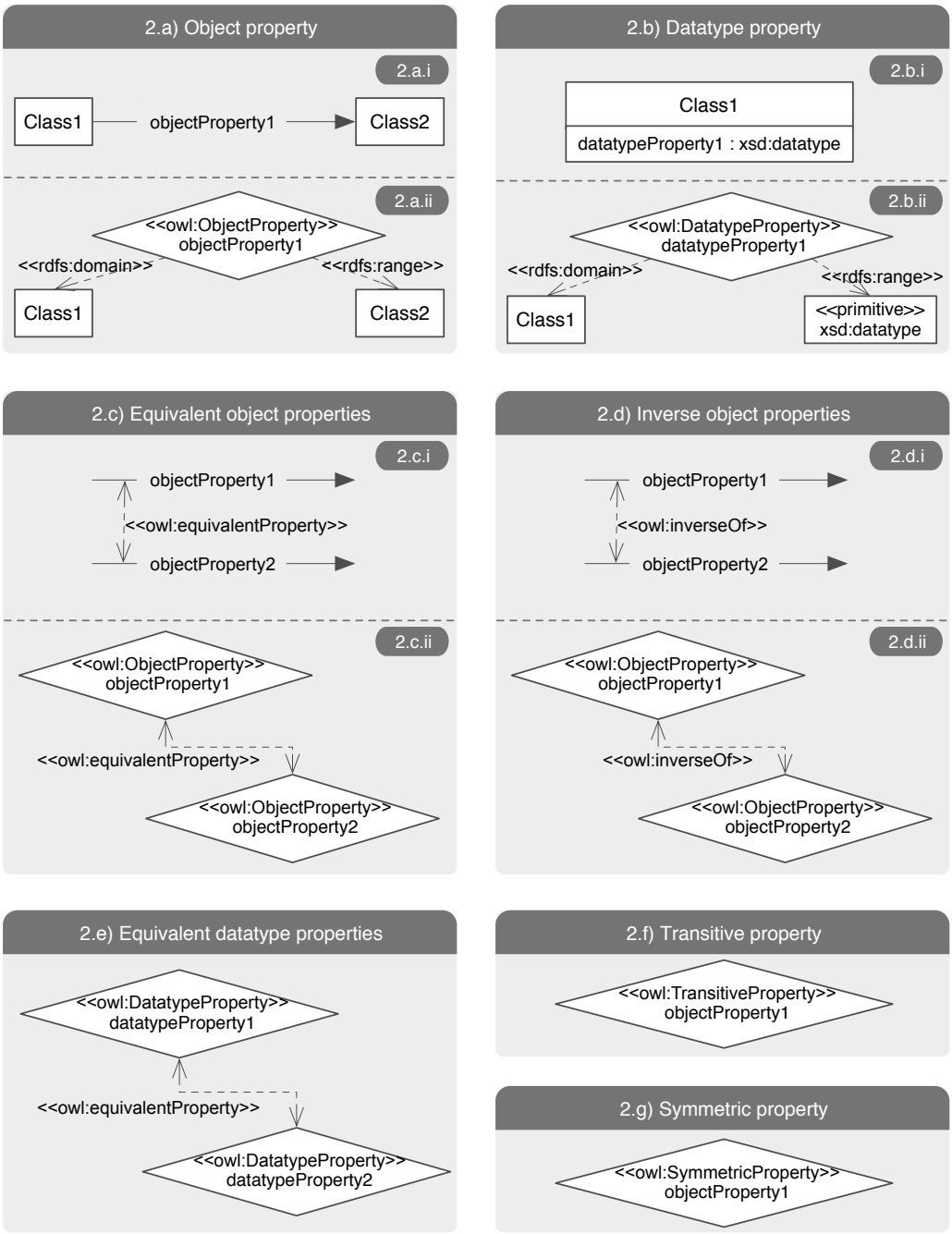


Figure 4.4: Notation for properties, relations between properties and property characteristics. Adapted from (Haase et al., 2009)

2) **Properties**: the graphical representation for properties, relations between properties and property characteristics are depicted in Figure 4.4. The constructs included in such figure are:

2.a) **Object properties** (relationships): object properties are represented by:

2.a.i) Labelled arrows: in this case the domain and range of the property are indicated by the origin and target of the arrow respectively. The name of the object property is represented by a label close to it.

2.a.ii) Labelled diamonds: in this case the domain and range of the property are indicated by dotted arrows labelled with the «`rdfs:domain`» and «`rdfs:range`» stereotypes respectively. The name of the object property is represented by a label within the diamond.

2.b) **Datatype properties** (attributes): datatype properties are represented by:

2.b.i) Labelled boxes: datatype properties can be represented as labelled boxed attached to boxes representing classes. The range might be included following the character ":" after the datatype label.[36]

2.b.ii) Labelled diamonds: in this case the domain and range of the property are indicated by dotted arrows labelled with the «`rdfs:domain`» and «`rdfs:range`» stereotypes respectively. The name of the datatype property is represented by a label within the diamond.

2.c) **Equivalent object properties** could be represented by means of:

2.c.i) Double-sided UML dependency with the «`owl:equivalentProperty`» stereotype linking the arrows that represent the involved object properties.

2.c.ii) Double-sided UML dependency with the «`owl:equivalentProperty`» stereotype linking the diamonds that represent the involved object properties.

2.d) **Inverse object properties** could be represented by means of:

2.d.i) Double-sided UML dependency with the «`owl:inverseOf`» stereotype linking the arrows that represent the involved object properties.

[36]The notation for attributes is explained in this section for the sake of readability as it would be needed for Chapter 5, even though no attribute or datatype appears in this chapter's examples.

2.d.ii) Double-sided UML dependency with the «owl:inverseOf» stereotype linking the diamonds that represent the involved object properties.

2.e) **Equivalent datatype properties** are represented by a double-sided dependency with the «owl:equivalentProperty» stereotype linking the diamonds that represent the datatype properties.[37]

2.f) **Transitive property** are represented by a labelled diamond, which represents the property itself, including the «owl:TransitiveProperty» stereotype.

2.g) **Symmetric properties** are represented by a labelled diamond, which represents the property itself, including the «owl:SymmetricProperty» stereotype.

3) **Individuals**: the graphical representation for individuals and class assertions are depicted in See Figure 4.5.

3.a) **Individuals** are represented by labelled boxes with underlined names.

3.b) **Class membership**:

3.b.i) Labelled box with the individual name followed by the character ":" and the class name, all underlined.

3.b.ii) UML dependency arrow with the stereotype «rdf:type».

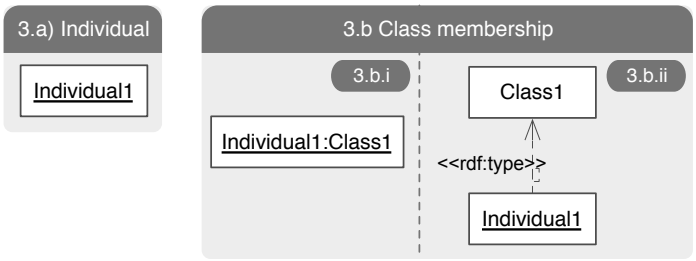


Figure 4.5: Notation for individuals and class membership. Adapted from (Haase et al., 2009)

Figure 4.6 collects the list of pitfalls together with the corresponding tables where each pitfall is described. Some pitfalls' titles have been slightly modified when necessary to provide a more accurate identification of the problem described. However the pitfall codes remain stable.

[37]The equivalent properties notation when the datatype properties are represented by boxes attached to classes is not considered for the sake of clarity.

P01. Creating polysemous elements (Table 4.1)
P02. Creating synonyms as classes (Table 4.2)
P03. Creating the relationship "is" instead of using "rdfs:subClassOf", "rdf:type" or "owl:sameAs" (Table 4.3)
P04. Creating unconnected ontology elements (Table 4.4)
P05. Defining wrong inverse relationships (Table 4.5)
P06. Including cycles in a class hierarchy (Table 4.6)
P07. Merging different concepts in the same class (Table 4.7)
P08. Missing annotations (Table 4.8)
P09. Missing domain information (Table 4.9)
P10. Missing disjointness (Table 4.10)
P11. Missing domain or range in properties (Table 4.11)
P12. Equivalent properties not explicitly declared (Table 4.12)
P13. Inverse relationships not explicitly declared (Table 4.13)
P14. Misusing "owl:allValuesFrom" (Table 4.14)
P15. Using "some not" in place of "not some" (Table 4.15)
P16. Using a primitive class in place of a defined one (Table 4.16)
P17. Overspecializing a hierarchy (Table 4.17)
P18. Overspecializing the domain or range (Table 4.18)
P19. Defining multiple domains or ranges in properties (Table 4.19)
P20. Misusing ontology annotations (Table 4.20)
P21. Using a miscellaneous class (Table 4.21)
P22. Using different naming conventions in the ontology (Table 4.22)
P23. Duplicating a datatype already provided by the implementation language (Table 4.23)
P24. Using recursive definitions (Table 4.24)
P25. Defining a relationship as inverse to itself (Table 4.25)
P26. Defining inverse relationships for a symmetric one (Table 4.26)
P27. Defining wrong equivalent properties (Table 4.27)
P28. Defining wrong symmetric relationships (Table 4.28)
P29. Defining wrong transitive relationships (Table 4.29)
P30. Equivalent classes not explicitly declared (Table 4.30)
P31. Defining wrong equivalent classes (Table 4.31)
P32. Several classes with the same label (Table 4.32)
P33. Creating a property chain with just one property (Table 4.33)
P34. Untyped class (Table 4.34)
P35. Untyped property (Table 4.35)
P36. URI contains file extension (Table 4.36)
P37. Ontology not available on the Web (Table 4.37)
P38. No OWL ontology declaration (Table 4.38)
P39. Ambiguous namespace (Table 4.39)
P40. Namespace hijacking (Table 4.40)
P41. No license declared (Table 4.41)

Figure 4.6: List of pitfalls and corresponding tables in this thesis.

<table>
<tr><td>Title</td><td>P01. Creating polysemous elements</td><td>Importance level</td><td>Critical</td></tr>
<tr><td>Aspects</td><td>Modelling decisions
Ontology understanding
Wrong inference</td><td>Affects to</td><td>Classes
Object properties
Datatype properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">An ontology element (class, object property or datatype property) whose identifier has different senses is included in the ontology to represent more than one conceptual idea or property.</td></tr>
<tr><td colspan="4">Examples</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2"> 
</td><td colspan="2">In the graphical example, the class representing the concept “Theatre”, which has different meanings, is defined as subclass of both “Building” and “PerformingArt”. Following this conceptualization, an instance of “Theatre” in the sense of “PerformingArt” will be also classified as “Building”, and the other way around.</td></tr>
<tr><td colspan="4">How to solve it</td></tr>
<tr><td colspan="4">For classes:
(a) Create a different class for each meaning of the polysemous concept.
(b) If required, extend the hierarchies to which the new classes belong.
(c) If there were properties attached to the polysemous concept, analyse whether they should be reallocated among the new classes.
(d) Check whether there exists an ad-hoc relation between the new concepts and add such a relationship if needed.
For properties:
(a) Create a different object or datatype property for each meaning of the polysemous property.
(b) If needed, specify domains and ranges for the new properties.</td></tr>
</table>

Table 4.1: P01. Creating polysemous elements

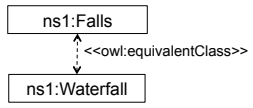
<table>
<tr><td>Title</td><td>P02. Creating synonyms as classes</td><td>Importance level</td><td>Minor</td></tr>
<tr><td>Aspects</td><td>Modelling decisions
Ontology understanding</td><td>Affects to</td><td>Classes</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">Several classes whose identifiers are synonyms are created and defined as equivalent (<code>owl:equivalentClass</code>) in the same namespace. This pitfall is related to the guidelines presented in [2], which explain that synonyms for the same concept do not represent different classes.</td></tr>
<tr><td colspan="4">Examples</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2"> 
</td><td colspan="2">The graphical example shows two classes, namely “Watefall” and “Falls”, identified by synonymous terms and denoting the same domain concept. These classes are defined in the same namespace as equivalent classes.</td></tr>
<tr><td colspan="4">How to solve it</td></tr>
<tr><td colspan="4">Create one class with different labels (via <code>rdfs:label</code> or other label annotation property), one for each synonymous term. See guides [1] and [2] for further details.</td></tr>
<tr><td>References</td><td colspan="3">[1] Aguado-De Cea, G., Montiel-Ponsoda, E., Poveda-Villalón, M., and Giraldo-Pasmin, O. X. (2015). Lexicalizing Ontologies: The issues behind the labels. In Multimodal communication in the 21st century: Professional and academic challenges. 33rd Conference of the Spanish Association of Applied Linguistics (AESLA), XXXIII AESLA.
[2] Noy, N. F., McGuinness, D. L., et al. (2001). Ontology development 101: A guide to creating your first ontology.</td></tr>
</table>

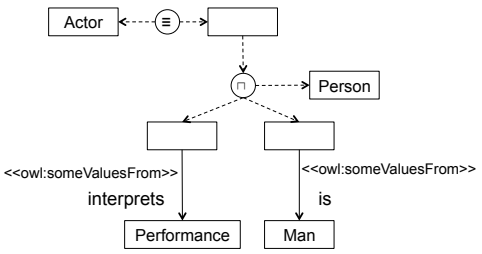
Table 4.2: P02. Creating synonyms as classes

Title	P03. Creating the relationship "is" instead of using "rdfs:subClassOf", "rdf:type" or "owl:sameAs"	**Importance level**	Critical
Aspects	Modelling decisions	**Affects to**	Object properties

Description

The relationship "is" is created in the ontology instead of using OWL primitives for representing the subclass relationship (`rdfs:subClassOf`), class membership (`rdf:type`), or the equality between instances (`owl:sameAs`). When concerning a class hierarchy, this pitfall is related to the guidelines for understanding the "is-a" relation provided in [1].

Examples

Graphical representation and/or OWL code	Natural language description
	In the graphical example, the class "Actor" is defined in the following way: $Actor \equiv Person \sqcap \exists interprets.Performance \sqcap \exists is.Man$. In this case, the relationship "is" is used within an existential axiom to indicate that, in order to belong to the class "Actor", an individual has to be of type "Man". The problem is that the fact of being a man is represented by the object property "is" instead of using the class membership as it is done for the class "Person".

How to solve it

Analyse if the relationship "is" is intended to be used as one of the following alternatives:

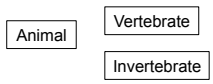
(a) To link two classes: if the relationship "is" holds between the classes A and B and every instance of A is also an instance of B, A should be defined as subclass of B by using the `rdfs:subClassOf` primitive instead of the property "is".

(b) To link an individual and a class: if the relationship "is" is intended to represent that an individual belongs to a given class (using another individual belonging to such class in the object of the "is" statement), the primitive that should be used is `rdf:type`.

(c) To link two different individuals: if the relationship "is" is created to state that two individuals are the same one, they should be linked using the primitive `owl:sameAs`.

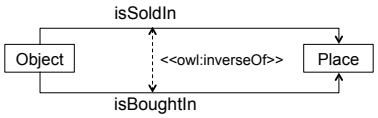
References	[1] Noy, N. F., McGuinness, D. L., et al. (2001). Ontology development 101: A guide to creating your first ontology.

Table 4.3: P03. Creating the relationship "is" instead of using "rdfs:subClassOf", "rdf:type" or "owl:sameAs"

<table>
<tr><td>Title</td><td>P04. Creating unconnected ontology elements</td><td>Importance level</td><td>Minor</td></tr>
<tr><td>Aspects</td><td>Requirement completeness
Real world modelling</td><td>Affects to</td><td>Classes
Object properties
Datatype properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">Ontology elements (classes, object properties and datatype properties) are created isolated, with no relation to the rest of the ontology.</td></tr>
<tr><td colspan="4">Examples</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2"> 

No explicit evidence of:
SubClassOf(:Vertebrate :Animal)
SubClassOf(:Invertebrate :Animal)</td><td colspan="2">The graphical example shows the classes “Animal”, “Vertebrate” and “Invertebrate” and the lack of <code>rdfs:subClassOf</code> statements among them.</td></tr>
<tr><td colspan="4">How to solve it</td></tr>
<tr><td colspan="4">For classes:
(a) For each unconnected class analyse whether it could be related to other classes by an object property.
(b) For each unconnected class analyse whether it could be related to a datatype by a datatype property.
(c) For each pair of classes analyse whether there is a hierarchical relationship between them, asking whether one of the classes is subclass of the other.
For properties:
(a) Analyse whether the properties’ domains and ranges could be specified (see also pitfall “P11. Missing-domain or range in properties”).
(b) For each pair of object properties analyse whether there is a hierarchical relationship between them, asking whether one of the object properties is a specialization of the other.
(c) For each pair of datatype properties analyse whether there is a hierarchical relationship between them, asking whether one of the datatype properties is a specialization of the other.</td></tr>
</table>

Table 4.4: P04. Creating unconnected ontology elements

<table>
<tr><td>Title</td><td>P05. Defining wrong inverse relationships</td><td>Importance level</td><td>Critical</td></tr>
<tr><td>Aspects</td><td>Wrong inference</td><td>Affects to</td><td>Object properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">Two relationships are defined as inverse relations when they are not necessarily inverse.</td></tr>
<tr><td colspan="4">Examples</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2"> 
</td><td colspan="2">In the graphical example we can see the result of considering two converse* relationships, “isSoldIn” and “isBoughtIn”, as inverse. This fact could cause an error at the semantic level, since if we have “object1 isSoldIn place3”, then the reasoner will infer that “place3 isBoughtIn object1” which is semantically wrong. In addition, as the domain of “isBoughtIn” is set as “Object”, the individual “place3” will be classified as “Object”, which is also wrong.

*Note: converse words are pairs of words that refer to a relationship from opposite points of view, such as buy/sell, parent/child, among others.</td></tr>
<tr><td colspan="4">How to solve it</td></tr>
<tr><td colspan="4">In order to solve this pitfall one should first check the following points:
(a) Check that whenever a relationship holds between individuals “a” and “b”, the inverse relationship necessarily holds between “b” and “a” and that the relations have the same semantics.
(b) Check whether the expressions used to define the domain of one relationship denote the same set of individuals denoted by the range of the other property, even though they might involve different class restrictions. Making this verification easier is one of the reasons why we recommend defining the domain and range of the relationships whenever possible.
Once the checks are done, the following actions might be taken:
Case 1: When (a) holds but (b) does not. If the relationships are actually inverse but the domain of one of them does not match the range of the other relationship, then the domains and ranges should be modified. One should make sure that the expression used in the domain of one relationship represents the same set of individuals represented by the range of the other property even though they are not using exactly the same class.
Case 2: When (a) does not hold. If it is not clear that the relationships are actually inverse, then the <code>owl:inverseOf</code> axiom should be removed. For addition information about creating inverse properties see “P13. Inverse relationships not explicitly declared”.</td></tr>
</table>

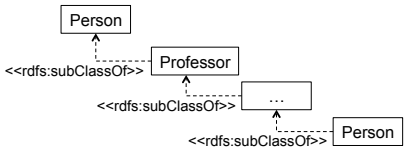
Table 4.5: P05. Defining wrong inverse relationships

Title	P06. Including cycles in a class hierarchy	**Importance level**	Critical
Aspects	Wrong inference	**Affects to**	Classes

Description

A cycle between two classes in a hierarchy is included in the ontology. A cycle appears when some class A has a subclass (directly or indirectly) B, and at the same time B is a superclass (directly or indirectly) of A. This pitfall was first identified in [1]. Guidelines presented in [2] also provide recommendations to avoid this pitfall.

Examples

Graphical representation and/or OWL code	Natural language description
	The graphical example shows a hierarchy where a class appears twice, namely "Person". More precisely, the class "Professor" is defined as subclass of "Person", and the class "Person" is defined indirectly as subclass of "Professor", thus creating a cycle. On the one hand, this situation conveys reasoning and logical consequences. When running a reasoner the duplicated class and all the intermediate classes in the hierarchy will be classified as equivalent classes. On the other hand, this case usually implies a semantic error as one of the `rdfs:subclassOf` relations involved in the cycle does not usually hold. In this case, it is the indirect subsumption between "Person" and "Professor" is not always true, as not every person is also a professor.

How to solve it

Case 1: When the cycle is not intended: This pitfall is usually solved by means of removing one of the subsumption relations. For doing this, one should identify which subsumption always holds. If the answer to "are all instances of A also instances of B?" is "no", then A is not subclass of B. If the answer to "are all instances of B also instances of A?" is "no", then B is not subclass of A. Once the incorrect `rdfs:subclassOf` is identified, it should be removed.

Case 2: When the cycle is intended and the two subsumptions involved are correct: This would mean that the classes are equivalents, what leads to two possible situations:

- Case 2.1: If both classes are defined in the same namespace, one should create just one class and attach different labels, one for the lexicalization of each class involved in the pitfall. This case represents a similar case to the one described in "P02. Creating synonyms as classes".
- Case 2.2: If both classes are defined in different namespaces, they should be declared as equivalent classes using the primitive `owl:equivalentClass`.

References

[1] Gómez-Pérez, A. (1999). Evaluation of Taxonomic Knowledge in Ontologies and Knowledge Bases. *Proceedings of the Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*. Alberta, Canada.

[2] Noy, N. F., McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology.

Table 4.6: P06. Including cycles in a class hierarchy

Title	P07. Merging different concepts in the same class	**Importance level**	Minor
Aspects	Modelling decisions Ontology understanding	**Affects to**	Classes
Description			
A class whose name refers to two or more different concepts is created.			
Examples			
Graphical representation and/or OWL code		Natural language description	
StyleAndPeriod		In the graphical example, a class called "StyleAndPeriod" shows how two different concepts, namely "Style" and "Period" are mixed up in one single class.	
How to solve it			
In order to solve this pitfall, one should create one class for each concept represented in the affected class. It is also advisable to check whether these new concepts might be related in any of the following ways: (a) Analyse whether the classes belong to the same hierarchy. In this case, a more general concept and the corresponding `rdfs:subClassOf` declaration might be created. (b) Analyse whether an ad-hoc relationship exists between the two classes. In this case, an object property might already exist in the ontology, otherwise it should be created. In this case, we recommend defining the domain and range of the object property (see "P11. Missing domain or range in properties" for additional information).			

Table 4.7: P07. Merging different concepts in the same class

<table>
<tr><td>Title</td><td>P08. Missing annotations</td><td>Importance level</td><td>Minor</td></tr>
<tr><td>Aspects</td><td>Ontology understanding
Ontology clarity</td><td>Affects to</td><td>Classes
Object properties
Datatype properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">This pitfall consists in creating an ontology element and failing to provide human readable annotations attached to it. Consequently, ontology elements lack annotation properties that label them (e.g. <code>rdfs:label</code>, <code>lemon:LexicalEntry</code>, <code>skos:prefLabel</code> or <code>skos:altLabel</code>) or that define them (e.g. <code>rdfs:comment</code> or <code>dc:description</code>). This pitfall is related to the guidelines provided in [3].</td></tr>
<tr><td colspan="4">Examples</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2">Person
No explicit evidence of:
AnnotationAssertion(rdfs:label :Person “name”)
AnnotationAssertion(rdfs:comment :Person “description”)</td><td colspan="2">The graphical example shows the class “Person” without any annotation that labels or describes it.</td></tr>
<tr><td colspan="4">How to solve it</td></tr>
<tr><td colspan="4">(a) Include label annotation properties (e.g. <code>rdfs:label</code>, <code>lemon:LexicalEntry</code>, <code>skos:prefLabel</code> or <code>skos:altLabel</code>) to provide terms that identify ontology elements.
(b) Include description annotation properties (e.g. <code>rdfs:comment</code> or <code>dc:description</code>) to provide natural language definitions of ontology elements.

See [1] and [2] for more detailed guidelines.</td></tr>
<tr><td>References</td><td colspan="3">[1] Aguado-De Cea, G., Montiel-Ponsoda, E., Poveda-Villalón, M., and Giraldo-Pasmin, O. X. (2015). Lexicalizing Ontologies: The issues behind the labels. In Multimodal communication in the 21st century: Professional and academic challenges. 33rd Conference of the Spanish Association of Applied Linguistics (AESLA), XXXIII AESLA.
[2] Montiel-Ponsoda, E., Vila Suero, D., Villazón-Terrazas, B., Dunsire, G., Escolano Rodríguez, E., Gómez-Pérez, A. (2011). Style guidelines for naming and labeling ontologies in the multilingual web.
[3] Vrandecic, D. (2010). Ontology Evaluation. PhD thesis.</td></tr>
</table>

Table 4.8: P08. Missing annotations

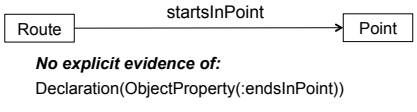
<table>
<tr><td>Title</td><td>P09. Missing domain information</td><td>Importance level</td><td>Minor</td></tr>
<tr><td>Aspects</td><td>No inference
Real world modelling
Requirement completeness</td><td>Affects to</td><td>Ontology</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">Part of the information needed for modelling the intended domain is not included in the ontology. This pitfall may be related to (a) the requirements included in the Ontology Requirement Specification Document (ORSD) that are not covered by the ontology, or (b) to the lack of knowledge that can be added to the ontology to make it more complete.</td></tr>
<tr><td colspan="4">Examples</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2"> 
</td><td colspan="2">The graphical example shows that the object property “startsInPoint” has been defined to indicate in which point a route starts. The lack of a property representing the end of the route is also indicated. However, that information should also be included.</td></tr>
<tr><td colspan="4">How to solve it</td></tr>
<tr><td colspan="4">It is advisable to check the ontology against the ORSD in order to discover missing classes, relationships and attributes and to include them in the ontology being built.
In addition, one could check for new information in the following ways, among others:
(a) Check whether any new class can be added and whether it belongs to a hierarchy.
(b) Check whether any new property might have a complementary action that indicates an inverse property. Then one should include the new property.</td></tr>
</table>

Table 4.9: P09. Missing domain information

<table>
<tr><td>Title</td><td>P10. Missing disjointness</td><td>Importance level</td><td>Important</td></tr>
<tr><td>Aspects</td><td>Real world modelling</td><td>Affects to</td><td>Classes
Object properties
Datatype properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">The ontology lacks disjoint axioms between classes or between properties that should be defined as disjoint. This pitfall is related with the guidelines provided in [1], [2] and [3].</td></tr>
<tr><td colspan="4">Examples</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2"> 
No explicit evidence of:
DisjointClasses(:Odd:Even)</td><td colspan="2">The graphical example shows that the classes “Odd” and “Even” are included in the ontology. However, the disjoint axiom <code>owl:disjointWith</code> among them is not declared. In this situation there could be individuals belonging to “Odd” and “Even” numbers at the same time which is conceptually wrong.</td></tr>
<tr><td colspan="4">How to solve it</td></tr>
<tr><td colspan="4">For classes:
(a) Look for pairs of concepts that can not share instances and define them as disjoint classes using the primitive <code>owl:disjointWith</code>. If a set of classes from the ontology are pairwise disjoint, then use <code>owl:AllDisjointClasses</code> from OWL 2.
For properties (only for OWL 2):
(a) Look for pairs of properties for which there are no two individuals that are interlinked by both properties and define them as disjoint properties using the primitive <code>owl:propertyDisjointWith</code>. If a set of properties from the ontology are pairwise disjoint, then use <code>owl:AllDisjointProperties</code>.</td></tr>
<tr><td>References</td><td colspan="3">[1] Gómez-Pérez, A. (2004). Ontology evaluation. In Handbook on ontologies, pages 251-273. Springer.
[2] Noy, N. F., McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology.
[3] Rector, A., Drummond, N., Horridge, M., Rogers, J., Knublauch, H., Stevens, R., Wang, H., and Wroe, C. (2004). OWL pizzas: Practical experience of teaching OWL-DL: Common errors & common patterns. In Engineering Knowledge in the Age of the Semantic Web, pages 63-81. Springer.</td></tr>
</table>

Table 4.10: P10. Missing disjointness

Title	P11. Missing domain or range in properties	**Importance level**	Important
Aspects	No inference Ontology understanding	**Affects to**	Object properties Datatype properties

Description

Object and/or datatype properties without domain or range (or none of them) are included in the ontology.

Examples

Graphical representation and/or OWL code	Natural language description
  ***No explicit evidence of:*** ObjectPropertyDomain(:writesLiteraryWork :Writer) ObjectPropertyRange(:writesLiteraryWork :LiteraryWork)	The graphical example shows that both the object property "writesLiteraryWork" and the classes "Writer" and "LiteraryWork" are included in the ontology. However, no connection between the classes and the object property is declared.

How to solve it

Domains and ranges should be declared in order to provide object and datatype properties with a more complete definition.

For domains:

(a) For each object or datatype property in the ontology without a domain defined, we recommend answering the following question "What is the most general class in the ontology whose instances could serve as subject of the property?". The class that represents the answer will be the domain of the property. If the answer is `owl:Thing`, we recommend looking for several classes whose instances could serve as subjects of the property. Then, join these classes by the "or" operator (i.e., `owl:unionOf`) and set this union of classes as the domain of the property.

For ranges:

(a) For each object property in the ontology without a range defined, we recommend answering the following question "What is the most general class in the ontology whose instances could serve as object of the property?". The class that represents the answer will be the range of the property. If the answer is `owl:Thing`, we recommend looking for several classes whose instances could serve as subjects of the property. Then, join these classes by the "or" operator (i.e., `owl:unionOf`) and set this union of classes as the range of the object property.

(b) For each datatype property in the ontology without a range defined, we recommend answering the following question "What would be the format of the data (strings of characters, positive numbers, dates, floats, etc.) used to fill in this information?". Choose the built-in datatype from the hierarchy in [1] that is closest to the answer to the previous question and set it as the range of the datatype property. If there is more than one plausible candidate, find their lowest common ancestor and set it as the range of the datatype property.

References	[1] Built-in datatype hierarchy http://www.w3.org/TR/xmlschema-2/built-in-datatypes

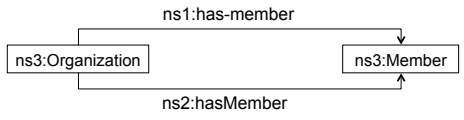
Table 4.11: P11. Missing domain or range in properties

Title	P12. Equivalent properties not explicitly declared	**Importance level**	Important
Aspects	No inference Ontology understanding	**Affects to**	Object properties Datatype properties

Description

The ontology lacks information about equivalent properties (`owl:equivalentProperty`) in the cases of duplicated relationships and/or attributes.

Examples

Graphical representation and/or OWL code	Natural language description
 ***No explicit evidence of:*** EquivalentObjectProperties(ns1:has-member ns2:hasMember)	The graphical example shows two properties defined in different namespaces (ns1 and ns2) that refer to the same relationship between concepts, more precisely, both indicate the members of an organization. However, such properties are not defined as equivalent of each other, even though one could say that both properties have the same meaning and every time one property holds, the other property holds as well.

How to solve it

In order to check whether two properties (object properties or datatype properties) defined in two different namespaces could be defined as equivalent properties one should:

(a) Check that both properties hold for the same set of individuals, verifying that their domains are the same class or denote the same set of individuals, even though they might involve different class restrictions.

(b) Check ranges:

  (b.1) For object properties: verify that the ranges of both object properties are the same class or denote the same set of individuals, even though they might involve different class restrictions.

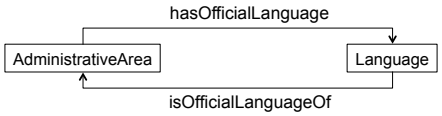
  (b.2) For datatypes properties: verify that the ranges of both datatype properties are the same datatype.

(c) Check that both properties represent the same real-world relation between classes or the same attributes in case of datatype properties.

If (a), (b) and (c) hold, the properties should be declared equivalent using `owl:equivalentProperty`.

Finally, check if both properties are really needed: if they are defined in the same namespace, they could be either redundant or they could have different intended meanings. In both cases, a disambiguation step would be needed. If they are redundant and they are defined in a namespace under your control, remove one of them. If they actually have the same meaning and are defined in namespaces you do not control, keep them, along with the equivalent property statement.

Table 4.12: P12. Equivalent properties not explicitly declared

<table>
<tr><td>Title</td><td>P13. Inverse relationships not explicitly declared</td><td>Importance level</td><td>Minor</td></tr>
<tr><td>Aspects</td><td>No inference
Ontology understanding</td><td>Affects to</td><td>Object properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">This pitfall appears when any relationship (except for those that are defined as symmetric properties using <code>owl:SymmetricProperty</code>) does not have an inverse relationship (<code>owl:inverseOf</code>) defined within the ontology.</td></tr>
<tr><td colspan="4">Examples</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2"> 

No explicit evidence of:
InverseObjectProperties(:hasOfficialLanguage :isOfficialLanguageOf)</td><td colspan="2">The graphical example shows two object properties, namely “hasOfficialLanguage” and “isOfficialLanguageOf”. The former describes that an administrative area has an official language, and the latter describes that a language is official language of an administrative area. In this case, both properties are defined in the ontology but no <code>owl:inverseOf</code> statement is declared between them.</td></tr>
<tr><td colspan="4">How to solve it</td></tr>
<tr><td colspan="4">In principle, all relationships, apart form the symmetric ones, could have an inverse relationship. An exception for this rule might be the properties created for an n-ary relationship, but not necessarily.

For each object property without an inverse property (<code>owl:inverseOf</code>) in the ontology, follow one of the alternatives:
(a) Check whether there is already an object property in the ontology that represents its inverse relationship. If so, verify that the domain of each relationship matches the range of the other. Make sure that the expressions used to define the domain of one relationship represent the same set of individuals denoted by the range of the other relationship, even though they might involve different class restrictions. If this is the case, define the two object properties as inverse using <code>owl:inverseOf</code>.
(b) Verify whether it is reasonable to create an inverse object property by altering the verb from active to passive voice or by using an adequate converse term*. In case this new object property is created, define it as inverse of the one being analysed using <code>owl:inverseOf</code>. Include, if possible, the domain and range of the new object property. Make sure that the expressions used to define the domain of each relationship denote the same set of individuals denoted by the range of the other property, even though they might involve different class restrictions.

*Converse terms are pairs of terms that refer to a relationship from opposite points of view, such as buy/sell, parent/child, among others. Note that a converse term does not always represent an inverse relationship and it might have to be adapted. For example, “buy” would not be the inverse of “sell” according to the semantics of <code>owl:inverseOf</code>. However, the relationship “hasParent” would be inverse of “hasChild”. See pitfall “P05. Defining wrong inverse relationships” for further details.</td></tr>
</table>

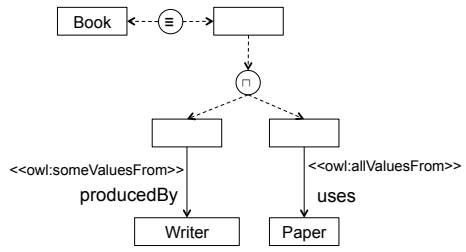
Table 4.13: P13. Inverse relationships not explicitly declared

Title	P14. Misusing “owl:allValuesFrom”	**Importance level**	Critical
Aspects	Modelling decisions	**Affects to**	Classes

Description

This pitfall consists in using the universal restriction (`owl:allValuesFrom`) as the default qualifier instead of the existential restriction (`owl:someValuesFrom`). Additional information about this pitfall is provided in [1].

Examples

Graphical representation and/or OWL code	Natural language description
	In the graphical example a definition of the class “Book” is provided by means of an `owl:equivalentClass` axiom in the following way: $Book \equiv \exists producedBy.Writer \sqcap \forall uses.Paper$. While the `owl:someValuesFrom` axiom is properly used for stating that the book has to be produced by at least one writer, it is not correct to say that all the materials used in the book have to belong to the class “Paper”, as for example, another material used during the production of a book might be “Ink”, among others.

How to solve it

An universal restriction (`owl:allValuesFrom`) can be used to restrict the range of a relationship, i.e., to state that, in the context of an axiom, only individuals belonging to the specified class can act as objects of that relationship. Considering a class “ClassA” used as target of an universal restriction for a given relationship, one should:

(a) Answer the question "Is it possible to have individuals that do not belong to “ClassA” acting as object of such property?" If the answer is “yes”, the universal restriction (`owl:allValuesFrom`) should be deleted. To analyse whether the universal restriction should be replaced by an existential one check the point (b).

(b) Check whether the intended meaning of the restriction is to state that at least one individual belonging to “ClassA” should appear as object in an instantiation of such property. In this case, an existential restriction (`owl:someValuesFrom`) should be used instead of the universal one (`owl:allValuesFrom`).

References	[1] Rector, A., Drummond, N., Horridge, M., Rogers, J., Knublauch, H., Stevens, R., Wang, H., and Wroe, C. (2004). OWL pizzas: Practical experience of teaching OWL-DL: Common errors & common patterns. In *Engineering Knowledge in the Age of the Semantic Web*, pages 63-81. Springer.

Table 4.14: P14. Misusing “owl:allValuesFrom”

Title	P15. Using "some not" in place of "not some"	**Importance level**	Critical
Aspects	Wrong inference	**Affects to**	Classes

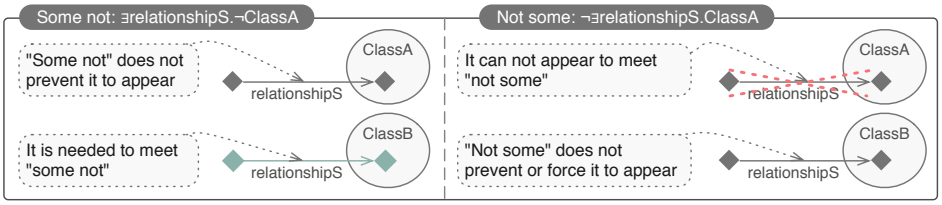
Description

The pitfall consists in using a "some not" structure when a "not some" is required. This is due to the misplacement of the existential quantifier (`owl:someValuesFrom`) and the negative operator (`owl:complementOf`).

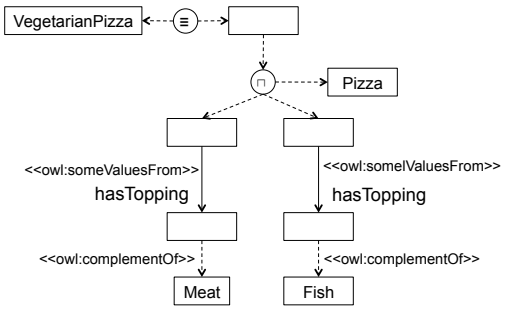
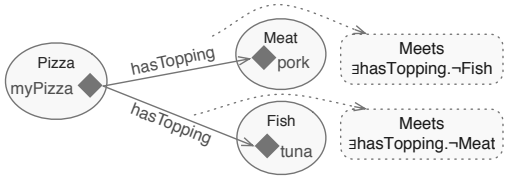
(a) When to use a "some not" structure ($\exists relationshipS.\neg ClassA$): to state that there is at least one individual acting as object of the relationship "relationshipS" and such individual do not belong to class "ClassA". This implies that there must be at least one instantiation of the relationshipS whose target does not belong to "ClassA". This does not prevent instances from ClassA acting as objects of the relationship.

(b) When to use a "not some" structure ($\neg\exists relationshipS.ClassA$): to state that no individuals in class "ClassA" act as objects of the relationship "relationshipS". This does not imply the existence of individuals that do not belong to ClassA acting as objects of the relationship.

This pitfall is explained in more detail in [1]. See figure below for more details about situations using "some not" or "not some".



Examples

Graphical representation and/or OWL code	Natural language description
	The graphical description shows an attempt to define "VegetarianPizza" by stating that it has some ingredients that are not "Meat" and some ingredients that are not "Fish". This representation follows a "some not" structure. The axiom provided for stating this is: $VegetarianPizza \equiv Pizza \sqcap \exists hasTopping.\neg Meat \sqcap \exists hasTopping.\neg Fish$. According to this modelling, any pizza would be classified as "VegetarianPizza" as long as it contains one topping that is not "Meat" and one topping that is not "Fish", which can be trivially satisfied for a pizza having both meat and fish as ingredients. See figure below for a graphical explanation of this situation. 

How to solve it

When a "some not" structure is used in place of a "not some" one (see the "Description" field for guidelines about when to use each one), one should place the owl:complementOf operator at the beginning of the axiom, that is replacing $\exists relationshipS.\neg ClassA$ by $\neg\exists relationshipS.ClassA$.

References

[1] Rector, A., Drummond, N., Horridge, M., Rogers, J., Knublauch, H., Stevens, R., Wang, H., and Wroe, C. (2004). OWL pizzas: Practical experience of teaching OWL-DL: Common errors & common patterns. In *Engineering Knowledge in the Age of the Semantic Web*, pages 63-81. Springer.

Table 4.15: P15. Using "some not" in place of "not some"

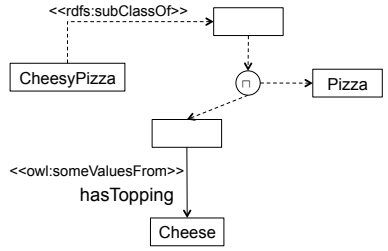
Title	P16. Using a primitive class in place of a defined one	**Importance level**	Critical
Aspects	No inference	**Affects to**	Classes

Description

"Primitive" classes are those for which there are only necessary conditions [1]. They are described using `rdfs:subClassOf`.
"Defined" classes are those for which there are necessary and sufficient conditions [1]. They are described using `owl:equivalentClass`.

This pitfall implies creating a primitive class rather than a defined one in case automatic classification of individuals is intended. It should be clarified that, in general, nothing will be inferred to be subsumed under a primitive class by the classifier [1]. This pitfall is related to the open world assumption.

Examples

Graphical representation and/or OWL code	Natural language description
	The graphical example shows the definition of a primitive class $CheesyPizza \sqsubset Pizza \sqcap \exists hasTopping.Cheese$, instead of creating it as a defined class in the following way: $CheesyPizza \equiv Pizza \sqcap \exists hasTopping.Cheese$. Therefore, the class "CheesyPizza" is defined in the example as the subset of individuals that belong to the class "Pizza" and have at least one ingredient from the class "Cheese". This model, being correct, will not classify under the class "CheesyPizza" individuals that have at least one topping from the class "Cheese". This is due to the creation of "CheesyPizza" as primitive (using `rdfs:subClassOf`) instead of defined (using `owl:equivalentClass`). If one would like to have such classification, one should use the `owl:equivalentClass`. This example is explained in more detail in [1].

How to solve it

To solve the problem we recommend answering for each primitive class "ClassA" the question "Do all the individuals that meet the necessary conditions included in the axioms of the primitive class "ClassA" actually belong to such class "ClassA"?":

(a) If the answer is "yes", one should create the class as defined by replacing the `rdfs:subClassOf` predicate by `owl:equivalentClass`.
(b) Otherwise, if meeting such constraint does not implies for an individual to belong to such class, one should keep the definition of the class as primitive by using `rdfs:subClassOf`.

References	[1] Rector, A., Drummond, N., Horridge, M., Rogers, J., Knublauch, H., Stevens, R., Wang, H., and Wroe, C. (2004). OWL pizzas: Practical experience of teaching OWL-DL: Common errors & common patterns. In *Engineering Knowledge in the Age of the Semantic Web*, pages 63-81. Springer.

Table 4.16: P16. Using a primitive class in place of a defined one

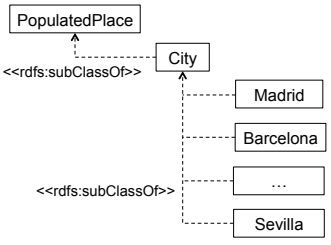
<table>
<tr><td>Title</td><td>P17. Overspecializing a hierarchy</td><td>Importance level</td><td>Important</td></tr>
<tr><td>Aspects</td><td>Modelling decisions</td><td>Affects to</td><td>Classes</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">The hierarchy in the ontology is specialized in such a way that the final leaves are defined as classes and these classes will not have instances.</td></tr>
<tr><td colspan="4">Examples</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2"> 
</td><td colspan="2">In the graphical example a three-level hierarchy is represented in which “City” is subclass of “PopulatedPlace” and particular cities in Spain are wrongly modelled as subclasses of “City”. According to this model, these classes could have instances, but, actually, they will not. This is due to the fact that there is no a set of individuals representing instances of the class “Madrid”. Instead, an individual “Madrid” could be an instance of “City”.</td></tr>
<tr><td colspan="4">How to solve it</td></tr>
<tr><td colspan="4">To discover if the leaves in a hierarchy are either classes or instances, one should answer the following question: “How many individuals will belong to this class in the knowledge base?” If the answer is 0 instances, then we recommend that such class is defined as an instance (using <code>rdf:type</code>), unless a canonical class is being modelled for some reason.
In addition, authors in [1] provide guidelines for distinguishing between a class and an instance when modelling hierarchies.</td></tr>
<tr><td>References</td><td colspan="3">[1] Noy, N. F., McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology.</td></tr>
</table>

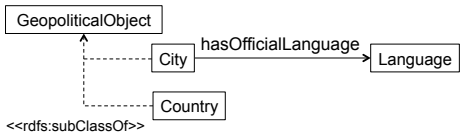
Table 4.17: P17. Overspecializing a hierarchy

Title	P18. Overspecializing the domain or range	**Importance level**	Important
Aspects	Wrong inference	**Affects to**	Object properties Datatype properties

Description

This pitfall consists in defining a domain or range not general enough for a property, i.e, no considering all the individuals or datatypes that might be involved in such a domain or range. This pitfall is related to the guidelines provided in [1] and [2].

Examples

Graphical representation and/or OWL code	Natural language description
	The graphical example shows a hierarchy containing "GeopoliticalObject" as root class, being "City" and "Country" its subclasses. The object property "hasOfficialLanguage" has been defined using the class "City" as domain and the class "Language" as range. As a country could also have at least one official language, it should be advisable to define the domain of the object property "hasOfficialLanguage" as "GeopoliticalObject" in order to allow both (cities and countries) to have such a property. The model in the graphical example could lead to unexpected inferences. For example, every individual acting as subject of "hasOfficialLanguage" would be classified as an instance of "City", even though they might not belong to such class.

How to solve it

When defining the domains and ranges of object and datatype properties one should check that all the individuals that might be involved in such a property (as subjects or objects) are taken into account.
For domains:

(a) For each object or datatype property with an overspecialised domain in the ontology, we recommend answering the following question "What is the most general class in the ontology whose instances could serve as subject of the property?". The class that represents the answer will be the domain of the property. If the answer is `owl:Thing`, we recommend looking for several classes whose instances could serve as subjects of the property. Then, join these classes by the "or" operator (i.e., `owl:unionOf`) and set this union of classes as the domain of the property.

For ranges:

(a) For each object property with an overspecialised range in the ontology, we recommend answering the following question "What is the most general class in the ontology whose instances could serve as object of the property?". The class that represents the answer will be the range of the property. If the answer is `owl:Thing`, we recommend looking for several classes whose instances could serve as objects of the property. Then, join these classes by the "or" operator (i.e., `owl:unionOf`) and set this union of classes as the range of the object property.

(b) For each datatype property in the ontology with a too narrowly defined range, we recommend answering the question "What would be the format of the data (decimal, integer, long, int, short, etc.) used to fill in this information?". Choose the built-in datatype from [3] that is closest to the answer to the previous question and set it as the range of the datatype property. If there is more than one plausible candidate, find their lowest common ancestor and set it as the range of the datatype property.

References

[1] Noy, N. F., McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology.
[2] Rector, A., Drummond, N., Horridge, M., Rogers, J., Knublauch, H., Stevens, R., Wang, H., and Wroe, C. (2004). OWL pizzas: Practical experience of teaching OWL-DL: Common errors & common patterns. In *Engineering Knowledge in the Age of the Semantic Web*, pages 63-81. Springer.
[3] Built-in datatype hierarchy http://www.w3.org/TR/xmlschema-2/built-in-datatypes

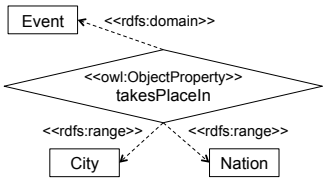
Table 4.18: P18. Overspecializing the domain or range

Title	P19. Defining multiple domains or ranges in properties	**Importance level**	Critical
Aspects	Wrong inference	**Affects to**	Object properties Datatype properties

Description

The domain or range (or both) of a property (relationships and attributes) is defined by stating more than one `rdfs:domain` or `rdfs:range` statements. In OWL multiple `rdfs:domain` or `rdfs:range` axioms are allowed, but they are interpreted as conjunction, being, therefore, equivalent to the construct `owl:intersectionOf`. This pitfall is related to the common error that appears when defining domains and ranges described in [2].

Examples

Graphical representation and/or OWL code	Natural language description
	The graphical example shows an object property, "takesPlacesIn", whose domain is the class "Event" and whose range is defined as "City" and as "Nation". As the range of the relationship is interpreted as an intersection, whenever an individual appears as object in such relationship, it will be classified both as instance of "City" and "Nation". This situation is probably not intended by the developer, since the intersection of "City" and "Nation" is the empty set.

How to solve it

If there is more than one `rdfs:domain` or `rdfs:range` statements defined for a given object or datatype property, check:

For domains:

(a) For each object or datatype property with multiple domains in the ontology, we recommend answering the following question "Does every individual in the domain of the given property necessarily belong to each and every one of the intersected classes?". If the answer is "no", developers should (a) join the classes that appear in the different domain declarations by the "or" operator (i.e., `owl:unionOf`) and set this union of classes as the domain of the property and (b) remove the initial `rdfs:domain` declarations.

For ranges:

(a) For each object property with multiple ranges in the ontology, we recommend answering the following question "Does every individual in the range of the given object property necessarily belong to each and every one of the intersected classes?". If the answer is "no", developers should (a) join the classes that appear in the different range declarations by the "or" operator (i.e., `owl:unionOf`) and set this union of classes as the range of the property and (b) remove the initial `rdfs:range` declarations.

(b) For each datatype property with multiple domains in the ontology, we recommend including at most one datatype as range. We recommend taking into account the built-in datatype hierarchy defined in [1] to select such datatype. If there is more than one plausible candidate, find their lowest common ancestor and set it as the range of the datatype property.

References	[1] Built-in datatype hierarchy http://www.w3.org/TR/xmlschema-2/built-in-datatypes [2] Rector, A., Drummond, N., Horridge, M., Rogers, J., Knublauch, H., Stevens, R., Wang, H., and Wroe, C. (2004). OWL pizzas: Practical experience of teaching OWL-DL: Common errors & common patterns. In *Engineering Knowledge in the Age of the Semantic Web*, pages 63-81. Springer.

Table 4.19: P19. Defining multiple domains or ranges in properties

<table>
<tr><td>Title</td><td>P20. Misusing ontology annotations</td><td>Importance level</td><td>Minor</td></tr>
<tr><td>Aspects</td><td>Ontology understanding</td><td>Affects to</td><td>Classes
Object properties
Datatype properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">The contents of some annotation properties are swapped or misused. This pitfall might affect annotation properties related to natural language information (for example, annotations for naming such as <code>rdfs:label</code> or for providing descriptions such as <code>rdfs:comment</code>).
Other types of annotation could also be affected as temporal, versioning information, among others.</td></tr>
<tr><td colspan="4">Examples</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2">:Crossroads rdf:type owl:Class ;
rdfs:label "the place of intersection of two or more roads"@en ;
rdfs:comment "Crossroads"@en .</td><td colspan="2">The owl code shown in the example contains swapped information for the <code>rdfs:label</code> and <code>rdfs:comment</code> annotation properties.</td></tr>
<tr><td colspan="4">How to solve it</td></tr>
<tr><td colspan="4">Check that the expected content is provided for each metadata annotation:
Check the content of names provided using annotations like <code>rdfs:label</code>, <code>skos:prefLabel</code> or <code>skos:altLabel</code>.
Check the content of definitions provided using annotations like <code>rdfs:comment</code> or <code>dcterms:description</code>.
Check the dates formats for annotations like <code>dcterms:created</code> or <code>dcterms:modified</code>.
Check the version formats for annotations like <code>owl:versionInfo</code>.</td></tr>
</table>

Table 4.20: P20. Misusing ontology annotations

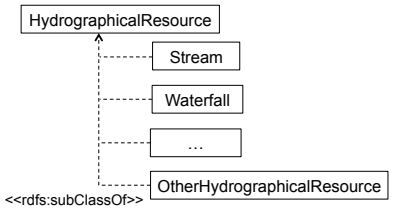
<table>
<tr><td>Title</td><td>P21. Using a miscellaneous class</td><td>Importance level</td><td>Minor</td></tr>
<tr><td>Aspects</td><td>Modelling decisions</td><td>Affects to</td><td>Classes</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">This pitfall refers to the creation of a class with the only goal of classifying the instances that do not belong to any of its sibling classes (classes with which the miscellaneous problematic class shares a common direct ancestor).</td></tr>
<tr><td colspan="4">Examples</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2"> 
</td><td colspan="2">The graphical example shows a hierarchy of hydrographical resources with the root class “HydrographicalResource”. This class has different subclasses, such as “Stream” and “Waterfall”, among others. The hierarchy includes a class named “OtherHydrographicalResource” which is intended to contain those individuals that do not fit in any of its sibling classes (“Stream”, “Waterfall”, etc.). This is a common modelling mistake, as not all individual need to be classified at the same level in the hierarchy. There might be individuals that do not belong to any of the defined subclasses, but to an upper class (in the example “HydrographicalResource”) in a particular level of the hierarchy.</td></tr>
<tr><td colspan="4">How to solve it</td></tr>
<tr><td colspan="4">To correct this pitfall, one should remove the class created as miscellanea. If there are instances classified under such class, they should be defined as instances of a direct superclass of the class to be removed.</td></tr>
</table>

Table 4.21: P21. Using a miscellaneous class

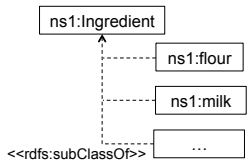
<table>
<tr><td>Title</td><td>P22. Using different naming conventions in the ontology</td><td>Importance level</td><td>Minor</td></tr>
<tr><td>Aspects</td><td>Ontology clarity</td><td>Affects to</td><td>Ontology</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">The ontology elements are not named following the same convention (for example CamelCase or use of delimiters as “-” or “_”) . Some notions about naming conventions are provided in [3].</td></tr>
<tr><td colspan="4">Examples</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2"> 
</td><td colspan="2">The graphical example shows classes in a same namespace that were created without following a common naming convention: some of them start with lowercase and others with uppercase.</td></tr>
<tr><td colspan="4">How to solve it</td></tr>
<tr><td colspan="4">To solve this pitfall one should choose a naming convention (for example CamelCase) and rename all the terms that do not follow the selected convention.
For further style guidelines for natural language annotations see [1] and [2].
The following aspects should be also taken into account:
(a) One can only edit or modify those terms defined in a namespace under one’s control. If a chosen naming convention is different from the convention used in ontologies being reused, or if two reused ontologies follow different naming conventions, then the elements in the reused ontologies cannot be changed. In this case, the pitfall might remain.
(b) If the ontology has been already published or if it is being used by third parties, one should consider not modifying the ontology elements URIs for the sake of backward compatibility.</td></tr>
<tr><td>References</td><td colspan="3">[1] Aguado-De Cea, G., Montiel-Ponsoda, E., Poveda-Villalón, M., and Giraldo-Pasmin, O. X. (2015). Lexicalizing Ontologies: The issues behind the labels. In Multimodal communication in the 21st century: Professional and academic challenges. 33rd Conference of the Spanish Association of Applied Linguistics (AESLA), XXXIII AESLA.
[2] Montiel-Ponsoda, E., Vila Suero, D., Villazón-Terrazas, B., Dunsire, G., Escolano Rodríguez, E., Gómez-Pérez, A. (2011). Style guidelines for naming and labeling ontologies in the multilingual web.
[3] Noy, N. F., McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology.</td></tr>
</table>

Table 4.22: P22. Using different naming conventions in the ontology

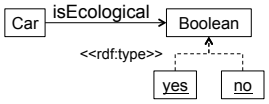
<table>
<tr><td>Title</td><td>P23. Duplicating a datatype already provided by the implementation language</td><td>Importance level</td><td>Important</td></tr>
<tr><td>Aspects</td><td>Modelling decisions</td><td>Affects to</td><td>Classes</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">A class and its corresponding individuals are created to represent existing datatypes in the implementation language.</td></tr>
<tr><td colspan="4">Examples</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2"> 
</td><td colspan="2">The graphical example shows the class “Car” linked to a class “Boolean”, which aims to model whether a car is ecological or not by means of the object property “isEcological”. The developer is thus creating a new class and two instances that duplicate a datatype already provided by the implementation language (<code>xsd:Boolean</code>).</td></tr>
<tr><td colspan="4">How to solve it</td></tr>
<tr><td colspan="4">When a class is created to represent a datatype that already exists in the implementation language, we recommend:
(a) Replacing the class created by the datatype provided by the implementation language (see hierarchy in [1]). If there is more than one plausible candidate, choose their lowest common ancestor.
(b) Replacing the object property, if any (this applies in case the class replaced by the datatype was set as range of an object property), by a datatype property.</td></tr>
<tr><td>References</td><td colspan="3">[1] Built-in datatype hierarchy http://www.w3.org/TR/xmlschema-2/built-in-datatypes</td></tr>
</table>

Table 4.23: P23. Duplicating a datatype already provided by the implementation language

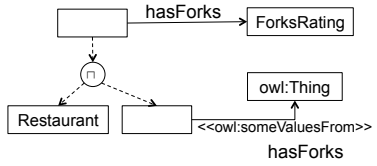
Title	P24. Using recursive definitions	**Importance level**	Important
Aspects	Modelling decisions	**Affects to**	Classes Object properties Datatype properties

Description

An ontology element (a class, an object property or a datatype property) is used in its own definition. Some examples of this would be:

(a) the definition of a class as the enumeration of several classes including itself
(b) the appearance of a class within its `owl:equivalentClass` or `rdfs:subClassOf` axioms
(c) the appearance of an object property in its `rdfs:domain` or range `rdfs:range` definitions
(d) the appearance of a datatype property in its `rdfs:domain` definition

Examples

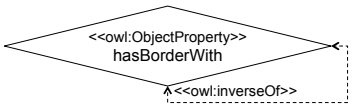
Graphical representation and/or OWL code	Natural language description
	The graphical example shows the definition of the relationship "hasForks", whose domain is declared to be the following unnamed class: $Restaurant \sqcap \exists hasForks.Thing$. In this case, the existential restriction over "hasForks" generates a superfluous recursion over the relationship's domain.

How to solve it

First, check whether the recursive definition is wrong or whether it is needed to define the term. Then follow one of the alternatives below:

(a) If the axiom containing the recursive definition is needed but the recursion itself is not precise, such axiom might need to be redefined in a more general way. In this case, one possibility is to keep the axiom replacing the recursive term by a more general one, using a superclass or super property of the element being defined.
(b) If the recursive definition is not intended it should be removed. In this case, if the recursion is part of a more complex axiom that include other restrictions, the rest of the axiom might not be removed.

Table 4.24: P24. Using recursive definitions

<table>
<tr><td>Title</td><td>P25. Defining a relationship as inverse to itself</td><td>Importance level</td><td>Important</td></tr>
<tr><td>Aspects</td><td>Modelling decisions</td><td>Affects to</td><td>Object properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">A relationship is defined as inverse of itself. In this case, this relationship could have been defined as <code>owl:SymmetricProperty</code> instead.</td></tr>
<tr><td colspan="4">Examples</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2"> 
</td><td colspan="2">The graphical example shows the object property “hasBorderWith”, which has been defined as inverse of itself by means of the primitive <code>owl:inverseOf</code>.</td></tr>
<tr><td colspan="4">How to solve it</td></tr>
<tr><td colspan="4">One should check whether the object property defined as inverse of itself satisfies the following condition: every time the object property holds between the individuals “a” and “b” it also holds between “b” and “a”. If the object property does fulfil the condition, it should be defined as symmetric (using <code>owl:SymmetricProperty</code>) and the <code>owl:inverseOf</code> statement should be deleted.

While the situation described in this pitfall will not lead to any reasoning problem and it could be considered a workaround to avoid increasing the ontology expressivity, it is more accurate to define the object property as symmetric (using <code>owl:SymmetricProperty</code>) instead of as inverse to itself.</td></tr>
</table>

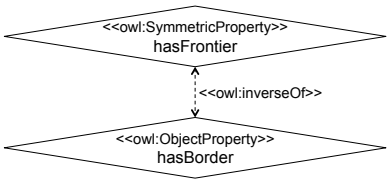
Table 4.25: P25. Defining a relationship as inverse to itself

Title	P26. Defining inverse relationships for a symmetric one	**Importance level**	Important
Aspects	Modelling decisions	**Affects to**	Object properties

Description

A symmetric object property (`owl:SymmetricProperty`) is defined as inverse of another object property (using `owl:inverseOf`).

Examples

Graphical representation and/or OWL code	Natural language description
	The graphical example shows the object property "hasFrontier", which is defined both as symmetric, using `owl:SymmetricProperty`, and as inverse of the object property "hasBorder", using `owl:inverseOf`.

How to solve it

Check whether the object property defined as symmetric satisfies the following condition: every time the property holds between the individuals "a" and "b", it also holds between "b" and "a". Then, follow the corresponding alternative:

(a) If the object property does not fulfil the condition, it might not be symmetric, in which case the `owl:SymmetricProperty` statement should be removed. Then, check whether the object properties are inverse of each other (see the description of pitfall "P05. Defining wrong inverse relationships" in Table 4.5 for further details).

(b) If the object property does fulfil the condition, check whether the object property defined as inverse should be defined as equivalent. If so, the `owl:inverseOf` statement may be replaced by an `owl:equivalentProperty` statement. For symmetric object properties defined as inverse of themselves see pitfall "P25. Defining a relationship as inverse to itself" in Table 4.25.

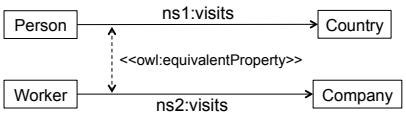
Table 4.26: P26. Defining inverse relationships for a symmetric one

Title	P27. Defining wrong equivalent properties	**Importance level**	Critical
Aspects	Wrong inference	**Affects to**	Object properties Datatype properties

Description

Two object properties or two datatype properties are defined as equivalent, using `owl:equivalentProperty`, even though they do not have the same semantics.

Examples

Graphical representation and/or OWL code	Natural language description
	The graphical example shows that the object property "visits" is defined in one namespace to capture that persons (represented by the class "Person") can visit countries ("Country"). This property is also defined in a different namespace to indicate that persons can visit companies (class "Company"). Both properties are defined as equivalent. Since both properties are defined as equivalent and their meanings differ, the model could lead to unexpected inferences, for example classifying instances of the class "Country" under the class "Company".

How to solve it

In order to have proper equivalent properties (object properties or datatype properties) that will not generate wrong inferences, check the following:

(a) Check that both properties hold for the same set of individuals, verifying that their domains are the same class or denote the same set of individuals, even though they might involve different class restrictions.

(b) Check ranges:

  (b.1) For object properties: verify that the ranges of both object properties are the same class or denote the same set of individuals, even though they might involve different class restrictions.

  (b.2) For datatypes properties: verify that the ranges of both datatype properties are the same datatype.

(c) Check that both properties represent the same real-world relation between classes or the same attributes in case of datatype properties.

If any of the above-mentioned options does not hold, the `owl:equivalentProperty` might be removed.

If all the above-mentioned options hold, check if both properties are really needed: if they are defined in the same namespace, they could be either redundant or they could have different intended meanings. In both cases, a disambiguation step would be needed. If they are redundant and they are defined in a namespace under your control, remove one of them. If they actually have the same meaning and are defined in namespaces you do not control, keep them, along with the equivalent property statement.

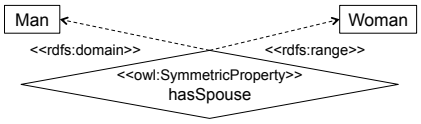
Table 4.27: P27. Defining wrong equivalent properties

Title	P28. Defining wrong symmetric relationships	**Importance level**	Critical
Aspects	Wrong inference	**Affects to**	Object properties

Description

A relationship is defined as symmetric, using `owl:SymmetricProperty`, when the relationship is not necessarily symmetric.

Examples

Graphical representation and/or OWL code	Natural language description
	The graphical example shows the object property "hasSpouse" defined between "Man" and "Woman". The object property is declared to be symmetric using `owl:SymmetricProperty`. This model indicates that every time the property holds between a man "m" and a woman "w", it also holds between the woman "w" and the man "m". This reasoning will not be erroneous in terms of property semantics, but it will lead to inference problems due to the domain and range declarations. Whenever an individual appears as subject of the property it is going to be classified as "Man", and if it appears as object it would be classified as "Woman". A reasoner would infer that the man "m" is also an instance of "Woman" and that the woman "w" is also an instance of "Man", which is clearly wrong. This situation could also lead to an inconsistency if the classes "Man" and "Woman" are defined as disjoint.

How to solve it

In order to have right symmetric properties that will not generate wrong deductions, one should answer the following question: "Every time that the object property is declared between the individuals "a" and "b", does the same object property hold between "b" and "a"?".

(a) If the answer is "no", the property might not be symmetric and the `owl:SymmetricProperty` should be removed.

(b) If the answer is "yes", one should check that the domain and the range of the object property denote the same set of individuals, even though they might involve different class restrictions. If the domain and range do not match, analyse whether they could be modified to fulfil such condition.

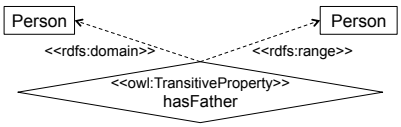
Table 4.28: P28. Defining wrong symmetric relationships

Title	P29. Defining wrong transitive relationships	**Importance level**	Critical
Aspects	Wrong inference	**Affects to**	Object properties

Description

A relationship is defined as transitive, using `owl:TransitiveProperty`, when the relationship is not necessarily transitive.

Examples

Graphical representation and/or OWL code	Natural language description
	The graphical example shows a transitive object property "hasFather" whose domain and range is the class "Person". If the relation holds between the individuals "a" and "b" and between "b" and "c", when applying inference it will be deduced that the relation also holds between "a" and "c", which is wrong.

How to solve it

In order to have right transitive object properties that will not have unexpected deductions, one should answer the following question: "Every time that the object property is declared between the individuals "a" and "b" and between "b" and "c", does the object property also hold between "a" and "c"?".

(a) If the answer is "no", the property might not be transitive and the `owl:TransitiveProperty` should be removed.

(b) If the answer is "yes", one should check that the domain and the range of the object property denote the same set of individuals, even though they might involve different class restrictions. If the domain and range do not match, analyse whether they could be modified to fulfil such condition.

Table 4.29: P29. Defining wrong transitive relationships

Title	P30. Equivalent classes not explicitly declared	**Importance level**	Important
Aspects	No inference	**Affects to**	Classes

Description

This pitfall consists in missing the definition of equivalent classes (`owl:equivalentClass`) in case of duplicated concepts. When an ontology reuses terms from other ontologies, classes that have the same meaning should be defined as equivalent in order to benefit the interoperability between both ontologies.

Examples

Graphical representation and/or OWL code	Natural language description
ns1:Car ns2:Motorcar ***No explicit evidence of:*** EquivalentClasses(:ns1:Car :ns2:Motorcar)	The graphical example shows two classes, namely "Car" and "Motorcar", defined in different namespaces and that refer to the same real world concept. However, no `owl:equivalentClass` axiom between them has been stated.

How to solve it

If the classes involved in the pitfall are defined in different namespaces and both are required, define them as equivalent by means of the `owl:equivalentClass` primitive. Consider also using other primitives such as `rdfs:subclassOf` in case they are not completely equivalent but one concept is more specific than the other.

If both classes are defined in the same namespace under the developer's control, we recommend removing one of them and attach its label (via `rdfs:label` or other label annotation property) to the remaining class. See "P02. Creating synonyms as classes" for further details.

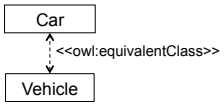
Table 4.30: P30. Equivalent classes not explicitly declared

Title	P31. Defining wrong equivalent classes	**Importance level**	Critical
Aspects	Wrong inference	**Affects to**	Classes

Description

Two classes are defined as equivalent, using `owl:equivalentClass`, when they are not necessarily equivalent.

Examples

Graphical representation and/or OWL code	Natural language description
	The graphical example shows the class "Car" being defined as equivalent to the class "Vehicle". However, "Car" is a hyponym of "Vehicle".

How to solve it

In order to check whether two equivalent classes are really equivalent one should answer the following question "Do each and every individual of one the classes also belong to the other class and vice-versa?".

(a) If the answer is "yes", and both classes represent the same real-word concept, the `owl:equivalentClass` should be kept.

(b) If the answer is "no", it means that the classes are not really equivalent and the `owl:equivalentClass` should be removed. In this case, another type of relationship might hold between the classes:

  (b.1) A hierarchical relation among the classes could exist. For example, one class might be hypernym or hyponym of the other. In this case, you should create a hierarchy between them using `rdfs:subClassOf`.

  (b.2) A mereological relation among the classes could exist. In this case, analyse the use of an object property "part of" in order to link both concepts.

Note: It is advisable to use Wordnet (http://wordnet.princeton.edu/) to check and discover these types of relations (hyponymy, hypernymy and mereology) between classes.

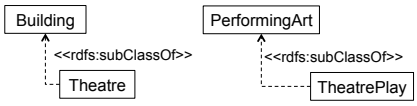
Table 4.31: P31. Defining wrong equivalent classes

Title	P32. Several classes with the same label	**Importance level**	Minor
Aspects	Ontology understanding	**Affects to**	Classes

Description

Two or more classes have the same content for natural language annotations for naming, for example the `rdfs:label` annotation. This pitfall might involve lack of accuracy when defining terms.

Examples

Graphical representation and/or OWL code	Natural language description
 AnnotationAssertion(rdfs:label :Theatre "Theatre") AnnotationAssertion(rdfs:label :TheratrePlay "Theatre") ***No explicit evidence of:*** EquivalentClasses(:Theatre :TheatrePlay)	The graphical example shows two classes "Theatre" and "TheatrePlay" with the same value for the annotation property `rdfs:label`. The first one, "Theatre", represents the building where the play takes place. The other class, "TheatrePlay", represents the play itself as performing art. While they represent different classes with different identifiers, they both have the same label.

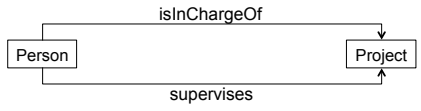
How to solve it

When two or more classes have the same content in an `rdfs:label` annotation and they are not equivalent classes (`owl:equivalentClass`), one should modify all the repeated labels making them more specific according to the meaning of the class they are attached to. It is possible to keep the original `rdfs:label` annotation for one of the classes if the rest of them are modified.
In addition, such labels should be self-contained, meaningful, and concise while as short as possible. For additional guidelines see [1].

References	[1] Aguado-De Cea, G., Montiel-Ponsoda, E., Poveda-Villalón, M., and Giraldo-Pasmin, O. X. (2015). Lexicalizing Ontologies: The issues behind the labels. In Multimodal communication in the 21st century: Professional and academic challenges. 33rd Conference of the Spanish Association of Applied Linguistics (AESLA), XXXIII AESLA.

Table 4.32: P32. Several classes with the same label

<table>
<tr><td>Title</td><td>P33. Creating a property chain with just one property</td><td>Importance level</td><td>Minor</td></tr>
<tr><td>Aspects</td><td>Modelling decisions</td><td>Affects to</td><td>Object properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">The OWL 2 construct <code>owl:propertyChainAxiom</code> allows a property to be defined as the composition of several properties (see http://www.w3.org/TR/owl2-new-features/F8:_Property_Chain_Inclusion for additional details). In this sense, when an individual “a” is connected with an individual “b” by a chain of two or more object properties (specified in the antecedent of the chain), it is necessary to connect “a” with “b” by using the object property in the consequent of the chain.

This pitfall consists in creating a property chain (<code>owl:propertyChainAxiom</code>) that includes only one property in the antecedent part.</td></tr>
<tr><td colspan="4">Examples</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2"> 

SubObjectPropertyOf(
ObjectPropertyChain(:isInChargeOf) :supervises)</td><td colspan="2">The graphical example shows the properties “isInChargeOf” and “supervises” both having domain “Person” and range “Project”. The property chain appears because if a person is in charge of a project, this means that the person supervises it ($isInChargeOf \rightarrow supervises$). Such property chain contains only one property in the antecedent, namely the property “isInChargeOf”, being the property in the consequent “supervises”.</td></tr>
<tr><td colspan="4">How to solve it</td></tr>
<tr><td colspan="4">In order to solve this pitfall one should include at least one more object property in the antecedent of the chain. Make sure the new properties are included in the correct place within the antecedent.

If no other object properties can be added in the antecedent of the chain, this means that the property chain does not exist. In this case, the property chain was probably included to state that every time that the property in the antecedent holds between two individuals, it also holds the property in the consequent. In that case, the property chain should be replaced by a <code>rdfs:subPropertyOf</code> statement between such properties.</td></tr>
</table>

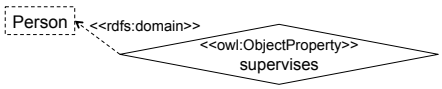
Table 4.33: P33. Creating a property chain with just one property

Title	P34. Untyped class	**Importance level**	Important
Aspects	Ontology language	**Affects to**	Classes

Description

An ontology element is used as a class without having been explicitly declared as such using the primitives `owl:Class` or `rdfs:Class`. This pitfall is related with the common problems listed in [1].

Examples

Graphical representation and/or OWL code	Natural language description
 ***No explicit evidence of:*** Declaration(Class(:Person))	The graphical example shows that the object property "supervises" has as domain the ontology element "Person". However, the ontologyy does not include the definition of "Person" as a class. Note that a dashed box is used to represent the ontology element that is not defined as class in the graphical example. This alternative notation is used for the sake of clarity. A regular box is not used as it would be confused with elements actually defined as classes in the ontology.

How to solve it

Add an statement in order to define the ontology element as a class by means of the primitives `owl:Class` or `rdfs:Class`.

References	[1] Hogan, A., Harth, A., Passant, A., Decker, S., and Polleres, A. (2010). Weaving the pedantic web. In Proceedings of the WWW2010 Workshop on Linked Data on the Web, LDOW 2010, Raleigh, USA, April 27, 2010.

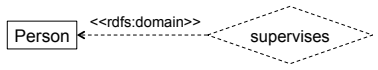
Table 4.34: P34. Untyped class

Title	P35. Untyped property	**Importance level**	Important
Aspects	Ontology language	**Affects to**	Object properties Datatype properties

Description

An ontology element is used as a property without having been explicitly declared as such using the primitives `rdf:Property`, `owl:ObjectProperty` or `owl:DatatypeProperty`. This pitfall is related with the common problems listed in [1].

Examples

Graphical representation and/or OWL code	Natural language description
 ***No explicit evidence of:*** Declaration(ObjectProperty(supervises)) Declaration(DataProperty(supervises))	The graphical example shows that the ontology element "supervises" has as domain the class "Person". No definition of "supervises" as a object or datatype property is provided. Note that a dashed diamond is used to represent the ontology element that is not defined as property in the graphical example. This alternative notation is used for the sake of clarity. A regular diamond is not used as it would be confused with elements actually defined as properties in the ontology.

How to solve it

Identify whether the ontology element affected represents an object property (relationship) or a datatype property (attribute):

(a) The element affected represents an object property if the range of the property is a class axiom or a set of individuals. If this is the case, add an statement to define the ontology element as an object property by means of the primitive `owl:ObjectProperty`.

(b) The element affected represents a datatype property if the range of the property is a datatype. If this is the case, add an statement to define the ontology element as a datatype property by means of the primitive `owl:DatatypeProperty`.

If you do not have enough knowledge to decide whether the given property is an object or datatype property, then the primitive `rdf:Property` could be used.

References	[1] Hogan, A., Harth, A., Passant, A., Decker, S., and Polleres, A. (2010). Weaving the pedantic web. In Proceedings of the WWW2010 Workshop on Linked Data on the Web, LDOW 2010, Raleigh, USA, April 27, 2010.

Table 4.35: P35. Untyped property

Title	P36. URI contains file extension	**Importance level**	Minor
Aspects	Application context	**Affects to**	Ontology
Description			
This pitfall occurs if file extensions such as ".owl", ".rdf", ".ttl", ".n3" and ".rdfxml" are included in an ontology URI. This pitfall is related with the recommendations provided in [1].			
Examples			
Graphical representation and/or OWL code		Natural language description	
<http://example.org/def/myontology.owl#> rdf:type owl:Ontology .		The graphical example shows an ontology declaration header where the URI assigned to the ontology (http://example.org/def/myontology.owl) contains the file extension ".owl"	
How to solve it			
Rename the ontology URI excluding the file extension. If the ontology URI overlaps with the namespace where the ontology elements are defined, update the ontology elements already declared accordingly.			
References	[1] Archer, P., Goedertier, S., and Loutas, N. (2012). *D7. 1.3-study on persistent URIs, with identification of best practices and recommendations on the topic for the Mss and the EC*. PwC EU Services.		

Table 4.36: P36. URI contains file extension

Title	P37. Ontology not available on the Web	**Importance level**	Critical
Aspects	Application context Ontology understanding	**Affects to**	Ontology
Description			
This pitfall occurs when the ontology code (OWL encoding) or its documentation (HTML document) is missing when looking up its URI. This pitfall deals with the first point from the Linked Data star system that states "On the web" ([1] and [3]). Guidelines in [5] also recommends to "Publish your vocabulary on the Web at a stable URI". This pitfall is also related to the problems listed in [4] and [6].			
How to solve it			
Publish the ontology according to the best practices (see http://www.w3.org/TR/swbp-vocab-pub/). For further details see [2].			
References	[1] Bernes-Lee Tim. (2006). "Linked Data - Design issues". http://www.w3.org/DesignIssues/LinkedData.html [2] Berrueta, D., Fernández, S., and Frade, I. (2008). Cooking HTTP content negotiation with Vapour. In Worshop on Scripting for the Semantic Web 2008. [3] Heath, T. and Bizer, C. (2011). Linked Data: Evolving the Web into a Global Data Space. Morgan & Claypool, 1st edition. [4] Hogan, A., Harth, A., Passant, A., Decker, S., and Polleres, A. (2010). Weaving the pedantic web. In Proceedings of the WWW2010 Workshop on Linked Data on the Web, LDOW 2010, Raleigh, USA, April 27, 2010. [5] Vatant, B. (2012). Is your linked data vocabulary 5-star?. http://bvatant.blogspot.fr/2012/02/is-your-linked-data-vocabulary-5-star_9588.html [6] Vrandecic, D. (2010). *Ontology Evaluation*. PhD thesis.		

Table 4.37: P37. Ontology not available on the Web

Title	P38. No OWL ontology declaration	**Importance level**	Important
Aspects	Application context Ontology language Ontology metadata	**Affects to**	Ontology

Description

This pitfall consists in not declaring the `owl:Ontology` tag, which provides the ontology metadata. The `owl:Ontology` tag aims at gathering metadata about a given ontology such as version information, license, provenance, creation date, and so on. It is also used to declare the inclusion of other ontologies.

Examples

Graphical representation and/or OWL code	Natural language description
No explicit evidence of: <http://example.org/def/myontology#> rdf:type owl:Ontology .	The graphical example represents the lack of an ontology declaration header.

How to solve it

Add an ontology declaration header in the ontology and fill it in with the corresponding metadata: title, description, creators, important dates, license, and so on.
See [1] for guidelines and suggestions about suitable vocabularies for different types of metadata information.

References	[1] Vandenbussche, P.Y., Vatant, B. (2012). *Metadata Recommendations For Linked Open Data Vocabularies*. http://lov.okfn.org/Recommendations_Vocabulary_Design.pdf

Table 4.38: P38. No OWL ontology declaration

Title	P39. Ambiguous namespace	**Importance level**	Critical
Aspects	Application context	**Affects to**	Ontology

Description

This pitfall consists in declaring neither the ontology URI nor the xml:base namespace. If this is the case, the ontology namespace is matched to the file location. This situation is not desirable, as the location of a file might change while the ontology should remain stable, as proposed in [2].

Examples

Graphical representation and/or OWL code	Natural language description
No explicit evidence of: @base <http://example.org/def/myontology#> . <http://example.org/def/myontology#> rdf:type owl:Ontology .	The graphical example represents the lack of (a) the value for the base namespace and (b) an ontology declaration header, leading to ambiguity in the ontology URI.

How to solve it

(1) Define the ontology namespace within the ontology declaration field using `owl:Ontology` (see "P38. No OWL ontology declaration").
(2) Include the given URI in the base URI field. See [1] for further information about the ontology declaration.

References	[1] Vandenbussche, P.Y., Vatant, B. (2012). Metadata Recommendations For Linked Open Data Vocabularies. http://lov.okfn.org/Recommendations_Vocabulary_Design.pdf [2] Vatant, B. (2012). Is your linked data vocabulary 5-star?. http://bvatant.blogspot.fr/2012/02/is-your-linked-data-vocabulary-5-star_9588.html

Table 4.39: P39. Ambiguous namespace

<table>
<tr><td>Title</td><td>P40. Namespace hijacking</td><td>Importance level</td><td>Critical</td></tr>
<tr><td>Aspects</td><td>Application context</td><td>Affects to</td><td>Classes
Object properties
Datatype properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">It refers to reusing or referring to terms from another namespace that are not defined in such namespace. This is an undesirable situation as no information can be retrieved when looking up those undefined terms. This pitfall is related to the Linked Data publishing guidelines provided in [1]: “Only define new terms in a namespace that you control” and to the guidelines provided in [2].</td></tr>
<tr><td colspan="4">Examples</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2">Prefix(rdfs:=<http://www.w3.org/2000/01/rdf-schema#>)
Prefix(myontology:=<http://example.org/def/myontology#>)
SubObjectPropertyOf(myontology:myProperty rdfs:Property)</td><td colspan="2">An example of this pitfall is to use “http://www.w3.org/2000/01/rdf-schema#Property” that is not defined in the rdfs namespace (http://www.w3.org/2000/01/rdf-schema#), instead of using “http://www.w3.org/1999/02/22-rdf-syntax-ns#Property”, that is actually defined in the rdf namespace (http://www.w3.org/1999/02/22-rdf-syntax-ns#)</td></tr>
<tr><td colspan="4">How to solve it</td></tr>
<tr><td colspan="4">Replace the incorrect URI with the right one existing in the referenced ontology. If the URI does not exist in the external ontology to be referenced, remove the term and either (a) look for an equivalent term in an existing ontology or (b) create the term in your own namespace.</td></tr>
<tr><td>References</td><td colspan="3">[1] Heath, T. and Bizer, C. (2011). Linked Data: Evolving the Web into a Global Data Space. Morgan & Claypool, 1st edition.
[2] Vrandecic, D. (2010). Ontology Evaluation. PhD thesis.</td></tr>
</table>

Table 4.40: P40. Namespace hijacking

<table>
<tr><td>Title</td><td colspan="2">P41. No license declared</td><td>Importance level</td><td>Important</td></tr>
<tr><td>Aspects</td><td colspan="2">Ontology metadata</td><td>Affects to</td><td>Ontology</td></tr>
<tr><td colspan="5">Description</td></tr>
<tr><td colspan="5">The ontology metadata omits information about the license that applies to the ontology.</td></tr>
<tr><td colspan="5">Examples</td></tr>
<tr><td colspan="3">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="3">Prefix(dcterms:=<http://purl.org/dc/terms/>)
Ontology(<http://example.org/def/myontology.owl>

No explicit evidence of:
Annotation(dcterms:license
<http://creativecommons.org/licenses/by/>))</td><td colspan="2">The OWL code shows the lack of a statement within the ontology declaration field stating the license applicable to the ontology.</td></tr>
<tr><td colspan="5">How to solve it</td></tr>
<tr><td colspan="5">Include a statement containing the license information using any of the following properties: <code>dc:rights</code>, <code>dcterms:rights</code>, <code>dcterms:license</code>, <code>cc:license</code> or <code>xhv:license</code>.
For detailed guidelines and additional options see in [2] and [4].
To link to the RDF representation of different licenses, it is recommended to use the RDF license dataset provided in [1] and explained in [3].</td></tr>
<tr><td>References</td><td colspan="4">[1] RDF License dataset http://rdflicense.linkeddata.es/dataset/
[2] Rodríguez-Doncel, V., Gómez-Pérez, A., and Mihindukulasooriya, N. (2013). Rights declaration in linked data. In Proceedings of the Fourth International Workshop on Consuming Linked Data, COLD 2013, Sydney, Australia, October 22, 2013.
[3] Rodríguez-Doncel, V., Villata, S., Gómez-Pérez, A. A Dataset of RDF Licenses. In Proc. of the 27th Int. Conf. on Legal Knowledge and Information System (JURIX), R. Hoekstra (Ed.), ISBN 978-1-61499-467-1, pp. 187-189, IOS Press, 2014.
[4] Vandenbussche, P.Y., Vatant, B. (2012). Metadata Recommendations For Linked Open Data Vocabularies. http://lov.okfn.org/Recommendations_Vocabulary_Design.pdf</td></tr>
</table>

Table 4.41: P41. No license declared

4.3 Assigning importance levels to pitfalls

It is obvious that not all the pitfalls are equally important; their impact in the ontology will depend on multiple factors. For this reason, the pitfall catalogue includes information about how critical each pitfall is. We propose three importance levels:

- **Critical (1)**: It is crucial to correct the pitfall. Otherwise, it could affect the ontology consistency, reasoning and applicability, among other characteristics. For example, the consequences of "P19. Defining multiple domains or ranges in properties" could lead to logical inconsistences in the ontology, which represents a critical error when reasoning over an ontology with instances.

- **Important (2)**: Though not critical for ontology function, it is important to correct this type of pitfall. For example, the logical consequences of "P25. Defining

a relationship as inverse to itself" are the same as if such relationship were defined as symmetric.

- **Minor (3)**: It does not represent a problem. However, correcting it makes the ontology in better form and understandable. For example, pitfall "P22. Using different naming conventions in the ontology" is about the appearance of the ontology and does not compromise the proper ontology functioning.

These levels do not have clear boundaries in the sense that a particular pitfall in a level could be debatable depending on: 1) modelling decisions; 2) ontology requirements; and 3) context of use by an ontology application.

As example of modelling decisions, in this work we consider an important pitfall to miss the definition of domains and ranges for the properties. This may be arguable, for example, for those cases where ontology engineers are more interested in increasing the interoperability of the model to the detriment of explicit semantics or definitions richness. In such a case, it would be enough if the evaluators define the fact of defining domains and ranges as a minor pitfall instead of doing it as we propose here. In this way, we provide a starting point for ontology evaluation that could be adapted to users' particular requirements.

As example of pitfalls that depend on ontology requirements and context of use, we consider an ontology that should be published according to the Linked Data rules and principles. In this scenario, pitfalls "P37. Ontology not available on the Web", "P39. Ambiguous namespace", and "P40. Namespace hijacking" are crucial while they might not be important in the context of an isolated application where the ontology is not designed for a Linked Data application. Another pitfall related to Linked Data context is "P36. URI contains file extension". In this case, it may be consider a minor pitfall as it does not affect the correct functioning of the ontology.

At the moment of including the importance levels in the catalogue, 35 out of the 41 pitfalls were already defined and published. In order to attach importance levels to the pitfalls, a study was carried out in which participants (ontology experts, researchers and practitioners in the semantic web community and OOPS! users) had to fill in a questionnaire providing the following information:

- **Level of confidence**: how confident (s)he felt in the ontology evaluation or ontology modelling domains. The possible values were "Expert", "Medium confidence"' and "Low confidence".

- **Importance level of each pitfall**: there was one question per pitfall (from P01 to P35) where the user had to select the importance level of the given pitfall. The possible values were "Critical", "Important" and "Minor" (see above).

- **Which pitfalls are not important**: A list with all the pitfalls was provided and the users were asked to indicate which pitfall would never represent a problem for them (no just pitfalls that could be a problem only in some cases or under certain conditions).

- **Other comments**: A free text box for providing any comment or suggestions.

Researchers, mainly experts on ontology modelling or evaluation, within the semantic web community[38] and OOPS! users were invited to fill in the questionnaire. We received 54 responses to the questionnaire (28 from ontology experts, 22 from participants with medium confidence and 4 from participants with low confidence). The questionnaire continues available on-line[39] to allow the community to assess the level of importance of the pitfalls.

In order to assign importance levels to pitfalls according to the data gathered through the survey,[40] we followed the steps:

1. Assign weights to each expertise level:

   - Expert $\longrightarrow weight_j = 3$
   - Medium confidence $\longrightarrow weight_j = 2$
   - Low confidence $\longrightarrow weight_j = 1$

   Where j is each of the survey answers provided by the 54 participants.

[38]The call was launched through several mailing list used by the semantic web community and through particular emails sent to known OOPS! users.

[39]The questionnaire is available at http://goo.gl/SEddMN (last visited on the 14th November, 2015)

[40]The data generated from the survey responses and the different ranking calculations are available at the URL: http://goo.gl/0IkbS2 (last visited on the 14th November, 2015).

2. Assign values for responses. For a given pitfall the response provided by the participants are replaced by a numerical value:[41]

   - Pitfall marked as "Critical" $\longrightarrow value_{i,j} = 3$
   - Pitfall marked as "Important" $\longrightarrow value_{i,j} = 2$
   - Pitfall marked as "Minor" $\longrightarrow value_{i,j} = 1$
   - Pitfall marked as "Not important" $\longrightarrow value_{i,j} = 0$

   Where i is each of the pitfalls for what the ranking is calculated and j is each of the survey answers provided by the 54 participants.

3. Rank the pitfalls according to the well-known **weighted sum** technique using the following formula:

$$score_i = \sum_j value_{i,j} * weight_j \tag{4.1}$$

Table 4.42 shows the ranking resulting from the weighted sum technique. The scores shown in such table have been normalized.

Once the pitfalls are ranked, an interval should be defined in order to split the given ranking into 3 parts, one for each importance level. To do this, we have used a method based on the range of the weight values. More precisely, the range (highest weight – lowest weight) is divided into 3 parts. Concretely, the range of the weighted sum ranking is 0.0193 (0.0379 – 0.0186). The division of such range among 3 parts gives us an interval of 0.0064. Finally, the range of the ranking is split into 3 sub-ranges, resulting in the following intervals:

- Minor: from 0.0186 to 0.0250 (0.0186 + 0.0064)
- Important: from 0.0250 to 0.0314 (0.0250 + 0.0064)
- Critical: from 0.0314 to 0.0379

[41] For processing the data and ranking the pitfalls we have assigned the value 3 for critical pitfalls, so that they appear in the top positions. However, for assigning importance levels within the catalogue we have set the "critical" position in 1, since the critical pitfalls should be corrected in first place.

	Weighted sum	
	Order	**Normalized Score**
Critical (1)	P06. Including cycles in a class hierarchy	0.0379
	P19. Defining multiple domains or ranges in properties	0.0375
	P01. Creating polysemous elements	0.0367
	P03. Creating the relationship "is" instead of using "rdfs:subClassOf", "rdf:type" or "owl:sameAs"	0.0364
	P29. Defining wrong transitive relationships	0.0348
	P28. Defining wrong symmetric relationships	0.0344
	P31. Defining wrong equivalent classes	0.0343
	P05. Defining wrong inverse relationships	0.0342
	P14. Misusing "owl:allValuesFrom"	0.0341
	P27. Defining wrong equivalent properties	0.0340
	P15. Using "some not" in place of "not some"	0.0335
	P16. Using a primitive class in place of a defined one	0.0335
Important (2)	P23. Duplicating a datatype already provided by the implementation language	0.0303
	P24. Using recursive definitions	0.0303
	P12. Equivalent properties not explicitly declared	0.0301
	P34. Untyped class	0.0284
	P10. Missing disjointness	0.0283
	P35. Untyped property	0.0281
	P25. Defining a relationship as inverse to itself	0.0279
	P30. Equivalent classes not explicitly declared	0.0279
	P18. Overspecializing the domain or range	0.0272
	P26. Defining inverse relationships for a symmetric one	0.0272
	P17. Overspecializing a hierarchy	0.0267
	P11. Missing domain or range in properties	0.0252
Minor (3)	P04. Creating unconnected ontology elements	0.0248
	P09. Missing domain information	0.0245
	P33. Creating a property chain with just one property	0.0240
	P02. Creating synonyms as classes	0.0239
	P07. Merging different concepts in the same class	0.0234
	P21. Using a miscellaneous class	0.0222
	P32. Several classes with the same label	0.0219
	P13. Inverse relationships not explicitly declared	0.0201
	P22. Using different naming conventions in the ontology	0.0189
	P20. Misusing ontology annotations	0.0187
	P08. Missing annotations	0.0186

Table 4.42: Pitfalls from P01 to P35 ranked according to the weighted sum

In order to demonstrate that the ranking method selected is robust, we compared it with two other ranking methods, namely, the "lexicographic order" (Miettinen, 1999) and the "centroid function" (Barron and Barrett, 1996).

The ranking obtained for the **lexicographic order** is shown in Table 4.43. As the "lexicographic order" does not involve weights or ranges, it does not make sense to split

the range in this fashion. The lexicographic order is calculated as follows: first, the pitfalls are ordered according to the number of votes obtained for the choice "critical". The more votes attained, the higher the pitfall is placed in the ranking. For example, P06 is first with 47 votes.[42] When two or more pitfalls have the same number of votes in this category, the information about the next importance levels is used to break the tie. For example, P29 and P14 have 37 votes for the value "critical", so the votes for "important" are used, that is, the P14 is placed first with 12 votes, and P14 is next with 9 votes.

The ranking obtained for the **centroid function** is shown in Table 4.44. This ranking is calculated with the same formula used in the weighted sum but assigning different weights for the level of confidence of each of the participants. In this sense, the centroid function is in charged of providing more accurate weights to be used in such formula. For the case of the centroid function we have also calculated the intervals for the "Critical", "Important", and "Minor" categories in the same manner as explained for the weighted sum.

Once the three rankings were computed, we measured how similar the orders established for the list of pitfalls were. To do so, we calculated the Kendall coefficient (Winkler and Hays, 1985), being the values obtained for each pair of rankings the following.[43]

- Weighted sum – lexicographic order: 0.882352941
- Weighted sum – centroid function: 0.905882353
- Lexicographic order – centroid function: 0.929411765

We can observe that the three values are very high; this fact means that the rankings are very similar and proves that the decision of choosing the weighted sum does not affect significantly the final classification.

When a new pitfall is inserted in the catalogue, an importance level has to be assigned to it. This importance level is decided in conjunction with the pitfall catalogue creators and maintainers, experienced ontological engineers, and the users proposing

[42] See file "SurveyImportanceLevelsLexcicographicOrder.pdf" at http://goo.gl/0IkbS2 (last visited on the 14th November, 2015)

[43] The data and calculations for obtaining the coefficients are available at http://goo.gl/QeSyHX (last visited on the 14th October, 2015)

Pitfall	Critical (1)	Important (2)	Minor (3)	Not important
P06. Including cycles in a class hierarchy	47	4	3	0
P19. Defining multiple domains or ranges in properties	42	9	2	0
P03. Creating the relationship "is" instead of using "rdfs:subClassOf", "rdf:type" or "owl:sameAs"	41	7	6	0
P01. Creating polysemous elements	38	13	3	0
P29. Defining wrong transitive relationships	37	12	2	1
P14. Misusing 'owl:allValuesFrom"	37	9	6	1
P31. Defining wrong equivalent classes	36	13	3	0
P16. Using a primitive class in place of a defined one	35	11	6	1
P15. Using "some not" in place of "not some"	34	13	3	1
P27. Defining wrong equivalent properties	33	14	5	0
P28. Defining wrong symmetric relationships	32	18	2	0
P05. Defining wrong inverse relationships	29	20	5	0
P24. Using recursive definitions	24	23	3	1
P12. Equivalent properties not explicitly declared	23	24	6	0
P10. Missing disjointness	23	18	11	2
P23. Duplicating a datatype already provided by the implementation language	22	25	5	0
P34. Untyped class	21	21	9	2
P35. Untyped property	19	24	8	2
P11. Missing domain or range in properties	19	17	12	4
P25. Defining a relationship as inverse to itself	17	24	12	0
P26. Defining inverse relationships for a symmetric one	15	26	11	0
P18. Overspecializing the domain or range	15	26	9	3
P17. Overspecializing a hierarchy	15	23	13	2
P30. Equivalent classes not explicitly declared	14	33	5	0
P04. Creating unconnected ontology elements	14	22	13	5
P07. Merging different concepts in the same class	13	21	14	5
P02. Creating synonyms as classes	10	24	17	3
P09. Missing domain information	7	33	11	3
P33. Creating a property chain with just one property	7	29	16	0
P21. Using a miscellaneous class	7	24	20	2
P13. Inverse relationships not explicitly declared	7	20	22	4
P32. Several classes with the same label	6	26	18	2
P20. Misusing ontology annotations	4	21	20	8
P08. Missing annotations	2	24	19	9
P22. Using different naming conventions in the ontology	2	22	26	1

Table 4.43: Pitfalls from P01 to P35 ranked according to the lexicographic order

the given pitfall (if any). For the pitfalls P36 to P41, five experts in Ontological Engineering, vocabulary publication and licensing have defined the pitfalls and assigned their importance levels. As a result, the importance levels shown in Table 4.45 have been attached to each pitfall.

	Centroid function	
	Order	**Normalized Score**
Critical (1)	P19. Defining multiple domains or ranges in properties	0.0379
	P06. Including cycles in a class hierarchy	0.0375
	P01. Creating polysemous elements	0.0369
	P03. Creating the relationship "is" instead of using "rdfs:subClassOf", "rdf:type" or "owl:sameAs"	0.0365
	P28. Defining wrong symmetric relationships	0.035
	P29. Defining wrong transitive relationships	0.035
	P05. Defining wrong inverse relationships	0.0348
	P27. Defining wrong equivalent properties	0.0344
	P31. Defining wrong equivalent classes	0.0341
	P14. Misusing "owl:allValuesFrom"	0.034
	P15. Using "some not" in place of "not some"	0.0339
	P16. Using a primitive class in place of a defined one	0.0334
Important (2)	P23. Duplicating a datatype already provided by the implementation language	0.0303
	P24. Using recursive definitions	0.0303
	P12. Equivalent properties not explicitly declared	0.0297
	P34. Untyped class	0.0282
	P35. Untyped property	0.0278
	P25. Defining a relationship as inverse to itself	0.0278
	P18. Overspecializing the domain or range	0.0277
	P10. Missing disjointness	0.0277
	P30. Equivalent classes not explicitly declared	0.0274
	P17. Overspecializing a hierarchy	0.0271
	P26. Defining inverse relationships for a symmetric one	0.0277
	P04. Creating unconnected ontology elements	0.0251
Minor (3)	P11. Missing domain or range in properties	0.0247
	P09. Missing domain information	0.0245
	P33. Creating a property chain with just one property	0.0241
	P02. Creating synonyms as classes	0.0239
	P07. Merging different concepts in the same class	0.0233
	P21. Using a miscellaneous class	0.0225
	P32. Several classes with the same label	0.0219
	P13. Inverse relationships not explicitly declared	0.0197
	P22. Using different naming conventions in the ontology	0.0188
	P08. Missing annotations	0.0187
	P20. Misusing ontology annotations	0.0186

Table 4.44: Pitfalls from P01 to P35 ranked according to the centroid function

Taking into account the importance levels extracted from the survey and those levels assigned by ontology experts, we have created a final classification of pitfalls as shown in Figure 4.7.

	Pitfall
Critical (1)	P37. Ontology not available on the Web P39. Ambiguous namespace P40. Namespace hijacking
Important (2)	P38. No OWL ontology declaration P41. No license declared
Minor (3)	P36. URI contains file extension

Table 4.45: Importance levels for pitfalls assigned by experts.

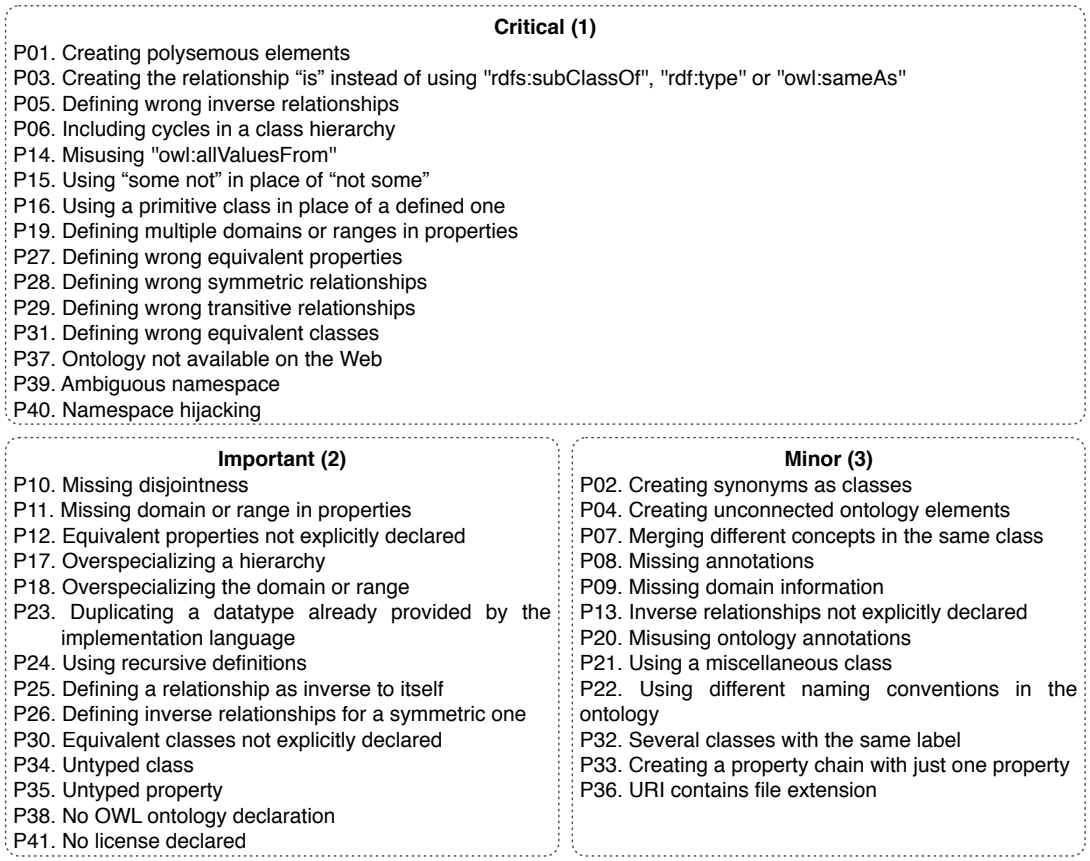


Figure 4.7: Classification of pitfalls by level of importance

4.4 Ontology pitfall classification

Since the list of pitfalls presented in Section 4.2 refers to different ontology perspectives, it is advisable to classify them according to some evaluation criteria. In this way, evaluators with an interest in a given aspect of ontology evaluation could easily identify

the group of pitfalls related to such aspect. For this reason, we have classified pitfalls according to the dimensions defined in (Gangemi et al., 2006), where a multi-layered approach to ontology evaluation is presented (see Section 2.4.6). The quality of an ontology may be measured relative to three main groups of dimensions, namely: structural, functional and usability-profiling. Even though these dimensions are enough to classify all the pitfalls in the catalogue, a more fine-grained classification is provided to deal with specific aspects that follow, extend and adapt the approach presented in (Poveda-Villalón et al., 2010). The aspects defined in (Poveda-Villalón et al., 2010) have been modified when necessary to represent a more trustworthy classification. The classification proposed in this thesis extends the classification in (Gangemi et al., 2006) as follows:

- **Structural dimension** (Gangemi et al., 2006): it is focused on syntax and formal semantics. We propose the following aspects (Poveda-Villalón et al., 2010):
  - **Modelling decisions**: this aspect involves evaluating whether developers use the primitives provided by ontology implementation languages in a correct way, and if there are modelling decisions that could be improved.
  - **No inference**: this aspect refers to checking whether desirable or expected knowledge could actually be inferred from the given ontology, but it is not inferred.
  - **Wrong inference**: this aspect refers to the evaluation of the inference of erroneous or invalid knowledge.
  - **Ontology language**: this aspect refers to checking whether the ontology is compliant both with the ontology language specification and with the syntax in which the ontology is formalized.
- **Functional dimension** (Gangemi et al., 2006): this is related to the intended use of a given ontology; thus the focus is on the ontology conceptualization. The following aspects are taken into account within this dimension:
  - **Real world modelling or common sense**: this aspect deals with the knowledge that domain experts expect to appear in the ontology, but it is not represented.

  - **Requirement completeness** (Poveda-Villalón et al., 2010): this aspect deals with the coverage of the requirements specified in the Ontology Requirements Specification Document.
  - **Application context**: this aspect refers to the adequacy of the ontology for a given application or use case.

- **Usability-profiling dimension** (Gangemi et al., 2006): it refers to the communication context of an ontology. For this dimension we contemplate the following aspects:

  - **Ontology understanding** (Poveda-Villalón et al., 2010): this aspect involves evaluating any kind of information that can help the user to understand the ontology content.
  - **Ontology clarity** (Poveda-Villalón et al., 2010): this aspect refers to the properties of ontology elements of being easily recognizable and understood by the user.
  - **Ontology metadata**: this aspect involves evaluating the existence of information that can help to understand the ontology context itself, instead of the conceptualization defined. Some examples of this information are: licensing, provenance, versioning, etc.

For each pitfall presented in Section 4.2 at least one of the above mentioned aspects has been assigned. In this way, one pitfall could belong to more than one aspect. For example, pitfall "P12. Equivalent properties not explicitly declared" (See Table 4.12) is classified both in the aspect "No inference" belonging to the structural dimension and in the aspect "Ontology understanding" belonging to the usability-profiling dimension. This is due to the double implications of such pitfall, that is, when it appears some inference might be lost and also the fact of not informing the user that two properties represent the same intended meaning relations might hinder the ontology understanding.

The aspect(s) in which the pitfall is classified have been assigned by the ontology experts creating and maintaining the catalogue. In case of pitfalls included in collaboration with other domain or ontology experts or OOPS! users, this classification is also done collaboratively by reaching an agreement.

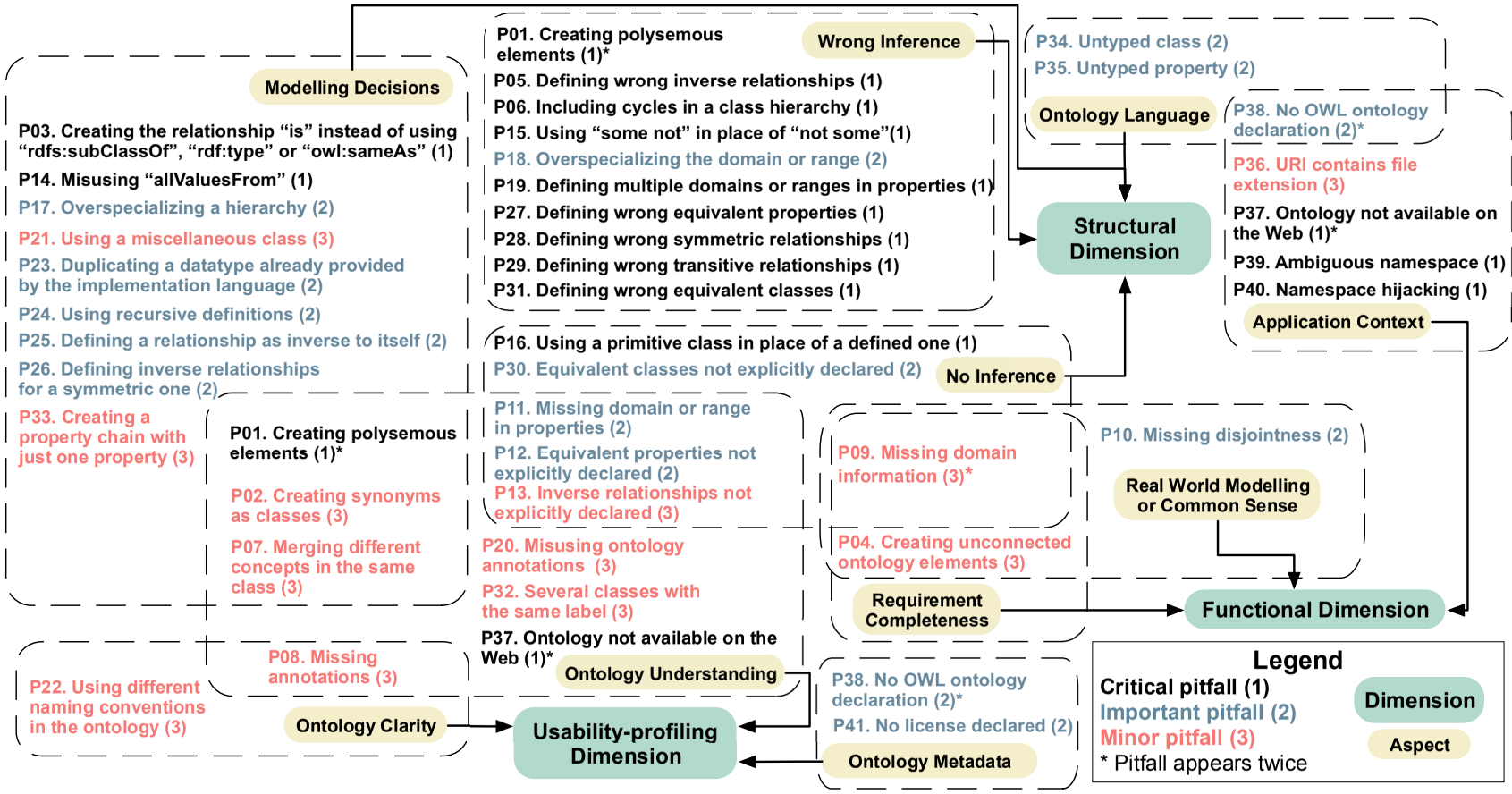


Figure 4.8: Pitfall classification. Based on (Poveda-Villalón et al., 2014)

Taking this into account, Figure 4.8 (Poveda-Villalón et al., 2014) represents the final classification at the time of writing the thesis for the 41 described pitfalls. Figure 4.8 also shows the importance level of each pitfall (see Section 4.3) both by attaching a number between brackets to each pitfall title and by using different colours; thus "critical" pitfalls are written in black followed by "(1)", "important" pitfalls are in blue followed by "(2)" and "minor" pitfalls are in pink followed by "(3)".

4.5 Workflow for pitfall catalogue update

As already mentioned in Section 3.3, the catalogue does not pretend to be exhaustive and it might be extended with new pitfalls. Figure 4.9 represents the proposed **workflow for including new pitfalls in the catalogue**. Such a figure also reminds the evolution of the catalogue and the implementation of detection methods in OOPS!. The ontology experts' suggestions of new pitfalls might stem from manual ontology inspection, literature review or collaborations among ontology experts. OOPS! users, who could be ontology engineers, ontology developers, domain experts, among others, can also suggest new pitfalls to be included in the catalogue. The **workflow for including new pitfalls into the catalogue** consists of the following activities:

1. OOPS! users or ontology experts propose to include a new pitfall in the catalogue.
2. Ontology experts maintaining the catalogue should review the proposed pitfall and decide whether to include it. The following activities apply when the pitfall is accepted.
   - 2.1 Link the pitfall to existing bibliography or references whenever possible.
   - 2.2 Classify the pitfall according to aspects and dimensions listed in Section 4.4. It could be possible to extend the aspects defined for a given dimension if the new pitfall does not fit any of the aspects considered. One pitfall could be classified in one or more aspects.
   - 2.3 Assign the importance level (critical, important or minor) to the pitfall in regard to the possible negative consequences of each pitfall in the ontology.

In the best case scenario, a detection method could be designed and implemented in OOPS! in order to detect the new pitfall, so that users could detect it automatically.

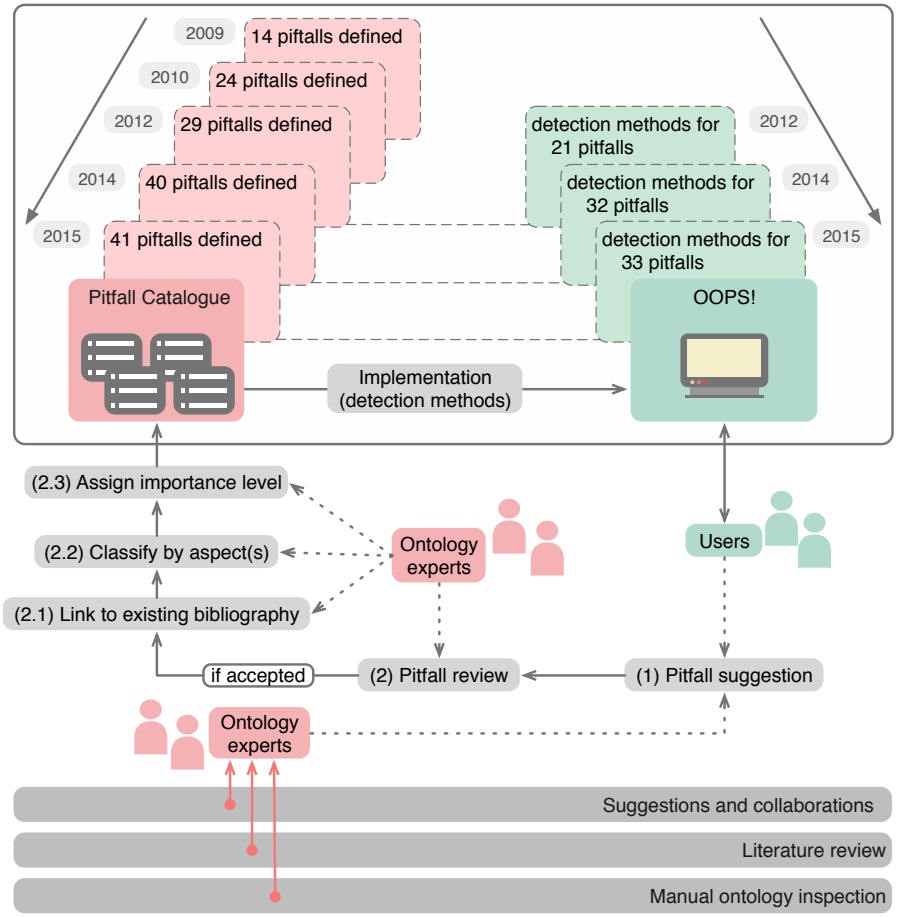


Figure 4.9: Workflow for pittall catalogue maintenance and evolution

4.6 Quality model for ontology diagnosis

A coetaneous thesis to this thesis is being developed in the field of semantic technology recommendations (Radulovic, 2016). Radulovic's thesis includes, among others, the following contributions: (a) quality model for semantic technologies and (b) RIDER, a generic recommendation software based on the Analytic Network Process (ANP) designed to automatically exploit evaluation results and user quality requirements. The quality model has been implemented by means of two ontologies, namely, the Quality Model Ontology (QMO)[44] and the Evaluation Result Ontology (EVAL).[45]

As indicated in (Vrandečić, 2010), ontology evaluation may be expressed by an

[44]http://purl.org/net/QualityModel (last visited on the 16th November, 2015)
[45]http://purl.org/net/EvaluationResult (last visited on the 16th November, 2015)

ontology, so that the results can be processed by the same tools for processing ontologies. In addition, the results of several different methods could be integrated in order to build complex evaluations out of a number of single evaluations. For this reason, and due to the fact that the QMO and EVAL ontologies are available online and already used in RIDER (Radulovic et al., 2013) we opt for align the catalogue of pitfalls presented in this work to such ontologies.

The objective of this section is to align the catalogue of pitfalls described in Section 4.2 with the ontology model proposed in (Radulovic, 2016). In this sense, Section 4.6.1 describes the QMO and EVAL ontologies that support Radulovic's quality model. Section 4.6.2 presents the instantiation of the QMO and EVAL ontologies for the catalogue of pitfalls.

4.6.1 Quality model for semantic technologies

Quality models represent a set of product's quality characteristics, sub-characteristics, quality measures, and the relationships between them, providing the basis for product evaluation while giving a better insight of the characteristics that influence product quality (Radulovic et al., 2015).

Radulovic et al., have proposed a quality model for semantic technologies (Radulovic, 2016; Radulovic et al., 2015) adapting the quality model defined in ISO 25010 (ISO, 2011a) standards. Such model is supported by the QMO ontology which describes its elements (e.g. quality characteristics, quality measures, etc.). Radulovic et al., have also developed the EVAL ontology for describing evaluation results. This ontology enables capturing knowledge of values obtained in a given evaluation. EVAL ontology extends QMO. An overview of both ontologies and how they are related is shown in Figure 4.10.

The `qmo:QualityMeasure` class represents the quality measure, which has as subclasses `qmo:BaseMeasure`, `qmo:DerivedMeasure` and `qmo:QualityIndicator`. A quality measure measures a quality characteristic, represented by the `qmo:QualityCharacteristic` class, and the two classes are connected by means of the `qmo:measuresCharacteristic` and `qmo:isMeasuredWith` properties.

The `eval:Evaluation` class represents the process of evaluating a particular product, service or action (`eval:EvaluationSubject` class). Each evaluation produces a quality value (represented by the `eval:QualityValue` class) which is related to a particular quality measure (`qmo:QualityMeasure`), using some input data (e.g, a document)

which is represented by the `eval:EvaluationData` class. Finally, each evaluation subject belongs to a certain category of products, services or actions, which is represented by the `eval:SubjectCategory` class.

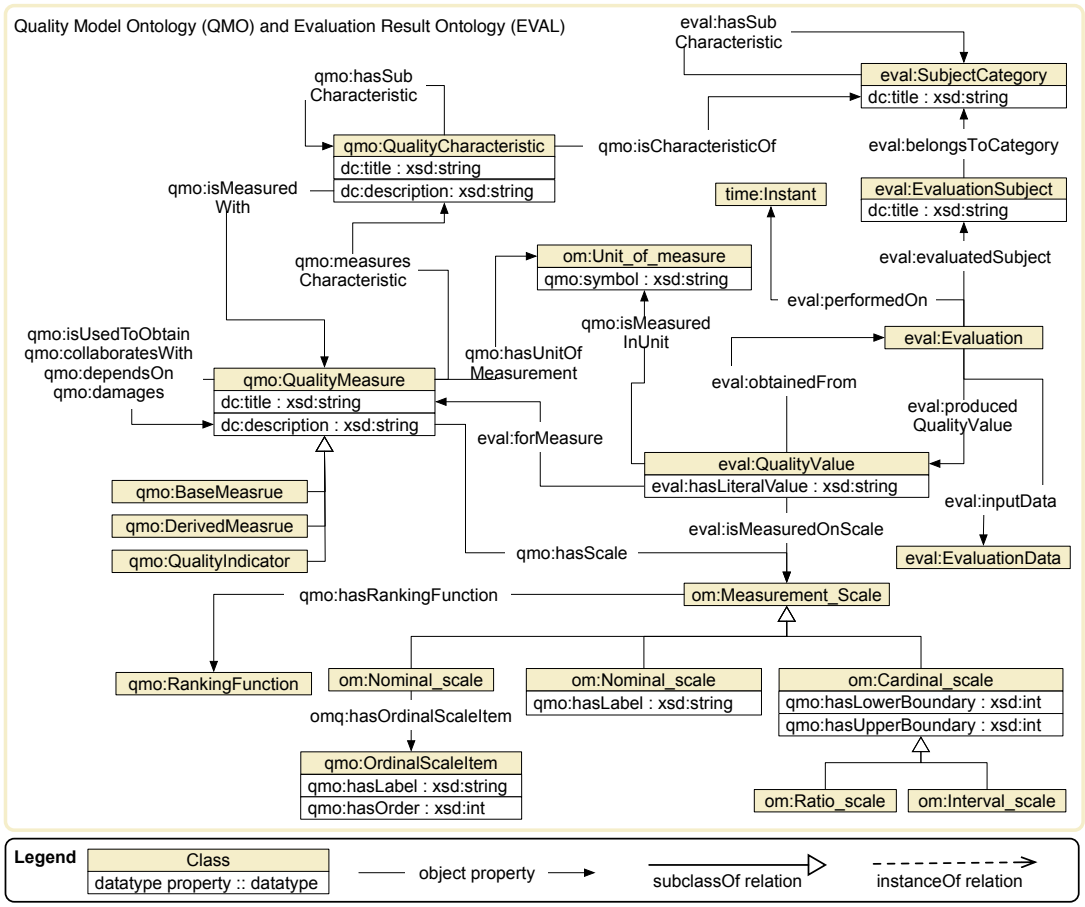


Figure 4.10: Overview of QMO and EVAL ontologies. Adapted from (Radulovic, 2016)

4.6.2 Quality model alignment for ontology diagnosis

A quality model in the domain of ontology evaluation could provide not only a guide on how to evaluate ontologies, but also a reference for researchers and practitioners for producing consistent, easily integrated and comparable ontology evaluations. For this reason, an instantiation of a quality model aligning the catalogue of pitfalls with existing QMO and EVAL ontologies has been developed and implemented as part of

this thesis. The resulting model[46] is shown in Figure 4.11 which includes the following information:[47]

- In the figure the "Ontology" subject is represented as an instance of the class `qmo:SubjectCategory`.
- The instances belonging to the class `qmo:QualityCharacteristic` are represented by the general characteristic "OntologyQuality" that has as sub-characteristics the three dimensions defined in Section 4.4, namely "Structural dimension", "Functional dimension" and "Usability-profiling dimension".
- Each dimension has as sub-characteristics the corresponding aspects defined in Section 4.4. Such quality characteristics are measured by means of quality measures (`qmo:QualityMeasure`) that corresponds to the pitfalls presented in this thesis. This information is represented by the relation `qmo:isMeasuredWith` between the ten aspects defined and the pitfalls grouped at the right of the figure.
- The inner box represent the group of pitfalls that are object of the `qmo:isMeasuredWith` property for a given quality characteristic.
- The broader box represent that all the pitfalls are instances of `qmo:QualityMeasure` and instances of `qmo:QualityIndicator`.
- At the bottom of the figure the pitfalls are linked to the scales in which they are measured. As it can be observed, a group of seven pitfalls (namely, P10, P22, P36, P37, P38 , P39 and P41) have scale boolean, which an instance of `om:Nominal_scale`. The labels assigned to this scale are "true" or "false" depending whether the pitfall appears or not. The rest of pitfalls have as scale a cardinal scale (either interval or ratio) which lower boundary is zero, and which upper boundary should be a natural number. In this case the instance representing such scale is of type `om:Cardinal_scale`.

[46]The quality model for ontology diagnosis based on the catalogue of pitfalls is available at http://oops.linkeddata.es/ontology/qmo (last visited on the 20th January, 2016)

[47]For the sake of clarity individuals are represented with ellipses instead of boxes (as presented in the UML_Ont profile in Section 4.2) as it is easier to identify them at a glimpse with a different geometry from the one used for classes.

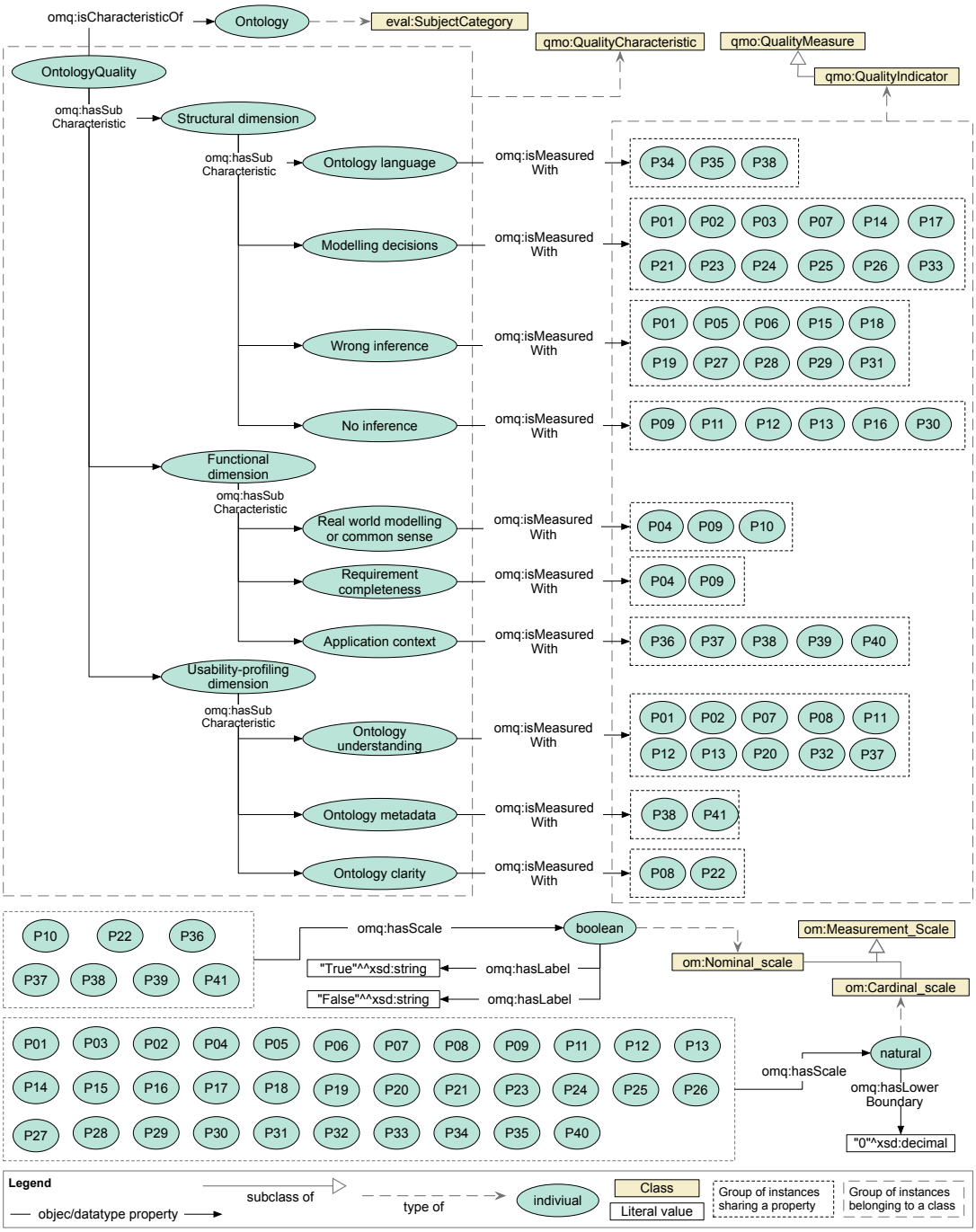


Figure 4.11: Pitfall catalogue for ontology diagnosis aligned to QMO and EVAL ontologies. Available at http://oops.linkeddata.es/ontology/qmo

It should be mentioned that the rest of the elements appearing in EVAL, are not needed at this step of defining the general framework. Such elements would be instantiated when annotating a particular evaluation of a given ontology, when the values for the indicators and for the measurement scales are known. For example, the attribute `qmo:hasUpperBoundary` can be stated once the number of classes, object properties and datatypes properties are known.

Out of scope of this thesis is the application of the quality model presented to the pitfalls previously evaluated with OOPS!. However, it is planned to integrate this model with OOPS! in future releases.

4.7 Summary

This chapter has presented the first contribution of the thesis, the **C1: catalogue of common pitfalls** (see Section 3.2). This contribution includes a general description template, used later on for describing the current 41 pitfalls in the catalogue. Such template is also proposed to be used in the future when incorporating new pitfalls to the catalogue according to the workflow presented in this chapter. This contribution might be considered as the backbone of this thesis providing the foundations for Chapter 5.

As part of the pitfall catalogue development, the process of assigning importance levels to the pitfalls has been presented. Such process involved the elaboration of a survey answered by 54 ontology development community members what has been used to assign the importance level to 35 pitfalls. It also includes the process followed for assigning new importance levels to the incoming pitfalls. This is done by agreement between the pitfall catalogue creators and maintainers, experienced ontological engineers, and the users (if any) proposing the given pitfall, as it has been the case for the latest 6 pitfalls.

The catalogue of common pitfalls also includes a classification of pitfalls according to existing ontology evaluation dimensions and the extension of such dimension including ontology evaluation aspects.

The second contribution of this thesis presented in this chapter is the quality model for ontology diagnosis. This contribution, **C2: Quality model of ontology diagnose** (see Section 3.2), has been carried out mainly by means of instantiating the Quality Model Ontology.

In summary, this chapter has presented the work developed in this thesis in order to address the first objective **O1: To help ontology engineers to diagnose their ontologies in order to find common pitfalls** (see Section 3.1) advancing therefore, in the state of the art for ontology diagnosis activity.

Chapter 5

Methods and technological support: OntOlogy Pitfall Scanner!

5.1 Introduction

Evaluating ontologies is a crucial and complex activity to be carried out in every ontology development project. Despite the vast amounts of frameworks, criteria, and methods for ontology evaluation (see Chapter 2) such activity is still largely neglected by developers and practitioners. Having a list of common mistakes with orientations and guides about how to evaluate ontologies is of great help for developers . However, manually going through the ontology in order to diagnose potential errors is a always a tedious and time-consuming task. Therefore, it is advisable and needed to focus on the development of technological supports that assist ontology practitioners to detect such pitfalls in a guided, easier and faster way.

In this chapter, we present OOPS! (OntOlogy Pitfall Scanner![48]), an on-line service intended to help ontology developers, mainly newcomers, during the ontology diagnose activity. OOPS! (a) is executed independently of any ontology development platform without configuration or installation; (b) works with main web browsers;[49] (c) enlarges the list of errors detected by most recent and available tools (see Section 2.5); and (d) provides both a web-based human interface and a REST service to be used by applications. To this end, OOPS! supports the (semi-)automatic diagnosis of OWL ontologies and it is based on the pitfall catalogue approach presented in Chapter 4.

[48] http://oops.linkeddata.es/

[49] Firefox, Chrome, Safari and Internet Explorer

It is worth noting that the ontology repair activity is out of scope of OOPS! at the moment of writing this dissertation. Even though the pitfall catalogue provides some indications for solving pitfalls' problems such feature is not included in OOPS! as it would imply the system to have ontology edition capabilities. At this stage the system consists in a stand alone application that suggest the ontology engineer some options to correct the ontology, requiring always human intervention in the repair activity.

This chapter is organized as follows. The architecture behind OOPS! is presented in Section 5.2 while Section 5.3 is devoted to list the detection methods implemented for 33 pitfalls.

5.2 System organization

When addressing the development of OOPS!, the following architectural and technical decisions were made. First of all, we decided not to implement the system as a plugin for any specific ontology editor so that the range of user is not limited to the users of the editor(s) chosen. In this way, OOPS! could be suitable for a broader range of users if no specific ontology editor had to be installed first. Then, the system was design as a web application so that no installation process is needed. In addition, this decision allows us to keep one reference and updated version of the system instead of spreading different releases (often outdated) difficult to track and notify for every update.

Figure 5.1 presents the underlying OOPS! architecture. This architecture is based on three layer, namely presentation, business and persistence layers.

The **presentation layer** allows the interaction with the system both for humans and machines. The system is accessible to humans by means of a **web user interface** and to machines throughout a **web REST service**. These interfaces are detailed in Annex A. The input ontology could be entered by its URI or the OWL code.[50] Once the otology is analysed the results are presented to the user. For each pitfall the following information is provided: (a) the code and title of the pitfall: (b) the number of occurrences of the pitfall in the ontology; (c) the importance level; (d) the pitfall description and a generic example; and (e) the list of ontology elements affected by the pitfall in the ontology.

[50] At the moment of writing this document the serializations accepted are RDF/XML and turtle.

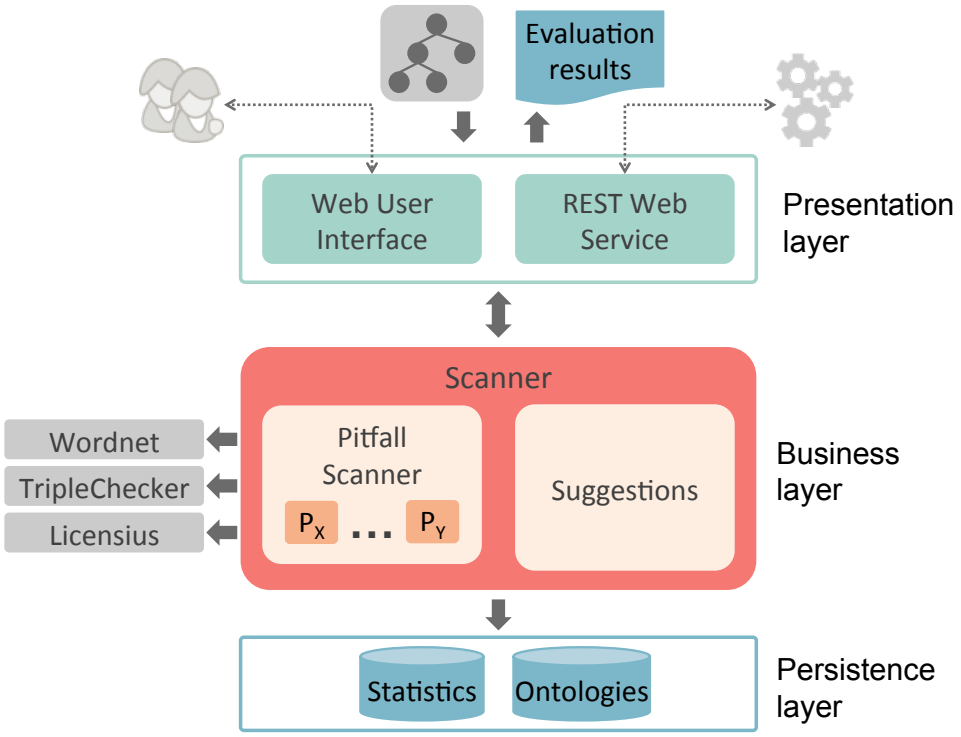


Figure 5.1: OOPS! architecture overview

The **business layer** is the one in charge of producing the evaluation results. For doing so, it takes as input the ontology to be analysed. Once the ontology is loaded into the system using and RDF parser,[51] the **Pitfall Scanner** module inspects the ontology looking for pitfalls among those implemented. At the moment of writing this thesis, 33 out of the 41 pitfalls defined in the catalogue are implemented. The 33 pitfalls implemented are those that can be detected (semi-)automatically with the information provided by the ontology OWL code (T-box). Those pitfalls that require an external reference framework (e.g. an ontology requirement document, an A-box or corpora, and/or domain knowledge) or human intervention are not yet automated as already mentioned in Section 3.5.

Table 5.1 shows complete list of correspondences between pitfalls, the tables where they are described in Chapter 4 and the tables where the detection methods are provided in Chapter 5. For those pitfalls that have not been implemented yet Table 5.1 provides

[51]The RDF parser used is Jena API http://jena.sourceforge.net (last visited on the 14th October, 2015)

the reason why a given pitfall is not detected.

This module is implemented following a modular approach in a way that: (a) every implemented pitfall is coded in a separated Java class; and (b) each pitfall defines it own business logic independently of the other pitfalls. For example, there is an overlap of situations detected by pitfall "P25. Defining a relationship as inverse to itself" and pitfall "P26. Defining inverse relationships for a symmetric one". In this sense, if a symmetric object property is defined as inverse of itself, OOPS! will detect both pitfalls. In addition, the system allows not only analysing all the automated pitfalls, but also choosing specific pitfalls or predefined groups according to the pitfall classification presented in this Figure 4.8.

As shown in Figure 5.1 some external resources are used during the pitfall detection, namely WordNet,[52] TripleChecker[53] and Licensius.[54] The detection methods that will make use of these resources and the roles played by them in OOPS! will be explain in more detail in Section 5.3.

During the pitfall scanning phase, the ontology elements prone to potential errors are detected, whereas some modelling suggestions are generated by the **Suggestions** module. More precisely, this module so far looks for object properties that might be defined as transitive or symmetric properties and present such option to the user.

Regarding the **persistence layer**, the system keeps track of the number of pitfalls appearing in each evaluated ontology and stores the ontologies themselves in the case of the user allows the system to do so. Otherwise, the ontology source code is not stored.

Next section describes the methods for detecting pitfalls in detail and the different approaches followed in order to implement such methods.

5.3 Detection methods

Gangemi and colleagues, discussing the matching problem and how to measure the extent to which an ontology reflects a given area of interest, pointed out that *"This seems to imply that no automatized method will ever suffice and that intellectual judgement will always be needed. However, automatic and semi-automatic techniques can be applied that make evaluation easier, less subjective, more complete and faster."* (taken literarily from

[52] http://wordnet.princeton.edu/ (last visited on the 21st September, 2015)
[53] http://graphite.ecs.soton.ac.uk/checker/ (last visited on the 21st September, 2015)
[54] http://licensius.appspot.com/ (last visited on the 21st September, 2015)

Pitfall	Pitfall description	Method description
P01. Creating polysemous elements	Table 4.1	Needs background knowledge.
P02. Creating synonyms as classes	Table 4.2	Table 5.2
P03. Creating the relationship "is" instead of using "rdfs:subClassOf", "rdf:type" or "owl:sameAs"	Table 4.3	Table 5.3
P04. Creating unconnected ontology elements	Table 4.4	Table 5.4
P05. Defining wrong inverse relationships	Table 4.5	Table 5.5
P06. Including cycles in a class hierarchy	Table 4.6	Table 5.6
P07. Merging different concepts in the same class	Table 4.7	Table 5.7
P08. Missing annotations	Table 4.8	Table 5.8, Table 5.9 and Table 5.10
P09. Missing domain information	Table 4.9	Needs ontology requirements document.
P10. Missing disjointness	Table 4.10	Table 5.11
P11. Missing domain or range in properties	Table 4.11	Table 5.12 and Table 5.13
P12. Equivalent properties not explicitly declared	Table 4.12	Table 5.14 and Table 5.15
P13. Inverse relationships not explicitly declared	Table 4.13	Table 5.16
P14. Misusing "owl:allValuesFrom"	Table 4.14	Needs background knowledge and human intervention.
P15. Using "some not" in place of "not some"	Table 4.15	Needs background knowledge and human intervention.
P16. Using a primitive class in place of a defined one	Table 4.16	Needs background knowledge.
P17. Overspecializing a hierarchy	Table 4.17	Needs background knowledge and/or ontology requirements document.
P18. Overspecializing the domain or range	Table 4.18	Needs background knowledge and/or ontology requirements document.
P.19 Defininining multiple domains or ranges in properties	Table 4.19	Table 5.17 and Table 5.18
P20. Misusing ontology annotations	Table 4.20	Table 5.19, Table 5.20 and Table 5.21
P21. Using a miscellaneous class	Table 4.21	Table 5.22
P22. Using different naming conventions in the ontology	Table 4.22	Table 5.23, Table 5.24, Table 5.25 and Table 5.26
P23. Duplicating a datatype already provided by the implementation language	Table 4.23	Needs background knowledge and human intervention
P24. Using recursive definitions	Table 4.24	Table 5.27 , Table 5.28 and Table 5.29
P25. Defining a relationship as inverse to itself	Table 4.25	Table 5.30
P26. Defining inverse relationships for a symmetric one	Table 4.26	Table 5.31
P27. Defining wrong equivalent properties	Table 4.27	Table 5.32 and Table 5.33
P28. Defining wrong symmetric relationships	Table 4.28	Table 5.34
P29. Defining wrong transitive relationships	Table 4.29	Table 5.35
P30. Equivalent classes not explicitly declared	Table 4.30	Table 5.36
P31. Defining wrong equivalent classes	Table 4.31	Table 5.37
P32. Several classes with the same label	Table 4.32	Table 5.38
P33. Creating a property chain with just one property	Table 4.33	Table 5.39
P34. Untyped class	Table 4.34	Table 5.40 and Table 5.41
P35. Untyped property	Table 4.35	Table 5.42 and Table 5.43
P36. URI contains file extension	Table 4.36	Table 5.44
P37. Ontology not available on the Web	Table 4.37	Table 5.45
P38. No OWL ontology declaration	Table 4.38	Table 5.46
P39. Ambiguous namespace	Table 4.39	Table 5.47
P40. Namespace hijacking	Table 4.40	Table 5.48
P41. No license declared	Table 4.41	Table 5.49

Table 5.1: Correspondences between pitfall descriptions and detection methods

(Gangemi et al., 2006) page 6). Being aware of such limitation of ontology evaluation automation, we have addressed the detection of as many pitfalls as possible from those defined in Section 4.2.

It is worth mentioning that even though some pitfalls might have been identified or discusses by other authors, most of the detection methods here shown are pure contributions of this thesis. However, in some cases, external resources are used to implement the method partly or completely. Such cases are appropriately indicated in the corresponding method descriptions.

The pitfall catalogue covers many different aspects of ontologies, such as their internal structure, their associated or embedded human-readable documentation, or their availability on the Web. As a consequence, the detection methods implemented make use of different techniques and technologies for diagnosing them. More precisely, the detection methods used within OOPS! are based on one (or more) of the following approaches:

- **Structural pattern matching**: the detection methods based on structural patterns inspect the internal structure of the ontology, analysing specific parts of the model. In these cases, a pitfall is flagged when a given structural pattern is spotted. Of the 33 pitfalls, 24 have been implemented using structural patterns. An example of method using structural pattern matching is the one developed for the pitfall "P05. Defining wrong inverse relationships" (see method in Table 5.5) in which the pattern addressed consists of a pair of inverse object properties where the domain of one of the properties does not match the range defined for the other one.

- **Linguistic analysis**:[55] the detection methods based on the analysis of language-related items make use of the content of annotations (e.g. `rdfs:label` or `rdfs:comment`) and identifiers (the ID part of the element URI) for detecting pitfalls. These methods are used in 9 out of the 33 implemented pitfalls. An example of a method using linguistic analysis is the one developed for the pitfall "P07. Merging different concepts in the same class" (see method in Table 5.7) that looks for concepts

[55]This technique has been renamed from "Lexical content analysis" to "Linguistic analysis" is order to gather a broader range of techniques related to linguistic aspects, not only the meaning of the terms themselves as indicated by the prior naming.

containing the syntagmatic structure "and" or "or" that they do not appear in any synset in WordNet (Miller and Fellbaum, 1998).

- **Specific characteristic search**: six detection methods have been automated by checking general characteristics of the ontology related neither to its internal structure nor to the content of the lexical entities. These characteristics could be related, for example, to the name given to the ontology as in the pitfall "P36. URI contains file extension" (see method in Table 5.44). This pitfall is detected when the ontology URI refers to the technology or ontology language used during its development as RDF or OWL.

In addition, some pitfalls can appear several times in the same ontology while others may appear at most once, since they affect the whole ontology instead of particular classes, object properties or datatype properties. Figure 5.2 shows how many times a given pitfall could be spotted in a given ontology and the type of technique(s) used for detecting each pitfall. For example, pitfall "P11. Missing domain or range in properties" (see methods in Table 5.12 and Table 5.13) is detected by seeking a given structural pattern and that it could appear as many times as relations are defined in the ontology.

Technique \ Cardinality	0..1 Appears at most once	0..N Could appear more than once
Structural pattern matching	P10	P02, P04, P05, P06, P08, P11, P13, P19, P24, P25, P26, P27, P28, P29, P33, P34, P35
Linguistic analysis	P22	P03, P12, P20, P30, P31, P32 P07, P21
Specific characteristic search	P36, P37, P38, P39, P41	P40

Figure 5.2: Classification of pitfalls based on the techniques used for their diagnoses

There are cases where a detection method uses more than one technique as indicated in Figure 5.2, by means of rectangles located between two cells. For example, to detect

"P30. Equivalent classes not explicitly declared" (see method in Table 5.36), OOPS! seeks a structural pattern for detecting the lack of equivalence between classes. For each pair of classes that are not defined as equivalents, it is checked whether the concepts they represent could be synonyms according to WordNet and possible equivalences between classes are proposed to the user. For "P31. Defining wrong equivalent classes" (see method in Table 5.37) exactly the opposite is checked, looking whether two concepts that are defined as equivalents and they are not considered synonyms in WordNet.

It should be noted that for some pitfalls, the detection methods applied might not cover all the possible situations in which a pitfall occurs, but a subset of them. While pitfall "P11. Missing domain or range in properties" is detected in all possible cases by the patterns presented in Table 5.12 and Table 5.13, it is not the case for pitfall "P05. Defining wrong inverse relationships". In P05 the patterns presented in Table 5.5 will not cover the cases in which some background and common sense knowledge is needed. For example, the pattern will not flag a pitfall when in a math ontology, the relationship "lessThan" is defined as inverse of "greaterThanOrEqual". We plan to improve these methods by incorporating natural language processing techniques and resources as proposed in (Suárez-Figueroa et al., 2013b).

In other cases, as already mentioned in Section 4.2, a detected pitfall might not represent a factual error, and this might be due to specific modelling decision or requirements. For example, "P02. Creating synonyms as classes" (see method in Table 5.2) might be implemented in some cases in order to support backwards compatibility between different versions of the same ontology. Another example is the case of "P11. Missing domain or range in properties" (see method in Table 5.12 and Table 5.13). There might be cases in which developers leave domains and ranges open in order to foster interoperability, leaving behind precision in definition.

Taking all the above-mentioned situations into account, we will describe the detection methods implemented for 33 pitfalls as part of this work. For doing so, the template presented in Figure 5.3 will be followed. Such template contains the following fields (mandatory fields are marked with an * in the template), and the possible values are explained in Figure 5.3 itself:

- **Code**: this field represents the identifier of the method. For this work we will use a combination of "MX-" (where X is an ordinal number, for example "M1" for

Code*	Code of the method		
Pitfall cardinality*	{1 |N}	**Method addresses***	{Ontology | [Classes, Object properties, Datatype properties]}
Technique*	{[Structural pattern matching, Linguistic analysis, Specific characteristic search]}	**Pitfall affects to***	{Ontology | [Classes, Object properties, Datatype properties]}
Description*			
General explanation of what the method consists in based on the pitfall to be detected.			
Limitations			
Corner cases or well-known exceptions where the pitfalls are not currently detected. This field is optional.			
Patterns*			
Graphical representation and/or OWL code		Natural language description	
Graphical representation of the pattern following: (a) the adaptation of the UML_Ont profile defined for ontologies in the NeOn Deliverable D1.1.2 (Haase et al., 2009); or (b) source OWL encoding in functional syntax, depending on the suitability of each representation technique for each case. In case this field includes the representation of lack of information, such information is indicated by "*No explicit evidence of:*" preceding the given statements.		Natural language explanation of the pattern graphical representation or OWL code.	
External resources used			
Pointers to external libraries, systems or resources used within the method implementation.			

Figure 5.3: Detection methods description template

"Method 1") followed by the code of the pitfall to be detected and a title for the method related to the pitfall title.

- **Cardinality**: this field indicates whether the pitfall can appear at most once (value "1") or more (value "N").

- **Technique**: this field indicates the techniques (one or more) used by the method in order to detect the addressed pitfall. The possible values are "Structural pattern matching", "Linguistic analysis" or "Specific characteristic search".

- **Method addresses**: this field indicates whether the method in its current implementation addressed the detection of the pitfall for: (a) "Classes", "Object properties", or "Datatype properties"; or (b) a combination of them; or (c) in the "Ontology" itself. This field is related to the field "Pitfall affects to". This field has been added in order to provide more precise information about to what extent the method addressed the pitfall description. For example, the pitfall "P04. Creating unconnected ontology elements" (see method in Table 5.4) could affect classes,

object properties and datatype properties. However, the method proposed so far only addresses the case for classes.

- **Pitfall affects to**: this field reminds the reader whether the pitfall affects the ontology itself or particular ontology elements. Therefore the possible values are: "Ontology" or a combination of the "Classes", "Object properties" and "Datatype properties". Further information about these values was presented in Section 4.2.

- **Description**: this field gives an explanation about what the method consists in.

- **Limitations**: this field presents situations not covered by the method or other drawbacks. This field is optional.

- **Patterns**: this field presents the patterns detected by the method and it is divided into two sub-fields for each provided pattern:

  - **Graphical representation and/or OWL code**: this field provides a graphical representation following the pattern or an excerpt of OWL encoding. It is worth noting that the notation presented in Section 4.2 is slightly modified in this field. The modification affects to the primitives represented by double-sided dependency as `«owl:equivalentClass»`, `«owl:disjointWith»`, `«owl:equivalentProperty»` and `«owl:inverseOf»`. The difference consists in the representation of the dependency arrow, while in Section 4.2 they were depicted by means of a double-sided dependency as they are symmetric relations. In this section, a single-sided dependency will be used. The rationale behind such modification is that the detection methods are closer to the ontology implementation language, in which the statements are represented by triples of the form <subject, predicate, object>. In the patterns depicted in the detection methods we take into account whether the ontology elements are the subject or object of such relationships (that takes the role of predicate) while in the pitfalls described in Section 4.2 the semantic of the relationships are represented in a conceptual level.

  - **Natural language**: this field provides a natural language explanation of the presented graphical pattern.

- **External resources used**: this field indicates the external resources used to detect the pitfall. These resources could contribute: (a) partially to the detection method in the sense that they act as supporting resources or (b) the method relies entirely on such resource and without the resource the method could have not been implemented.

Optional fields will only be included in the table describing a given pitfall if there is a value to be included. Otherwise the field will not be shown in the table.

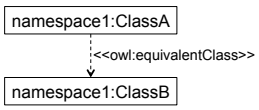
Code	M1-P02. Detecting synonyms created as classes		
Pitfall cardinality	N	**Method addresses**	Classes
Technique	Structural pattern matching	**Pitfall affects to**	Classes
Description			
This method iterates over the list of classes included in the ontology. If a class has an equivalent class (`owl:equivalentClass`) and both classes are defined in the same namespace (pattern A), then a pitfall is flagged.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A:  		The graphical pattern shows two classes defined in the same namespace between which an `owl:equivalentClass` statement holds.	

Table 5.2: Detection method proposed for P02

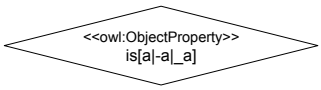
Code	M1-P03. Detecting the relationship "is"		
Pitfall cardinality	N	**Method addresses**	Object properties
Technique	Structural pattern matching Linguistic analysis	**Pitfall affects to**	Object properties
Description			
This method iterates over the list of object properties included in the ontology. If an object property's name is formed by the verb form "is" with or without the indefinite article "a" (possibly linked to the verb by a separator (e.g. "-", "_") as indicated in the pattern A), then a pitfall is flagged.			
Limitations			
This method addresses only the English language.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A:  		The graphical pattern shows the object property named "is" and the possibility of including the indefinite article "a" with an optional separator as part of the property.	

Table 5.3: Detection method proposed for P03

Code	M1-P04. Detecting unconnected classes		
Pitfall cardinality	N	**Method addresses**	Classes
Technique	Structural pattern matching	**Pitfall affects to**	Classes Object properties Datatype properties
Description			
This method iterates over the list of classes. For each class, the method checks whether the class: (a) is not part of any hierarchy (using `rdfs:subClassOf`) (b) does not have any definition axiom (using `rdfs:subClassOf` or `owl:equivalentClass` and a class restriction expression) (c) does not appear in any other class definition axiom (using `rdfs:subClassOf` or `owl:equivalentClass` and a class restriction expression) (d) does not appear in any property domain (`rdfs:domain`) or range (`rdfs:range`) If a class meets all the above-mentioned conditions, then a pitfall is flagged.			
Limitations			
This method only considers classes whereas the pitfall is defined also for object properties and datatype properties.			
Patterns			
No graphical pattern is provided for this method due to the recursive aspect of the method and the high number of combinations.			

Table 5.4: Detection method proposed for P04

<table>
<tr><td>Code</td><td colspan="3">M1-P05. Detecting wrong inverse relationships</td></tr>
<tr><td>Pitfall cardinality</td><td>N</td><td>Method addresses</td><td>Object properties</td></tr>
<tr><td>Technique</td><td>Structural pattern matching</td><td>Pitfall affects to</td><td>Object properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">This method iterates over the list of object properties in the ontology.
For each pair of inverse object properties (<code>owl:inverseOf</code>), the method checks whether:
(a) the domain of an object property and the range of its inverse object property are defined but they do not match each other (pattern A)
(b) the range of an object property and the domain of its inverse object property are defined but they do not match each other (pattern B)
If for a pair of inverse object properties at least one of the above-mentioned conditions is met, then a pitfall is flagged.

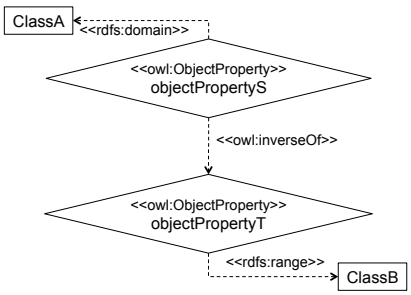
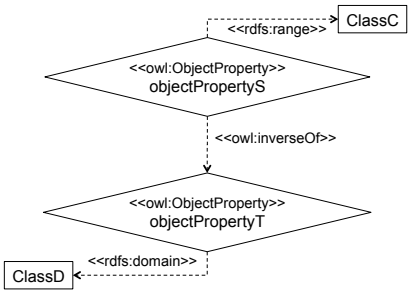
The method checks if the domain of one relationship and the range of the other relationship are the same class, and vice versa. If class restrictions apply, the method checks whether they are syntactically equivalent.</td></tr>
<tr><td colspan="4">Limitations</td></tr>
<tr><td colspan="4">The method does not check if two different class restrictions or anonymous classes logically represent identical sets of individuals.</td></tr>
<tr><td colspan="4">Patterns</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2">Pattern A:
 
</td><td colspan="2">The graphical pattern shows an object property whose domain is the class “ClassA” and which is defined as inverse of (<code>owl:inverseOf</code>) an object property whose range is the class “ClassB”.</td></tr>
<tr><td colspan="2">Pattern B:
 
</td><td colspan="2">The graphical pattern shows an object property whose range is the class “ClassC” and which is defined as inverse of (<code>owl:inverseOf</code>) an object property whose domain is the class “ClassD”.</td></tr>
</table>

Table 5.5: Detection method proposed for P05

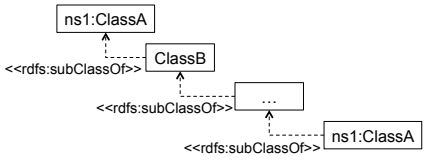
Code	M1-P06. Detecting cycles in a class hierarchy		
Pitfall cardinality	N	**Method addresses**	Classes
Technique	Structural pattern matching	**Pitfall affects to**	Classes
Description			
This method iterates over the list of named classes in the ontology. The method compares each class with the list of its superclasses. If a given class appears in its list of superclasses (pattern A), then a pitfall is flagged. Note that the check is done considering the class URI (namespace+identifier) not only the class identifier. The namespace of the rest of the classes in the hierarchy are indifferent.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A:  		The graphical pattern shows a hierarchy in which the "ClassA" (defined in a given namespace) appears in two different levels of a given branch in the hierarchy.	

Table 5.6: Detection method proposed for P06

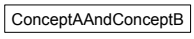
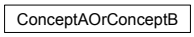
Code	M1-P07. Detecting different concepts merged in the same class		
Pitfall cardinality	N	**Method addresses**	Classes
Technique	Linguistic analysis	**Pitfall affects to**	Classes
Description			
This method iterates over the list of classes in the ontology. If a given class identifier contains the copulative conjunction "and" (pattern A) or the disjunctive conjunction "or" (pattern B), then the method checks whether such compound identifier does represent a concept itself in WordNet [1] (for example "bed and breakfast"). If the compound identifier does not appear in at least one synset in WordNet [1], then a pitfall is flagged.			
Limitations			
This method addresses only the English language and it should be extended in order to support other languages.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A:  		The graphical pattern shows a class whose identifier refers to the intersection of two concepts by including the copulative conjunction "and".	
Pattern B:  		The graphical pattern shows a class whose identifier refers to the union of two concepts by including the disjunctive conjunction "or".	
External resources used			
[1] Miller, G., Fellbaum, C. (1998) WordNet: An Electronic Lexical Database. Cambridge, MA: MIT Press.			

Table 5.7: Detection method proposed for P07

<table>
<tr><td>Code</td><td colspan="3">M1-P08. Detecting missing annotations in classes</td></tr>
<tr><td>Pitfall cardinality</td><td>N</td><td>Method addresses</td><td>Classes</td></tr>
<tr><td>Technique</td><td>Structural pattern matching</td><td>Pitfall affects to</td><td>Classes
Object properties
Datatype properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">This method iterates over the list of classes in the ontology.
If a given class lacks an <code>rdfs:label</code> annotation (pattern A) or an <code>rdfs:comment</code> annotation (pattern B), then a pitfall is flagged.</td></tr>
<tr><td colspan="4">Limitations</td></tr>
<tr><td colspan="4">The method does not look for other annotation properties that could be used to provide natural language identifiers and descriptions as for example <code>lemon:LexicalEntry</code>, <code>skos:prefLabel</code>, <code>skos:altLabel</code> or <code>dc:description</code>.</td></tr>
<tr><td colspan="4">Patterns</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2">Pattern A:
 

No explicit evidence of:
AnnotationAssertion(rdfs:label :ClassA “name”)</td><td colspan="2">The graphical pattern shows a class that lacks the label annotation <code>rdfs:label</code>.</td></tr>
<tr><td colspan="2">Pattern B:
 

No explicit evidence of:
AnnotationAssertion(rdfs:comment :ClassA “description”)</td><td colspan="2">The graphical pattern shows a class that lacks the description annotation <code>rdfs:comment</code>.</td></tr>
</table>

Table 5.8: Detection method 1 proposed for P08

Code	M2-P08. Detecting missing annotations in object properties		
Pitfall cardinality	N	**Method addresses**	Object properties
Technique	Structural pattern matching	**Pitfall affects to**	Classes Object properties Datatype properties
Description			
This method iterates over the list of object properties in the ontology. If a given object property lacks an `rdfs:label` annotation (pattern A) or an `rdfs:comment` annotation (pattern B), then a pitfall is flagged.			
Limitations			
The method does not look for other annotation properties that could be used to provide natural language identifiers and descriptions as for example `lemon:LexicalEntry`, `skos:prefLabel`, `skos:altLabel` or `dc:description`.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A:  ***No explicit evidence of:*** AnnotationAssertion(rdfs:label :objectPropertyS “name”)		The graphical pattern shows an object property that lacks the label annotation `rdfs:label`.	
Pattern B:  ***No explicit evidence of:*** AnnotationAssertion(rdfs:comment :objectPropertyS “description”)		The graphical pattern shows an object property that lacks the description annotation `rdfs:comment`.	

Table 5.9: Detection method 2 proposed for P08

<table>
<tr><td>Code</td><td colspan="3">M3-P08. Detecting missing annotations in datatype properties</td></tr>
<tr><td>Pitfall cardinality</td><td>N</td><td>Method addresses</td><td>Datatype properties</td></tr>
<tr><td>Technique</td><td>Structural pattern matching</td><td>Pitfall affects to</td><td>Classes
Object properties
Datatype properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">This method iterates over the list of datatype properties in the ontology.
If a given datatype property lacks an <code>rdfs:label</code> annotation (pattern A) or an <code>rdfs:comment</code> annotation (pattern B), then a pitfall is flagged.</td></tr>
<tr><td colspan="4">Limitations</td></tr>
<tr><td colspan="4">The method does not look for other annotation properties that could be used to provide natural language identifiers and descriptions as for example <code>lemon:LexicalEntry</code>, <code>skos:prefLabel</code>, <code>skos:altLabel</code> or <code>dc:description</code>.</td></tr>
<tr><td colspan="4">Patterns</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2">Pattern A:
 

No explicit evidence of:
AnnotationAssertion(rdfs:label :dataPropertyS “name”)</td><td colspan="2">The graphical pattern shows a datatype property that lacks the label annotation <code>rdfs:label</code>.</td></tr>
<tr><td colspan="2">Pattern B:
 

No explicit evidence of:
AnnotationAssertion(rdfs:comment :dataPropertyS “description”)</td><td colspan="2">The graphical pattern shows a datatype property that lacks the description annotation <code>rdfs:comment</code>.</td></tr>
</table>

Table 5.10: Detection method 3 proposed for P08

<table>
<tr><td>Code</td><td colspan="3">M1-P10. Detecting lack of class disjointness</td></tr>
<tr><td>Pitfall cardinality</td><td>1</td><td>Method addresses</td><td>Classes</td></tr>
<tr><td>Technique</td><td>Structural pattern matching</td><td>Pitfall affects to</td><td>Classes
Object properties
Datatype properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">This method iterates over the list of axioms included in the ontology and searches for the following patterns:
(a) lack of the <code>owl:disjointWith</code> primitive from OWL 1 indicating disjointness between two classes (pattern A).
(b) lack of the <code>owl:AllDisjointClasses</code> primitive from OWL 2 indicating pairwise disjointness for a set of classes (pattern B).
(c) lack of the <code>owl:disjointUnionOf</code> primitive from OWL 2 indicating disjoint union of classes (pattern C).
If all the above-mentioned patterns occur (no disjoint axiom is found), then a pitfall is flagged.</td></tr>
<tr><td colspan="4">Limitations</td></tr>
<tr><td colspan="4">If there are no statements indicating disjointness among classes in the ontology, the pitfall is triggered. However the method does not check whether two or more particular classes are disjoint as it would need background knowledge or the definitions of the ontology requirements that are not provided to the system. That is, if the ontology only contains classes that can actually share individuals, the method will produce a false positive.</td></tr>
<tr><td colspan="4">Patterns</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2">Pattern A:

No explicit evidence of:
DisjointClasses(:ClassA :ClassB)</td><td colspan="2">The graphical pattern shows the lack of disjoint axiom between two classes using the primitive <code>owl:disjointWith</code> from OWL1.1.</td></tr>
<tr><td colspan="2">Pattern B:

No explicit evidence of:
DisjointClasses(:ClassC :ClassD :ClassE)</td><td colspan="2">The graphical pattern shows the lack of disjoint axiom between a set of classes using the primitive <code>owl:AllDisjointClasses</code> from OWL2.</td></tr>
<tr><td colspan="2">Pattern C:

No explicit evidence of:
DisjointUnion(:ClassF :ClassG :ClassH :ClassI)</td><td colspan="2">The graphical pattern shows the lack of definition of a class (“Class F” in the graphical example) as the disjoint union of a group of classes that are pairwise disjoint (“ClassG”, “ClassH” and “ClassI” in the graphical example) using the primitive <code>owl:AllDisjointClasses</code> from OWL2.</td></tr>
</table>

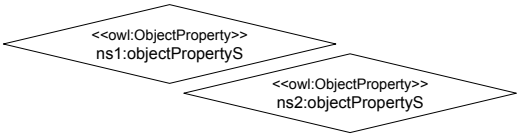
Table 5.11: Detection method proposed for P10

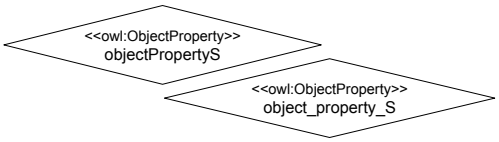
Code	M1-P11. Missing domain or range in object properties		
Pitfall cardinality	N	**Method addresses**	Object properties
Technique	Structural pattern matching	**Pitfall affects to**	Object properties Datatype properties
Description			
This method iterates over the list of object properties included in the ontology. If an object property lacks domain (pattern A) or range (pattern B) (or both), then a pitfall is flagged. Note: The current implementation of the method flags as a pitfall those object properties that inherit the domain or range from their ancestor properties, even though it is commonly not considered as such by developers.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A:  ***No explicit evidence of:*** ObjectPropertyDomain(:objectPropertys :Class)		The graphical pattern shows an object property that lacks a domain definition (rdfs:domain).	
Pattern B:  ***No explicit evidence of:*** ObjectPropertyRange(:objectPropertyS :Class)		The graphical pattern shows an object property that lacks a range definition (rdfs:range).	

Table 5.12: Detection method 1 proposed for P11

Code	M2-P11. Missing domain or range in datatype properties		
Pitfall cardinality	N	**Method addresses**	Datatype properties
Technique	Structural pattern matching	**Pitfall affects to**	Object properties Datatype properties
Description			
This method iterates over the list of datatype properties included in the ontology. If a datatype property lacks domain (pattern A) or range (pattern B) (or both), then a pitfall is flagged. Note: The current implementation of the method flags as a pitfall those object properties that inherit the domain or range from their ancestor properties, even though it is commonly not considered as such by developers.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A:  ***No explicit evidence of:*** DataPropertyDomain(:datatypePropertyS :Class)		The graphical pattern shows a datatype property that lacks a domain definition (`rdfs:domain`).	
Pattern B:  ***No explicit evidence of:*** DataPropertyRange(:datatypePropertyS xsd:datatype1)		The graphical pattern shows a datatype property that lacks a range definition (`rdfs:range`).	

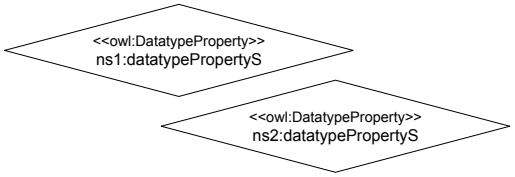
Table 5.13: Detection method 2 proposed for P11

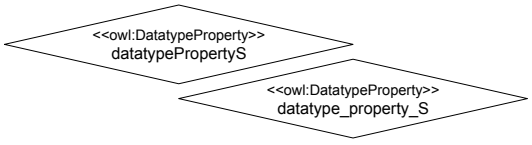
<table>
<tr><td>Code</td><td colspan="3">M1-P12: Detecting equivalent object properties not explicitly declared</td></tr>
<tr><td>Pitfall cardinality</td><td>N</td><td>Method addresses</td><td>Object properties</td></tr>
<tr><td>Technique</td><td>Structural pattern matching
Linguistic analysis</td><td>Pitfall affects to</td><td>Object properties
Datatype properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">This method iterates over the list of object properties included in the ontology.
For each pair of object properties, the method checks the following conditions:
(a) Both object properties have the same identifier (in different namespaces) and they are not equivalent (<code>owl:equivalentProperty</code>) or sub properties (<code>rdfs:subPropertyOf</code>) of each other (pattern A).
(b) The identifiers of both object properties contain the same lexical description but they are represented following different naming conventions and the object properties are not equivalent (<code>owl:equivalentProperty</code>) or sub properties (<code>rdfs:subPropertyOf</code>) of each other (pattern B).
The pitfall is flagged for each and every occurrence of the above-mentioned patterns.</td></tr>
<tr><td colspan="4">Limitations</td></tr>
<tr><td colspan="4">The current implementation of the method does not look for synonyms between verbs used for labelling properties or equivalent meaning of relationships. That is, if two relationships “isBornInPlace” and “hasBirthPlace” are defined in the ontology, the method will not infer that they are equivalent object properties.</td></tr>
<tr><td colspan="4">Patterns</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2">Pattern A:
 

No explicit evidence of:
EquivalentObjectProperties(ns1:objectPropertyS ns2:objectPropertyS)
SubObjectPropertyOf (ns1:objectPropertyS ns2:objectPropertyS)
SubObjectPropertyOf (ns2:objectPropertyS ns1:objectPropertyS)</td><td colspan="2">The graphical pattern shows a pair of object properties defined in different namespaces and sharing the same identifiers. Such object properties are not defined as equivalent and none of them is a sub property of the other.</td></tr>
<tr><td colspan="2">Pattern B:
 

No explicit evidence of:
EquivalentObjectProperties(:objectPropertyS :object_property_S)
SubObjectPropertyOf (:objectPropertyS :object_property_S)
SubObjectPropertyOf (:object_property_S :objectPropertyS)</td><td colspan="2">The graphical pattern shows a pair of object properties whose identifiers contains the same terms but are expressed with different naming conventions. Such object properties are not defined as equivalent and none of them is a sub property of the other.</td></tr>
</table>

Table 5.14: Detection method 1 proposed for P12

<table>
<tr><td>Code</td><td colspan="3">M2-P12. Detecting equivalent datatype properties not explicitly declared</td></tr>
<tr><td>Pitfall cardinality</td><td>N</td><td>Method addresses</td><td>Datatype properties</td></tr>
<tr><td>Technique</td><td>Structural pattern matching
Linguistic analysis</td><td>Pitfall affects to</td><td>Object properties
Datatype properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">This method iterates over the list of datatype properties included in the ontology.
For each pair of datatype properties, the method checks the following conditions:
(a) Both datatype properties have the same identifier (in different namespaces) and they are not equivalent (<code>owl:equivalentProperty</code>) or sub properties (<code>rdfs:subPropertyOf</code>) of each other (pattern A).
(b) The identifiers of both datatype properties contain the same lexical description but they are represented following different naming conventions and the object properties are not equivalent (<code>owl:equivalentProperty</code>) or sub properties (<code>rdfs:subPropertyOf</code>) of each other (pattern B).
The pitfall is flagged for each and every occurrence of the above-mentioned patterns.</td></tr>
<tr><td colspan="4">Limitations</td></tr>
<tr><td colspan="4">The current implementation of the method does not look for synonyms between verbs used for labelling properties nor equivalent meaning of attributes. That is, if two attributes “birthday” and “day of birth” are defined in the ontology, the method will not infer that they are equivalent datatype properties.</td></tr>
<tr><td colspan="4">Patterns</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2">Pattern A:
 

No explicit evidence of:
EquivalentDataProperties (ns1:dataPropertyS ns2:dataPropertyS)
SubDataPropertyOf (ns1:dataPropertyS ns2:dataPropertyS)
SubDataPropertyOf (ns2:dataPropertyS ns1:dataPropertyS)</td><td colspan="2">The graphical pattern shows a pair of datatype properties defined in different namespaces and sharing the same identifiers. Such datatype properties are not defined as equivalent and none of them is a sub property of the other.</td></tr>
<tr><td colspan="2">Pattern B:
 

No explicit evidence of:
EquivalentDataProperties (:dataPropertyS :data_property_S)
SubDataPropertyOf (:dataPropertyS :data_property_S)
SubDataPropertyOf (:data_property_S :dataPropertyS)</td><td colspan="2">The graphical pattern shows a pair of datatype properties whose identifiers contains the same terms but are expressed with different naming conventions. Such datatype properties are not defined as equivalent and none of them is a sub property of the other.</td></tr>
</table>

Table 5.15: Detection method 2 proposed for P12

Code	M1-P13. Detecting inverse relationships not explicitly declared		
Pitfall cardinality	N	**Method addresses**	Object properties
Technique	Structural pattern matching	**Pitfall affects to**	Object properties

Description

This method iterates over the list of object properties included in the ontology.
If an object property is not defined as symmetric property (`owl:SymmetricProperty`) and does not have any inverse property defined (`owl:inverseOf`), then a pitfall is flagged (pattern A).

In addition, if two object properties satisfy the above condition (pattern A) and the domain of each of them matches the range of the other property, and none of the object properties has an inverse property defined (pattern B), then the two object properties are suggested as potential inverse properties.

Limitations

The method identifies as pitfall every occurrence of object property without inverse property.
In order to provide suggestions of possible pairs of inverse properties, the method looks for pairs of properties where (a) none of the properties has an inverse property defined and (b) the domain defined for each property is the same class as the range of the other property. This suggestion method, based on structural analysis, has a limitation as the lexical form of the properties is not analysed. Therefore, if there are more than two properties defined among two classes that fulfil the conditions mentioned, the proposed pairs are set according to the appearing order instead of the meaning of the properties.

Patterns

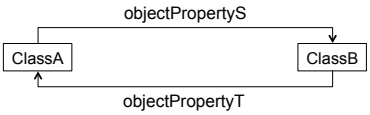
Graphical representation and/or OWL code	Natural language description
Pattern A:  ***No explicit evidence of:*** SymmetricObjectProperty(:objectPropertyS) InverseObjectProperties(:objectPropertyS :objectPropertyX)	The graphical pattern shows an object property that has no inverse property defined and that itself is not defined as symmetric.
Pattern B (for suggesting inverse relationships):  ***No explicit evidence of:*** SymmetricObjectProperty(:objectPropertyT) SymmetricObjectProperty(:objectPropertyS) InverseObjectProperties(:objectPropertyS :objectPropertyX) InverseObjectProperties(:objectPropertyX :objectPropertyS) InverseObjectProperties(:objectPropertyT :objectPropertyY) InverseObjectProperties(:objectPropertyY :objectPropertyT)	The graphical pattern shows two object properties without inverse property defined and they are not symmetric properties. In addition, the range of each property and the domain of the other property is the same class. In this case both properties are suggested to be inverse of each other based on structural criteria.

Table 5.16: Detection method proposed for P13

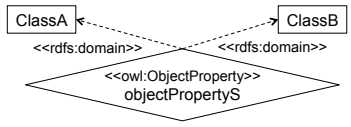
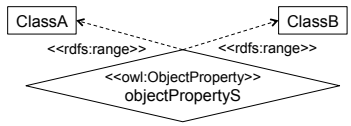
Code	M1-P19. Detecting multiple domains or ranges in object properties		
Pitfall cardinality	N	**Method addresses**	Object properties
Technique	Structural pattern matching	**Pitfall affects to**	Object properties Datatype properties
Description			
This method iterates over the list of object properties included in the ontology. If an object property has more than one `rdfs:domain` axiom (pattern A) and the classes the domain axioms refer to do not match, then a pitfall is flagged. If an object property has more than one `rdfs:range` axiom (pattern B) and the classes the range axioms refer to do not match, then a pitfall is flagged. The method checks if the multiple classes declared as domain or range are the same class or equivalent classes. If class restrictions apply, the method checks whether they are syntactically equivalent.			
Limitations			
The method does not check if two different class restrictions or anonymous classes logically represent identical sets of individuals.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A:  		The graphical pattern shows an object property that has two `rdfs:domain` axioms defined.	
Pattern B:  		The graphical pattern shows an object property that has two `rdfs:range` axioms defined.	

Table 5.17: Detection method 1 proposed for P19

<table>
<tr><td>Code</td><td colspan="3">M2-P19. Detecting multiple domains or ranges in datatype properties</td></tr>
<tr><td>Pitfall cardinality</td><td>N</td><td>Method addresses</td><td>Datatype properties</td></tr>
<tr><td>Technique</td><td>Structural pattern matching</td><td>Pitfall affects to</td><td>Object properties
Datatype properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">This method iterates over the list of datatype properties included in the ontology.
If a datatype property has more than one <code>rdfs:domain</code> axiom (pattern A) and the classes the domain axioms refer to do not match, then a pitfall is flagged.
If a datatype property has more than one <code>rdfs:range</code> axiom (pattern B) and the datatypes the range axioms refer to are not the same one, then a pitfall is flagged.

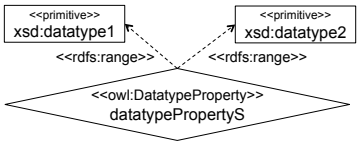
The method checks if the multiple classes declared as domain are the same class or equivalent classes. If class restrictions apply, the method checks whether they are syntactically equivalent.</td></tr>
<tr><td colspan="4">Limitations</td></tr>
<tr><td colspan="4">The method does not check if two different class restrictions or anonymous classes logically represent identical sets of individuals.</td></tr>
<tr><td colspan="4">Patterns</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2">Pattern A:
 
</td><td colspan="2">The graphical pattern shows a datatype property that has two <code>rdfs:domain</code> axioms defined.</td></tr>
<tr><td colspan="2">Pattern B:
 
</td><td colspan="2">The graphical pattern shows a datatype property that has two <code>rdfs:range</code> axioms defined.</td></tr>
</table>

Table 5.18: Detection method 2 proposed for P19

Code	M1-P20. Detecting misuses of ontology annotations in classes		
Pitfall cardinality	N	**Method addresses**	Classes
Technique	Structural pattern matching Linguistic analysis	**Pitfall affects to**	Classes Object properties Datatype properties

Description

This method iterates over the list of classes included in the ontology.
If for a given class:
its `rdfs:label` annotation includes more tokens than its `rdfs:comment` (pattern A) or
its `rdfs:label` annotation is empty (pattern B) or
its `rdfs:comment` annotation is empty (pattern C) or
its `rdfs:label` annotation and its `rdfs:comment` have the same content (pattern D)
then, a pitfall is flagged.

Limitations

The method does not detect other annotations that also provide natural language descriptions and identifiers as, for example, `lemon:LexicalEntry`, `skos:prefLabel`, `skos:altLabel` or `dc:description`.

Patterns

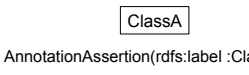
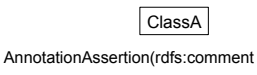
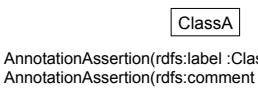
Graphical representation and/or OWL code	Natural language description
Pattern A: 	The graphical pattern shows a class whose `rdfs:label` annotation includes a definition of the concept and whose `rdfs:comment` annotation contains the name given to the concept.
Pattern B: 	The graphical pattern shows a class whose `rdfs:label` is empty.
Pattern C: 	The graphical pattern shows a class whose `rdfs:comment` is empty.
Pattern D: 	The graphical pattern shows a class whose `rdfs:label` and `rdfs:comment` have the same content.

Table 5.19: Detection method 1 proposed for P20

<table>
<tr><td>Code</td><td colspan="3">M2-P20. Detecting misuses of ontology annotations in object properties</td></tr>
<tr><td>Pitfall cardinality</td><td>N</td><td>Method addresses</td><td>Object properties</td></tr>
<tr><td>Technique</td><td>Structural pattern matching
Linguistic analysis</td><td>Pitfall affects to</td><td>Classes
Object properties
Datatype properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">This method iterates over the list of object properties included in the ontology.
If for a given object property:
its <code>rdfs:label</code> annotation includes more tokens than its <code>rdfs:comment</code> (pattern A) or
its <code>rdfs:label</code> annotation is empty (pattern B) or
its <code>rdfs:comment</code> annotation is empty (pattern C) or
its <code>rdfs:label</code> annotation and its <code>rdfs:comment</code> have the same content (pattern D)
then, a pitfall is flagged.</td></tr>
<tr><td colspan="4">Limitations</td></tr>
<tr><td colspan="4">The method does not detect other annotations that also provide natural language descriptions and identifiers as, for example, <code>lemon:LexicalEntry</code>, <code>skos:prefLabel</code>, <code>skos:altLabel</code> or <code>dc:description</code>.</td></tr>
<tr><td colspan="4">Patterns</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2">Pattern A:
 

AnnotationAssertion(
rdfs:label :objectPropertyS “a definition in natural language”)
AnnotationAssertion(rdfs:comment :objectPropertyS “name”)</td><td colspan="2">The graphical pattern shows an object property whose <code>rdfs:label</code> annotation includes a definition of the relationship and whose <code>rdfs:comment</code> annotation contains the name given to the relationship.</td></tr>
<tr><td colspan="2">Pattern B:
 

AnnotationAssertion(rdfs:label :objectPropertyS “”)</td><td colspan="2">The graphical pattern shows an object property whose <code>rdfs:label</code> is empty.</td></tr>
<tr><td colspan="2">Pattern C:
 

AnnotationAssertion(rdfs:comment :objectPropertyS “”)</td><td colspan="2">The graphical pattern shows an object property whose <code>rdfs:comment</code> is empty.</td></tr>
<tr><td colspan="2">Pattern D:
 

AnnotationAssertion(rdfs:label :objectPropertyS “content”)
AnnotationAssertion(rdfs:comment :objectPropertyS “content”)</td><td colspan="2">The graphical pattern shows an object property whose <code>rdfs:label</code> and <code>rdfs:comment</code> have the same content.</td></tr>
</table>

Table 5.20: Detection method 2 proposed for P20

<table>
<tr><td>Code</td><td colspan="3">M3-P20. Detecting misuses of ontology annotations in datatype properties</td></tr>
<tr><td>Pitfall cardinality</td><td>N</td><td>Method addresses</td><td>Datatype properties</td></tr>
<tr><td>Technique</td><td>Structural pattern matching
Linguistic analysis</td><td>Pitfall affects to</td><td>Classes
Object properties
Datatype properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">This method iterates over the list of datatype properties included in the ontology.
If for a given datatype property:
its <code>rdfs:label</code> annotation includes more tokens than its <code>rdfs:comment</code> (pattern A) or
its <code>rdfs:label</code> annotation is empty (pattern B) or
its <code>rdfs:comment</code> annotation is empty (pattern C) or
its <code>rdfs:label</code> annotation and its <code>rdfs:comment</code> have the same content (pattern D)
then, a pitfall is flagged.</td></tr>
<tr><td colspan="4">Limitations</td></tr>
<tr><td colspan="4">The method does not detect other annotations that also provide natural language descriptions and identifiers as, for example, <code>lemon:LexicalEntry</code>, <code>skos:prefLabel</code>, <code>skos:altLabel</code> or <code>dc:description</code>.</td></tr>
<tr><td colspan="4">Patterns</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2">Pattern A:
 

AnnotationAssertion(
rdfs:label :dataPropertyR “a definition in natural language”)
AnnotationAssertion(rdfs:comment :dataPropertyR “name”)</td><td colspan="2">The graphical pattern shows a datatype property whose <code>rdfs:label</code> annotation includes a definition of the attribute and whose <code>rdfs:comment</code> annotation contains the name given to the attribute.</td></tr>
<tr><td colspan="2">Pattern B:
 

AnnotationAssertion(rdfs:label :dataPropertyR “”)</td><td colspan="2">The graphical pattern shows a datatype property whose <code>rdfs:label</code> is empty.</td></tr>
<tr><td colspan="2">Pattern C:
 

AnnotationAssertion(rdfs:comment :dataPropertyR “”)</td><td colspan="2">The graphical pattern shows a datatype property whose <code>rdfs:comment</code> is empty.</td></tr>
<tr><td colspan="2">Pattern D:
 

AnnotationAssertion(
rdfs:label :dataPropertyR “content”)
AnnotationAssertion(rdfs:comment :dataPropertyR “content”)</td><td colspan="2">The graphical pattern shows a datatype property whose <code>rdfs:label</code> and <code>rdfs:comment</code> have the same content.</td></tr>
</table>

Table 5.21: Detection method 3 proposed for P20

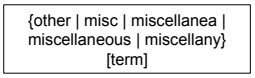
Code	M1-P21. Detecting miscellaneous classes		
Pitfall cardinality	N	**Method addresses**	Classes
Technique	Linguistic analysis	**Pitfall affects to**	Classes
Description			
This method iterates over the list of classes included in the ontology. If a given class identifier contains any of the lemmas “other”, “misc”, “miscellanea”, “miscellaneous” or “miscellany” (pattern A), then a pitfall is flagged.			
Limitations			
This method addresses only the English language. Another limitation is the false positive that might arise if the ontology under analysis describes a domain in which lemmas that match the pattern A are needed (as for example in the bibliographical domain).			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A: 		The graphical pattern shows a class whose identifier contains one of the lemmas indicating a variety of things (see the description field) and optionally other terms.	

Table 5.22: Detection method proposed for P21

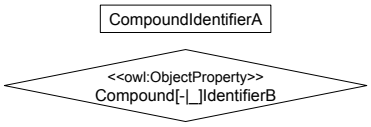
Code	M1-P22. Detecting different naming conventions across ontology elements		
Pitfall cardinality	1	**Method addresses**	Ontology
Technique	Linguistic analysis	**Pitfall affects to**	Ontology
Description			
This method iterates over the list of ontology elements (classes, object properties and datatype properties) included in the ontology. If two elements (regardless their ontology element type) identified by compound terms follow different naming conventions as indicated in pattern A (for example the CamelCase convention or conventions using delimiters as “-” or “_”), then a pitfall is flagged.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A: 		The graphical pattern shows two ontology elements, a class and an object property, whose identifiers follow different naming convention. Even though the method also compares lowercase identifiers between classes, object properties and datatype properties, for the sake of clarity only the combination of one class and one object property is represented. We omit the rest of combinations between the three types of elements and the different notations.	

Table 5.23: Detection method 1 proposed for P22

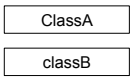
Code	M2-P22. Detecting different naming conventions across classes		
Pitfall cardinality	1	**Method addresses**	Classes
Technique	Linguistic analysis	**Pitfall affects to**	Ontology
Description			
This method iterates over the list of classes included in the ontology. If a class starts with lowercase and a different class starts with uppercase (pattern A), then a pitfall is flagged.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A: 		The graphical pattern shows a class whose identifier starts with lowercase and a class whose identifier starts with uppercase.	

Table 5.24: Detection method 2 proposed for P22

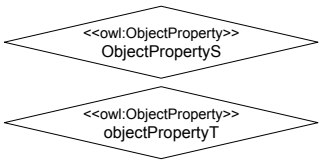
Code	M3-P22. Detecting different naming conventions across object properties		
Pitfall cardinality	1	**Method addresses**	Object properties
Technique	Linguistic analysis	**Pitfall affects to**	Ontology
Description			
This method iterates over the list of object properties included in the ontology. If an object property starts with lowercase and a different object property starts with uppercase (pattern A), then a pitfall is flagged.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A: 		The graphical pattern shows an object property whose identifier starts with lowercase and an object property whose identifier starts with uppercase.	

Table 5.25: Detection method 3 proposed for P22

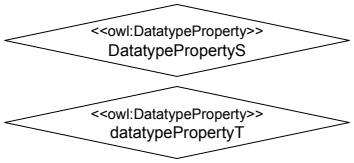
Code	M4-P22. Detecting different naming conventions across datatype properties		
Pitfall cardinality	1	**Method addresses**	Datatype properties
Technique	Linguistic analysis	**Pitfall affects to**	Ontology
Description			
This method iterates over the list of datatype properties included in the ontology. If a datatype property starts with lowercase and a different datatype property starts with uppercase (pattern A), then a pitfall is flagged.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A:  		The graphical pattern shows a datatype property whose identifier starts with uppercase and a datatype property whose identifier starts with uppercase.	

Table 5.26: Detection method 4 proposed for P22

Code	M1-P24. Detecting recursive definitions in classes		
Pitfall cardinality	N	**Method addresses**	Classes
Technique	Structural pattern matching	**Pitfall affects to**	Classes Object properties Datatype properties
Description			
This method iterates over the list of ontology classes included in the ontology. If a given class appears in its own axiom definitions (`owl:equivalentClass` or `rdfs:subClassOf`), then a pitfall is flagged.			
Patterns			
No graphical pattern is provided for this method due to the recursive aspect of the method and the high number of combinations.			

Table 5.27: Detection method 1 proposed for P24

Code	M2-P24. Detecting recursive definitions in object properties		
Pitfall cardinality	N	**Method addresses**	Object properties
Technique	Structural pattern matching	**Pitfall affects to**	Classes Object properties Datatype properties
Description			
This method iterates over the list of object properties included in the ontology. If a given object property appears in its own domain (`rdfs:domain`) or range (`rdfs:range`) definitions, then a pitfall is flagged.			
Patterns			
No graphical pattern is provided for this method due to the recursive aspect of the method and the high number of combinations.			

Table 5.28: Detection method 2 proposed for P24

Code	M3-P24. Detecting recursive definitions in datatype properties		
Pitfall cardinality	N	**Method addresses**	Datatype properties
Technique	Structural pattern matching	**Pitfall affects to**	Classes Object properties Datatype properties
Description			
This method iterates over the list of datatype properties included in the ontology. If a given datatype property appears in its own domain (`rdfs:domain`) definition, then a pitfall is flagged.			
Patterns			
No graphical pattern is provided for this method due to the recursive aspect of the method and the high number of combinations.			

Table 5.29: Detection method 3 proposed for P24

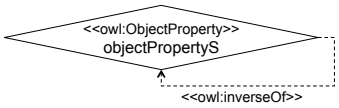
Code	M1-P25. Detecting relationships inverse to themselves		
Pitfall cardinality	N	**Method addresses**	Object properties
Technique	Structural pattern matching	**Pitfall affects to**	Object properties
Description			
This method iterates over the list of object properties included in the ontology. If a given object property acts at the same time as subject and object in an `owl:inverseOf` statement (pattern A), then a pitfall is flagged.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A:  		The graphical pattern shows an object property that has an `owl:inverseOf` axiom whose target object property is the same object property.	

Table 5.30: Detection method proposed for P25

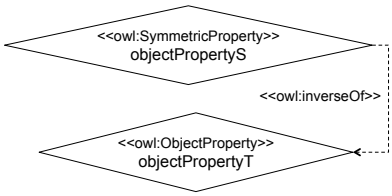
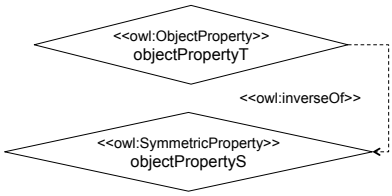
<table>
<tr><td>Code</td><td colspan="3">M1-P26. Detecting inverse relationships for symmetric ones</td></tr>
<tr><td>Pitfall cardinality</td><td>N</td><td>Method addresses</td><td>Object properties</td></tr>
<tr><td>Technique</td><td>Structural pattern matching</td><td>Pitfall affects to</td><td>Object properties</td></tr>
<tr><td colspan="4">Description</td></tr>
<tr><td colspan="4">This method iterates over the list of object properties defined as symmetric (<code>owl:SymmetricProperty</code>) in the ontology.
If a given object property acts as subject (pattern A) or as object (pattern B) of an inverse property statement (<code>owl:inverseOf</code>), then a pitfall is flagged.</td></tr>
<tr><td colspan="4">Patterns</td></tr>
<tr><td colspan="2">Graphical representation and/or OWL code</td><td colspan="2">Natural language description</td></tr>
<tr><td colspan="2">Pattern A:
 
</td><td colspan="2">The graphical pattern shows a symmetric object property that has an <code>owl:inverseOf</code> axiom.</td></tr>
<tr><td colspan="2">Pattern B:
 
</td><td colspan="2">The graphical pattern shows an object property that has an <code>owl:inverseOf</code> axiom whose target object property is a symmetric one.</td></tr>
</table>

Table 5.31: Detection method proposed for P26

Code	M1-P27. Detecting wrong equivalent object properties		
Pitfall cardinality	N	**Method addresses**	Object properties
Technique	Structural pattern matching	**Pitfall affects to**	Object properties Datatype properties

Description

This method iterates over the list of object properties in the ontology.
For each pair of equivalent object properties (`owl:equivalentProperty`), the method checks whether:

(a) the domains of both object properties are defined but they do not match each other (pattern A)
(b) the ranges of both object properties are defined but they do not match each other (pattern B)

If for a pair of equivalent object properties at least one of the above-mentioned conditions is met, then a pitfall is flagged.

The method checks if the domains compared or the ranges compared are the same class or equivalent classes. If class restrictions apply, the method checks whether they are syntactically equivalent.

Limitations

The method does not check if two different class restrictions or anonymous classes logically represent identical sets of individuals.

Patterns

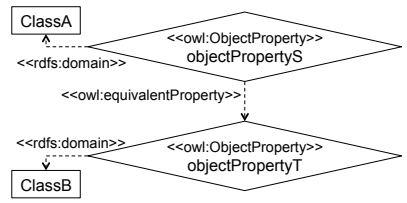
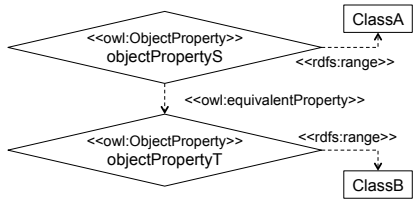
Graphical representation and/or OWL code	Natural language description
Pattern A:  	The graphical pattern shows two equivalent object properties with different domains.
Pattern B:  	The graphical pattern shows two equivalent object properties with different ranges.

Table 5.32: Detection method 1 proposed for P27

Code	M2-P27. Detecting wrong equivalent datatype properties		
Pitfall cardinality	N	**Method addresses**	Datatype properties
Technique	Structural pattern matching	**Pitfall affects to**	Object properties Datatype properties

Description

This method iterates over the list of datatype properties in the ontology.
For each pair of equivalent datatype properties (`owl:equivalentProperty`), the method checks whether:

(a) the domains of both datatype properties are defined but they do not match each other (pattern A)

(b) the ranges of both datatype properties are defined but they do not match each other (pattern B)

If for a pair of equivalent object properties at least one of the above-mentioned conditions is met, then a pitfall is flagged.

The method checks if the domains compared are the same class or equivalent classes. If class restrictions apply, the method checks whether they are syntactically equivalent.For ranges, the method checks whether they are the same datatype.

Limitations

The method does not check if two different class restrictions or anonymous classes logically represent identical sets of individuals.

Patterns

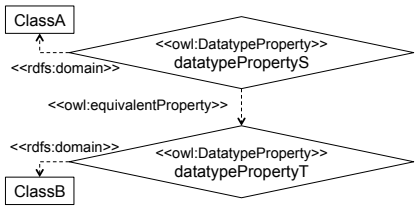
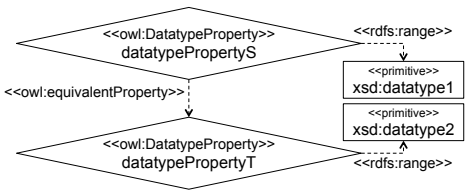
Graphical representation and/or OWL code	Natural language description
Pattern A:  	The graphical pattern shows two equivalent datatype properties with different domains.
Pattern B:  	The graphical pattern shows two equivalent datatype properties with different ranges.

Table 5.33: Detection method 2 proposed for P27

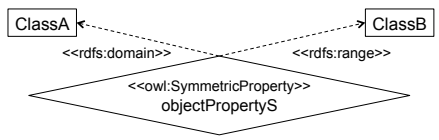
Code	M1-P28. Detecting wrong symmetric relationships		
Pitfall cardinality	N	**Method addresses**	Object properties
Technique	Structural pattern matching	**Pitfall affects to**	Object properties
Description			
This method iterates over the list of object properties defined as symmetric (`owl:SymmetricProperty`) in the ontology. For a given symmetric object property, if its domain and range are defined and they do not match (pattern A), then a pitfall is flagged. The method checks if the domain and the range are the same class or equivalent classes. If class restrictions apply, the method checks whether they are syntactically equivalent.			
Limitations			
The method does not check if two different class restrictions or anonymous classes logically represent identical sets of individuals.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A:  		The graphical pattern shows a symmetric object property whose domain is different from its range.	

Table 5.34: Detection method proposed for P28

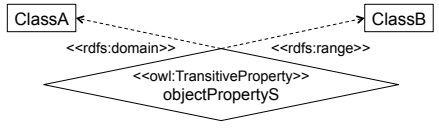
Code	M1-P29. Defining wrong transitive relationships		
Pitfall cardinality	N	**Method addresses**	Object properties
Technique	Structural pattern matching	**Pitfall affects to**	Object properties
Description			
This method iterates over the list of object properties defined as transitive (`owl:TransitiveProperty`) in the ontology. For a given transitive object property, if its domain and range are defined and they do not match (pattern A), then a pitfall is flagged. The method checks if the domain and the range are the same class or equivalent classes. If class restrictions apply, the method checks whether they are syntactically equivalent.			
Limitations			
The method does not check if two different class restrictions or anonymous classes logically represent identical sets of individuals.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A:  		The graphical pattern shows a transitive object property whose domain is different from its range.	

Table 5.35: Detection method proposed for P29

Code	M1-P30. Detecting equivalent classes not explicitly declared		
Pitfall cardinality	N	**Method addresses**	Classes
Technique	Structural pattern matching Linguistic analysis	**Pitfall affects to**	Classes

Description

This method iterates over the list of classes included in the ontology to search for classes that are not defined as equivalent (`owl:equivalentClass`) but they could be.
For each pair of classes that are not defined as equivalent (`owl:equivalentClass`), their names (the content of an `rdfs:label` in English or their identifiers in case no label in English is provided) are compared:

(a) In case both names are single terms:
  (a.1) If the names of both classes appear in a common synset in WordNet [1], then a pitfall is flagged.

(b) In case at least one name is a compound term:
  (b.1) Remove stop words.
  (b.2) Split the compound terms in simple terms.
  (b.3) Generate all possible combinations of pairs taking one simple term from each class name.
  (b.4) For each combination, check whether both simple terms appear in a common synset in WordNet
  (b.5) If for all combinations of simple terms, both terms appear in a common synset, then a pitfall is flagged.

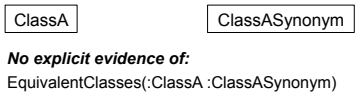
Limitations

Even though the method considers compound terms, it currently does not take into account cases in which the class names differ on the number of single terms they are made up of, for example "Birthday" and "Natal day".

This method addresses only the English language.

The method only detects annotations in `rdfs:label`. Other annotation properties for naming, although identified (for example `skos:prefLabel`), are not yet included.

Patterns

Graphical representation and/or OWL code	Natural language description
Pattern A:  	The graphical pattern shows two classes whose identifiers represent synonymous concepts and that are not defined as equivalent classes by means of the primitive `owl:equivalentClass`. To verify that the identifiers do represent synonymous concepts in at least one sense, the method checks whether they belong to the same synset in WordNet [1].

External resources used

[1] Miller, G., Fellbaum, C. (1998) WordNet: An Electronic Lexical Database. Cambridge, MA: MIT Press.

Table 5.36: Detection method proposed for P30

Code	M1-P31. Defining wrong equivalent classes		
Pitfall cardinality	N	**Method addresses**	Classes
Technique	Structural pattern matching Linguistic analysis	**Pitfall affects to**	Classes

Description

This method iterates over the list of classes included in the ontology.
For each pair of classes that are defined as equivalent (`owl:equivalentClass`), their names (the content of an `rdfs:label` in English or their identifiers in case no label in English is provided) are compared:

(a) In case both names are single terms:
  (a.1) If the names of both classes do not appear in a common synset in WordNet [1], then a pitfall is flagged.

(b) In case at least one name is a compound term:
  (b.1) Remove stop words.
  (b.2) Split the compound terms in simple terms.
  (b.3) Generate all possible combinations of pairs taking one simple term from each class name.
  (b.4) For each combination, check whether both simple terms appear in a common synset in WordNet
  (b.5) If for at least one combination of simple terms, the terms do not appear in a common synset, then a pitfall is flagged.

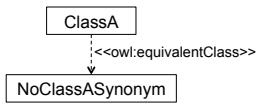
Limitations

Even though the method considers compound terms, it currently does not take into account cases in which the class names differ on the number of single terms they are made up of, for example "Birthday" and "Natal day".

This method addresses only the English language.

The method only detects annotations in `rdfs:label`. Other annotation properties for naming, although identified (for example `skos:prefLabel`), are not yet included.

Patterns

Graphical representation and/or OWL code	Natural language description
Pattern A:  	The graphical pattern shows two classes defined as equivalent classes by means of the primitive `owl:equivalentClass` and whose identifiers do not represent synonymous concepts. In order to check that the identifiers do not represent synonymous concepts the method checks whether they are part of the same synset in WordNet [1].

External resources used

[1] Miller, G., Fellbaum, C. (1998) WordNet: An Electronic Lexical Database. Cambridge, MA: MIT Press.

Table 5.37: Detection method proposed for P31

Code	M1-P32. Several classes with the same label		
Pitfall cardinality	N	**Method addresses**	Classes
Technique	Structural pattern matching Linguistic analysis	**Pitfall affects to**	Classes
Description			
This method iterates over the list of classes included in the ontology. For each pair of classes whose `rdfs:label` in English have the same content and that are not defined as equivalent classes (`owl:equivalentClass`) (pattern A), then a pitfall is flagged.			
Limitations			
The method only detects annotations in `rdfs:label`. Other annotation properties for naming, although identified (for example `skos:prefLabel`), are not yet included.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A: ClassA ClassB AnnotationAssertion(rdfs:label :ClassA "content") AnnotationAssertion(rdfs:label :ClassB "content") ***No explicit evidence of:*** EquivalentClasses(:ClassA :ClassB)		The graphical pattern shows two classes that are not defined as equivalent classes by means of the primitive `owl:equivalentClass` and that share the content of the `rdfs:label` naming annotation.	

Table 5.38: Detection method proposed for P32

Code	M1-P33. Creating a property chain with just one property		
Pitfall cardinality	N	**Method addresses**	Object properties
Technique	Structural pattern matching	**Pitfall affects to**	Object properties
Description			
This method iterates over the list of property chain axioms declared in the ontology. If a given property chain includes only one object property in its antecedent (Pattern A), then a pitfall is flagged.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A: <<owl:ObjectProperty>> objectPropertyS <<owl:ObjectProperty>> objectPropertyT SubObjectPropertyOf(ObjectPropertyChain(:objectPropertyS) :objectPropetyT)		The graphical pattern shows two object properties and a property chain (functional syntax) defined, in which only one object property, namely "objectPropertyS", appears in the antecedent part of the chain.	

Table 5.39: Detection method proposed for P33

Code	M1-P34. Untyped class		
Pitfall cardinality	N	**Method addresses**	Classes
Technique	Structural pattern matching	**Pitfall affects to**	Classes

Description

This method looks for ontology elements that are used as classes but have not been defined as such using the primitives `owl:Class` or `rdfs:Class`.
The method checks if an ontology element is not defined as a class and:

- it appears as domain (pattern A) or range (pattern B) in an object property.
- it appears as domain (pattern C) in a datatype property.
- it appears as subject (pattern D) or object (pattern E) in an `rdfs:subClassOf` statement.
- it appears as subject (pattern F) or object (pattern G) in an `owl:disjointWith` statement.
- it appears as subject (pattern H) or object (pattern I) in an `owl:equivalentClass` statement.

If at least one of the above-mentioned conditions is met, then a pitfall is flagged.

Limitations

This method does not check the ontology elements appearing in class description axioms as: (a) value constrains (for example in `owl:allValuesFrom` or `owl:someValuesFrom` constructs); (b) cardinality constraints (for example in `owl:maxCardinality` or `owl:minCardinality` constructs); (c) intersections (`owl:intersectionOf`); (d) unions (`owl:unionOf`); or (e) complements (`owl:complementOf`).

Patterns (Part A)

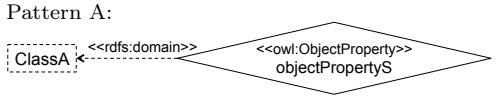
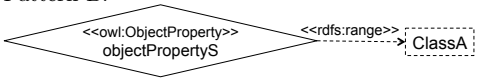
Graphical representation and/or OWL code	Natural language description
Along the graphical representations a dashed box is used to represent the ontology element that is not defined as class in the patterns. This alternative notation is used for the sake of clarity using graphical patterns instead of functional syntax. A regular box is not used as it would be confused with elements actually defined as classes in the ontology.	
Pattern A:  ***No explicit evidence of:*** Declaration(Class(:ClassA))	The graphical pattern shows an ontology element (ClassA) defined as domain of an object property. The ontology lacks the definition of such element as a class.
Pattern B:  ***No explicit evidence of:*** Declaration(Class(:ClassA))	The graphical pattern shows an ontology element (ClassA) defined as range of an object property. The ontology lacks the definition of such element as a class.
Pattern C:  ***No explicit evidence of:*** Declaration(Class(:ClassA))	The graphical pattern shows an ontology element (ClassA) defined as domain of a datatype property. The ontology lacks the definition of such element as a class.

Table 5.40: Detection method proposed for P34 (Part A)

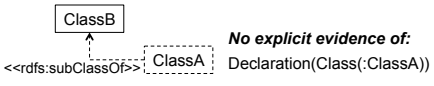
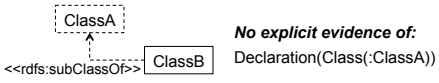
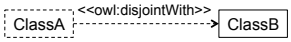
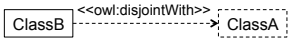
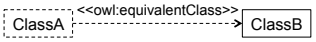
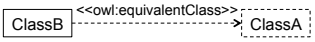
Code	M1-P34. Untyped class
Patterns (Part B)	
Graphical representation and/or OWL code	Natural language description
Pattern D:  	The graphical pattern shows an ontology element (ClassA) defined as subclass of a class (ClassB). The ontology lacks the definition of such element as a class.
Pattern E:  	The graphical pattern shows a class (ClassB) defined as subclass of an ontology element (ClassA). The ontology lacks the definition of such element as a class.
Pattern F:  **No explicit evidence of:** Declaration(Class(:ClassA))	The graphical pattern shows an ontology element (ClassA), which is defined as disjoint with a class (ClassB). The ontology lacks the definition of such element as a class.
Pattern G:  **No explicit evidence of:** Declaration(Class(:ClassA))	The graphical pattern shows a class (ClassB) defined as disjoint with an ontology element (ClassA). The ontology lacks the definition of such ontology element as a class.
Pattern H:  **No explicit evidence of:** Declaration(Class(:ClassA))	The graphical pattern shows an ontology element (ClassA), which is defined as equivalent to a class (ClassB). The ontology lacks the definition of such element as a class.
Pattern I:  **No explicit evidence of:** Declaration(Class(:ClassA))	The graphical pattern shows a class (ClassB) defined as equivalent to an ontology element (ClassA) . The ontology lacks the definition of such ontology element as a class.

Table 5.41: Detection method proposed for P34 (Part B)

Code	M1-P35. Untyped property		
Pitfall cardinality	N	**Method addresses**	Object properties Datatype properties
Technique	Structural pattern matching	**Pitfall affects to**	Object properties Datatype properties

Description

This method looks for ontology elements that are used as object properties or datatype properties but have not been defined as such using the primitives `owl:ObjectProperty` or `owl:DatatypeProperty`.
The method checks if the ontology element:

- is not defined as an object property or datatype property and it has a domain declared (pattern A).
- is not defined as an object property and it has a class defined as range (pattern B).
- is not defined as a datatype property and it has a datatype defined as range (pattern C).
- is not defined as an object property or datatype property and it appears as subject (pattern D) or object (pattern E) in an `rdfs:subPropertyOf` statement.
- is not defined as an object property or datatype property and it appears as subject (pattern F) or object (pattern G) in an `owl:propertyDisjointWith` statement.
- is not defined as an object property or datatype property and it appears as subject (pattern H) or object (pattern I) in an `owl:equivalentProperty` statement.
- is not defined as an object property and it appears as subject (pattern J) or object (pattern K) in an `owl:inverseOf` statement.

If at least one of the above-mentioned conditions is met, then a pitfall is flagged.

Limitations

This method does not check the ontology elements appearing in class description axioms as: (a) value constrains (for example in `owl:allValuesFrom` or `owl:someValuesFrom` constructs); or (b) cardinality constraints (for example in `owl:maxCardinality` or `owl:minCardinality` constructs).

Patterns (Part A)

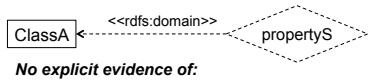
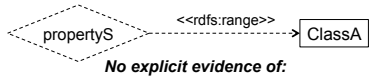
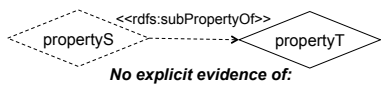
Graphical representation and/or OWL code	Natural language description
Along the graphical representations a dashed box is used to represent the ontology element that is not defined as a property in the patterns. This alternative notation is used for the sake of clarity using graphical patterns instead of functional syntax. A regular diamond is not used as it would be confused with elements actually defined as properties in the ontology.	
Pattern A:  ***No explicit evidence of:*** Declaration(ObjectProperty(propertyS)) Declaration(DataProperty(propertyS))	The graphical pattern shows an ontology element (propertyS) having a class defined as `rdfs:domain` statement. The ontology lacks the definition of such ontology element as an object or datatype property.
Pattern B:  ***No explicit evidence of:*** Declaration(ObjectProperty(propertyS))	The graphical pattern shows an ontology element (propertyS) having a class defined as `rdfs:range`. The ontology lacks the definition of such ontology element as an object property.
Pattern C:  ***No explicit evidence of:*** Declaration(DataProperty(propertyS))	The graphical pattern shows an ontology element (propertyS) having a datatype defined as `rdfs:range`. The ontology lacks the definition of such ontology element as a datatype property.
Pattern D:  ***No explicit evidence of:*** Declaration(ObjectProperty(propertyS)) Declaration(DataProperty(propertyS))	The graphical pattern shows an ontology element (propertyS) defined as subproperty of a property (propertyT). The ontology lacks the definition of such ontology element as an object or datatype property.

Table 5.42: Detection method proposed for P35 (Part A)

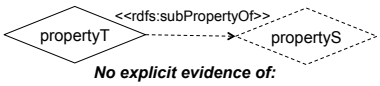
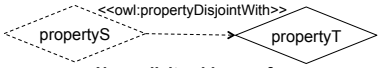
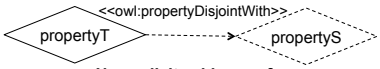
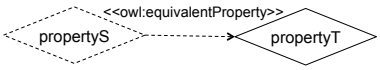
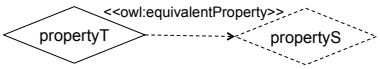
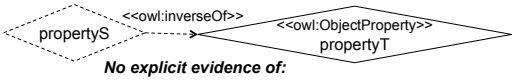
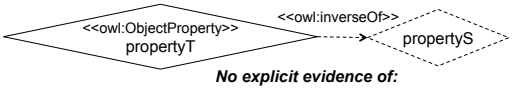
Code	M1- P35. Untyped property
Patterns (Part B)	
Graphical representation and/or OWL code	Natural language description
Pattern E:  ***No explicit evidence of:*** Declaration(ObjectProperty(propertyS)) Declaration(DataProperty(propertyS))	The graphical pattern shows a property (propertyT) defined as subproperty of an ontology element (propertyS). The ontology lacks the definition of such ontology element as an object or datatype property.
Pattern F:  ***No explicit evidence of:*** Declaration(ObjectProperty(propertyS)) Declaration(DataProperty(propertyS))	The graphical pattern shows an ontology element (propertyS) defined as disjoint with a property (propertyT). The ontology lacks the definition of such ontology element as an object or datatype property.
Pattern G:  ***No explicit evidence of:*** Declaration(ObjectProperty(propertyS)) Declaration(DataProperty(propertyS))	The graphical pattern shows a property (propertyT) defined as disjoint with an ontology element (propertyS). The ontology lacks the definition of such ontology element as an object or datatype property.
Pattern H:  ***No explicit evidence of:*** Declaration(ObjectProperty(propertyS)) Declaration(DataProperty(propertyS))	The graphical pattern shows an ontology element (propertyS) defined as equivalent to a property (propertyT). The ontology lacks the definition of such ontology element as an object or datatype property.
Pattern I:  ***No explicit evidence of:*** Declaration(ObjectProperty(propertyS)) Declaration(DataProperty(propertyS))	The graphical pattern shows a property (propertyT) defined as equivalent to an ontology element (propertyS). The ontology lacks the definition of such ontology element as an object or datatype property.
Pattern J:  ***No explicit evidence of:*** Declaration(ObjectProperty(propertyS))	The graphical pattern shows an ontology element (propertyS) defined as inverse of an object property (propertyT). The ontology lacks the definition of such ontology element as an object or datatype property.
Pattern K:  ***No explicit evidence of:*** Declaration(ObjectProperty(propertyS))	The graphical pattern shows an object property (propertyT) defined as inverse of an ontology element (propertyS). The ontology lacks the definition of such ontology element as an object or datatype property.

Table 5.43: Detection method proposed for P35 (Part B)

Code	M1- P36. URI contains file extension		
Pitfall cardinality	1	**Method addresses**	Ontology
Technique	Specific characteristic search	**Pitfall affects to**	Ontology
Description			
This method checks whether the ontology URI contains any of the file extensions "owl", "rdf", "n3" or "ttl". If case this happens, then the pitfall is flagged.			
Limitations			
The current implementation of the method only detects this pitfall when the user provides the ontology URI. In this implementation, the method does not check this pitfall when the user provides the ontology source code.			

Table 5.44: Detection method proposed for P36

Code	M1-P37. Ontology not available on the Web		
Pitfall cardinality	1	**Method addresses**	Ontology
Technique	Specific characteristic search	**Pitfall affects to**	Ontology
Description			
This method checks whether looking up the ontology URI, the OWL ontology code or an HTML document is retrieved. If none of the formats is retrieved, then the pitfall is flagged.			
Limitations			
The current implementation of the method only detects this pitfall when the user provides the ontology URI.			

Table 5.45: Detection method proposed for P37

Code	M1-P38. No OWL ontology declaration		
Pitfall cardinality	1	**Method addresses**	Ontology
Technique	Specific characteristic search	**Pitfall affects to**	Ontology
Description			
This method flags a pitfall when the ontology lacks the ontology declaration (definition of the ontology as type of rdf:type "http://www.w3.org/2002/07/owlOntology") or the ontology header (using the tag `owl:Ontology`.). The method applies regular expressions and string matching to detect the different ways of declaring ontology headers and stating the ontology declaration for RDF/XML and turtle syntaxes.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A: ***No explicit evidence of:*** Ontology(<http://ontologyURI>)		The graphical pattern shows an ontology that lacks the ontology declaration.	

Table 5.46: Detection method proposed for P38

Code	M1- P39. Ambiguous namespace		
Pitfall cardinality	1	**Method addresses**	Ontology
Technique	Specific characteristic search	**Pitfall affects to**	Ontology
Description			
This method reports a pitfall when the ontology lacks the definition of a base URI and the ontology declaration (definition of the ontology as type of rdf:type "http://www.w3.org/2002/07/owlOntology") or the ontology header (using the tag `owl:Ontology`.). The method applies regular expressions and string matching to detect the different ways of declaring base URIs, declaring ontology headers and stating the ontology declaration for RDF/XML and turtle syntaxes.			
Patterns			
Graphical representation and/or OWL code		Natural language description	
Pattern A: ***No explicit evidence of:*** @base <http://ontologyURI> . Ontology(<http://ontologyURI>)		The graphical pattern shows an ontology that lacks a base prefix definition and an ontology declaration.	

Table 5.47: Detection method proposed for P39

Code	M1- P40. Namespace hijacking		
Pitfall cardinality	N	**Method addresses**	Classes Object properties Datatype properties
Technique	Specific characteristic search	**Pitfall affects to**	Classes Object properties Datatype properties
Description			
This method completely relies on the third-party software TripleChecker [1]. TripleChecker checks whether the ontology elements (elements used as classes, object properties and datatype properties) presumably defined in external namespaces do actually retrieve any information in RDF/XML or turtle when looking up their URIs. TripleChecker sends an HTTP request with the accept header: `Accept: application/rdf+xml; q=0.9, text/turtle; q=0.8, */*; q=0.1`. For each ontology element for which no information is retrieved, the pitfall is flagged. TripleChecker also applies the above-mentioned check over namespaces. However, it is not considered in this method's output as it is not considered in the pitfall description of "P40. Namespace hijacking".			
Limitations			
This method only works with URIs as input and only checks elements used as classes or properties (it does not include elements defined as such), due to third-party software limitations.			
External resources used			
[1] http://graphite.ecs.soton.ac.uk/checker			

Table 5.48: Detection method proposed for P40

Code	M1- P41. No license declared		
Pitfall cardinality	1	**Method addresses**	Ontology
Technique	Specific characteristic search	**Pitfall affects to**	Ontology

Description

This method completely relies on the third-party software Licensius [1].
The functionality "getLicense" provided by Licensius checks whether an ontology contains a license declaration using any of the predicates: `dc:rights`, `dcterms:rights`, `dcterms:license`, `cc:license` or `xhv:license`.
This method reports a pitfall if none of the above-mentioned predicates are detected by Licensius (pattern A).

Limitations

Taken literally from Licensius [1]: *Other properties that could have been scanned (and are not) include:* `doap:license`, `premis:licenseTerms`, `omv:hasLicense`, `mo:License`, `vaem:hasLicenseType`, `terms:RightsStatement` *and* `terms:LicenseDocument`.

This method assumes that licenses are described using an RDF predicate for license definition. The analysis of natural language license information using natural language processing techniques is out of the scope.

Patterns

Graphical representation and/or OWL code	Natural language description
Pattern A: Prefix(dc:=<http://purl.org/dc/elements/1.1/ >) Prefix(dcterms:=<http://purl.org/dc/terms/>) Prefix(cc:=<http://purl.org/dc/terms/>) Prefix(xhv:=<http://www.w3.org/1999/xhtml/vocab#>) Ontology(<http://example.org/def/myontology.owl> ***No explicit evidence of:*** Annotation(dc:rights:license) Annotation(dcterms:rights :license) Annotation(dcterms:license :license) Annotation(cc:license :license) Annotation(xhv:license :license))	The graphical pattern shows an ontology that lacks the definition of license information. More precisely, none of the following predicates is declared in the ontology header: `dc:rights`, `dcterms:rights`, `dcterms:license`, `cc:license` and `xhv:license`.

External resources used

[1] http://licensius.appspot.com. Documentation available at http://cosasbuenas.es/static/licensius

Table 5.49: Detection method proposed for P41

5.4 Summary

This chapter has presented the third contribution of the thesis, the **C3: Design and implementation of detection methods for the pitfalls defined in the catalogue whenever it is possible**. This contribution includes a general description template, used later on for describing in detail the current 48 methods implemented to flag automatically 33 out of the 41 pitfalls. These descriptions include graphical representation of structural patterns, references to external resources and limitations of the implemented methods. The list of pitfalls for which no method has been proposed and the rations behind such situation have also been provided.

This chapter has also presented the fourth contribution of this thesis, the **C4: Technological support called OOPS!**. This system has been developed in order to help ontology developers to diagnose their ontologies according to the systematic approach proposed in this thesis. OOPS! architecture has been presented detailing main decisions taken at the beginning of the implementation. A general description of the system's logic and interaction as well as reused resources has also been commented.

In summary, this chapter has presented the work developed in this thesis in order to address the second objective **O2: To ease the ontology diagnosis activity by means of providing suitable technological support, lessening thus the effort required from ontology engineers** (see Section 3.1). These contributions represent a step forward in the state of the art of tools for ontology diagnose.

Chapter 6

Evaluation

6.1 Introduction

This chapter provides the qualitative evaluation of the contributions of this thesis. For doing so, we first verify the implementation of the methods proposed in Chapter 5 (Section 6.2) and then compare OOPS! functionalities and architecture with regards to other tools presented in the state of the art (Section 6.3). Section 6.4 presents an empirical study of the pitfalls detected over 969 ontologies. Section 6.5 details the OOPS! validation from final users' point of view. Finally, evidence of use and worldwide adoption and integration within third-part software is detailed in Section 6.6.

6.2 Software verification

In order to verify that OOPS! meets the requirements stablished by the methods proposed in Section 5.3, individual test ontologies have been defined for each pitfall to be checked. We followed a modular approach to generate the tests in order to maximize the independence among tests' results. In that case, undesired interactions between independent tests could occur. At the same time, this design is easier to maintain and update and allows the integration of several test with lower effort. If two or more pitfalls are going to be tested at once, the corresponding modular tests could be imported (using `owl:imports`) or merged in a new ontology file that includes as many tests as needed. Each test includes the data needed to test each pattern described in the methods. If there is more than one method proposed for the given pitfall they would be included in the same test file (except for P22 as explained below).

The test ontology files are accessible online[56] and their URIs follow the pattern: "http://oops.linkeddata.es/data/test" plus the code of the pitfall to be detected. For example, the URI "http://oops.linkeddata.es/data/testP02" will retrieve the ontology to test "P02. Creating synonyms as classes".There are some exceptions to this rule:

- For detecting "P22. Using different naming conventions in the ontology" (see Table 4.22) there are four methods proposed (see Table 5.23, Table 5.24, Table 5.25 and Table 5.26). As P22 is flagged when at least one pattern from the four methods is detected and the pitfall affects the ontology (its cardinality is 1), it is needed to create different test files, one for each method. If the four patterns were represented in one single file, the first pattern found would flag the pitfall and as the cardinality is 1 the rest of patterns would not be checked. This case is different from other pitfalls with several methods but with cardinality N (namely, P08, P11, P12, P19, P20, P24 and P27), in which all the methods are checked. For these reason we need to check each method independently of the other three. Therefore, there four tests to check that the pitfall is detected by any of the proposed methods are:
  - http://oops.linkeddata.es/data/testP22M1 (for method "M1-P22. Detecting different naming conventions across ontology elements" in Table 5.23)
  - http://oops.linkeddata.es/data/testP22M2 (for method "M2-P22. Detecting different naming conventions across classes" in Table 5.24)
  - http://oops.linkeddata.es/data/testP22M3 (for method "M3-P22. Detecting different naming conventions across object properties" in Table 5.25)
  - http://oops.linkeddata.es/data/testP22M4 (for method "M4-P22. Detecting different naming conventions across datatype properties" in Table 5.26)
- For the case of the method "M1- P36. URI contains file extension" the URI to be used is http://oops.linkeddata.es/data/testP36.owl
- For the case of "M1-P37. Ontology not available on the Web" the URI to be used is "http://oops.linkeddata.es/data/testP37". However, no file (HTML nor RDF) is retrieved.

[56]The tests are listed in http://oops.linkeddata.es/tests (last visited on the 16th January, 2016)

All the tests can be executed using either their URI or their OWL code except for the methods "M1- P36. URI contains file extension" and "M1-P37. Ontology not available on the Web" that can be only be check using the URI.

The 48 methods proposed for detecting 33 pitfalls have been tested and the results have satisfactorily verified the implementation of such methods except for the pitfall "P40. Namespace hijacking" due to third-party software limitations.

6.3 OOPS! comparison with existing ontology evaluation tools

Several tools for ontology evaluation were presented in the state of the art (see Section 2.5). This section aims at providing insight of to what extent OOPS! outperforms, increases or improves ontology evaluation functionalities offered by existing tools. To do so, we compare, in the following subsections, the number of pitfalls detected by each tool and the software characteristics of the tools.

6.3.1 Pitfall coverage

This section compares OOPS! against similar tools for ontology diagnose based on a check-list approach focused on the T-Box. Table 6.1 provides a comparison between OOPS! and XD-Analyzer, Moki and OntoCheck. Table 6.1 lists all the pitfalls (rows) defined in the catalogue (See Section 4.2) regardless they are implemented or not in OOPS!. For each pitfall it is indicated whether a given tool (columns) fully covers the pitfall detection by means of the symbol "✓" or whether it is partially covered "≈". Empty cells mean that the tool does not address the pitfall detection at all.

Table 6.1 shows that OOPS! covers a wider range of pitfalls: OOPS! diagnoses 33 pitfalls, XD-Analyzer covers 7, OntoCheck 4 and Moki 3.

It can be observed that **XD-Analyzer** provides a more complete detection for pitfall "P04. Creating unconnected ontology elements" than OOPS! and Moki. OOPS and Moki only detect such pitfall in classes while XD-Analyzer addresses also object and datatype properties.

As Table 6.1 takes as reference the pitfall catalogue presented in Section 4.2, this table does not include those common errors detected by the analysed tools that are not directly mapped to any of such pitfalls. Next, we list those common mistakes detected

Pitfall	OOPS!	XD-Analyzer	Moki	Onto-Check
P01. Creating polysemous elements				
P02. Creating synonyms as classes	✓			
P03. Creating the relationship "is" instead of using "rdfs:subClassOf", "rdf:type" or "owl:sameAs"	✓			
P04. Creating unconnected ontology elements	≈	✓	≈	≈
P05. Defining wrong inverse relationships	✓			
P06. Including cycles in a class hierarchy	✓			
P07. Merging different concepts in the same class	✓			✓
P08. Missing annotations	✓	✓	✓	✓
P09. Missing domain information				
P10. Missing disjointness	✓			
P11. Missing domain or range in properties	✓	✓	✓	
P12. Equivalent properties not explicitly declared	✓			
P13. Inverse relationships not explicitly declared	✓	✓		
P14. Misusing "owl:allValuesFrom"				
P15. Using "some not" in place of "not some"				
P16. Using a primitive class in place of a defined one				
P17. Overspecializing a hierarchy				
P18. Overspecializing the domain or range				
P.19 Defining multiple domains or ranges in properties	✓	✓		
P20. Misusing ontology annotations	✓			
P21. Using a miscellaneous class	✓			
P22. Using different naming conventions in the ontology	✓			✓
P23. Duplicating a datatype already provided by the implementation language				
P24. Using recursive definitions	≈			
P25. Defining a relationship as inverse to itself	✓			
P26. Defining inverse relationships for a symmetric one	✓			
P27. Defining wrong equivalent properties	✓			
P28. Defining wrong symmetric relationships	✓			
P29. Defining wrong transitive relationships	✓			
P30. Equivalent classes not explicitly declared	✓			
P31. Defining wrong equivalent classes	✓			
P32. Several classes with the same label	✓			
P33. Creating a property chain with just one property	✓			
P34. Untyped class	✓	✓		
P35. Untyped property	✓	✓		
P36. URI contains file extension	✓			
P37. Ontology not available on the Web	✓			
P38. No OWL ontology declaration	✓			
P39. Ambiguous namespace	✓			
P40. Namespace hijacking	✓			
P41. No license declared	✓			

Table 6.1: Comparative of pitfall coverage between tools

by other tools, namely XD-Analyzer and Moki, which are not currently implemented in OOPS! or included in Table 6.1.

The features covered by XD-Analyzer that are not included in OOPS! are listed below. The descriptions of the issues provided by XD-Analyzer are highlighted in italics and taken literally from XD-Analyzer output. For each issue, the relation between each issue and this thesis is explained:

XD-1: Missing type. *Each entity must be the instance of something. This is valid for entities of the T-Box (e.g. a class should be an instance of owl:Class) as well as entities in the A-Box.*

- This issue is considered an "Error" by XD-Analyzer. OOPS! partially covers in "P34. Untyped class" and "P35. Untyped property" this issue regarding the T-Box. The A-Box part is out of scope of this thesis according to restriction R2 (Section 3.5).

XD-2: Architectural import notice. *Most of the locally defined entities do not specialize imported entities.*

- This issue is considered a "Suggestion" by XD-Analyzer as it is a good practice to specialize imported entities. However the lack of such a good practice is not considered a pitfall, therefore it is not part of our catalogue yet.

XD-3: Unused imported ontology. *All imported ontologies should have at least one entity referenced locally. The ontology imports another ontology but does not use any of its declared entities.*

- This issue is considered a "Warning" by XD-Analyzer as it is a good practice to reference imported entities. However the lack of such a good practice is not considered a pitfall, therefore it is not part of our catalogue yet.

Good practices, as the above-mentioned "Architectural import notice" and "Unused imported ontology" could be implemented in further OOPS! extensions as suggestions.

The comparison of **Moki** and OOPS! only takes into account those checks related to the "Domain model" provided by Moky as they are related to ontology elements (named in Moki as concepts, individuals and properties). Moki also provides some checks for

"Process model" and "Integrated model" that are out of scope of this thesis. In the following list, the descriptions of the issues provided by Moki are highlighted in italics and taken literally from Moki's documentation (Pammer, 2010). The relation between each issue detected by Moki and this thesis is explained next:

Moki-1: Concepts without individuals. *Concepts without individuals are not instantiated by any individual. This is not a modelling mistake by itself* (Pammer, 2010).

- This issue is not considered in this thesis as it is related to the A-Box, which is out of scope of this thesis according to restriction R2 (Section 3.5).

Moki-2: Individuals with no type defined. *This means that the model contains only the knowledge that these individuals exist, but not for instance of what kind they are (their type). It is strongly suspected that these individuals are either redundant, or a type is known but has by mistake not been defined for them* (Pammer, 2010).

- This issue is not considered in this thesis as it is related to the A-Box, which is out of scope of this thesis according to restriction R2 (Section 3.5).

Moki-3: Non-shared concepts. *Non-shared concepts have been edited only by a single person and thus may not represent a shared view. Consider asking multiple persons for feedback or input* (Pammer, 2010).

Moki-4: Non-shared properties. *Non-shared properties have been edited only by a single person and thus may not represent a shared view. Consider asking multiple persons for feedback or input* (Pammer, 2010).

The issues "Moki-3: Non-shared concepts" and "Moki-4: Non-shared properties" are related to the collaborative aspect of ontology development addressed in Moki. They are considered as a particular issue of the method supported by Moki (which is not consider a pitfall in other developments or methodologies).

OntoCheck does not detect any other common error than those included in Table 6.1.

Table 6.1 reveals that there is a number of pitfalls not detected by any of the current ontology evaluation tools. These pitfalls are: P01, P09, P14, P15, P16, P17, P18 and P23. Explanations regarding why OOPS! does not address them have been provided in Table 5.1.

Eyeball is not considered in Table 6.1 because this tool is more focused on the syntax level and data-driven common errors rather than on the OWL schema. It is worth noting that the type of diagnose performed by Eyeball and OOPS! do not overlap much. In addition, Eyeball is considered to be integrated within OOPS! in further versions, for example to extend its functionality including the A-Box diagnose, which is currently out of scope of this thesis according to restriction R2 (Section 3.5).

OQuaRE is not considered in Table 6.1 because this tool does not check common errors but computes measurements from the ontology structure and compares these measurements to certain predefined values. The values obtained can not be mapped directly to the list of pitfalls or common errors provided by the rest of systems compared in this conceptual study.

6.3.2 Software characteristics comparison

Several tools for helping users in evaluating OWL ontologies have been proposed and presented in Section 2.5. This section compares OOPS! with existing ontology evaluation systems, namely Moki, OQuaRE, XD-Analyzer, OntoCheck and Eyeball. The systems are compared according to the following characteristics, regarding software distribution types and ease of use:

- **GUI:** whether the system provides a Graphical User Interface.
- **API:** whether the system provides an Application Programming Interface. As a Web Service is a remote API, web services are considered in this characteristic.
- **No installation process:** whether the user does not need to follow an installation process in order to use the system.
- **Independence of ontology editors:** whether the system can be used independently of existing ontology development environments or editors.
- **Custom evaluation:** whether the system allows personalized evaluations.
- **Offline use:** whether the system could be executed offline.

Table 6.2 shows the comparison between tools in which a cell containing a "✓" symbol means that the system has that characteristic, while a cell containing a "≈" symbol means that the tool partially covers such characteristic. An empty cell means that the tool does not cover the characteristic at all. Tools in the table are classified according to three different types of tools namely "web based applications", "plug-ins" and "desktop applications or APIs".

		Web based			Plug-in		Desktop / API
		OOPS!	Moki	OQuaRE	XD-Analyzer	Onto-Check	Eyeball
Characteristic	GUI	✓	✓	✓	✓	✓	≈
	API	✓		✓			✓
	No installation process	✓		✓			
	Independence of ontology editor	✓		✓			✓
	Custom evaluation	✓		≈			✓✓
	Offline use		✓		✓	✓	✓

Table 6.2: Comparative of tools according to usability characteristics

It can be observed that most of the systems provide a **GUI**. So far, Eyeball provides only an experimental GUI. Therefore, it is considered to cover this characteristic only partially.

Regarding whether an **API** is provided, only OOPS!, Moki and Eyeball cover this characteristic. OOPS! and Moki provide a Web Service while Eyeball provides a Java API.

Only the web-based applications OOPS! and OQuaRE do not require any **installation process**. Even though Moki is also web-based, it does require installation process as the wiki environment should be set up.[57]

Regarding whether the tools are **independent of ontology editors** we observe that OOPS!, OQuaRE and Eyeball can be used without installing any ontology edition platform. However, this characteristic implies a drawback, as users are not able to modify their ontologies, within the ontology evaluation tool, according to the evaluation

[57]The online version (last visited on the 28th October, 2015) only provides a demo for testing purposes.

results. Therefore, users need to use an ontology editor in case they need to repair their ontologies.

For the case of **custom evaluation**, OQuaRE covers it partially as it provides a Web Service to evaluate particular metrics. However no combination of such metrics is allowed and its web user interface only allows the evaluation of the whole batch of metrics defined. On the other hand, Eyeball obtains a double "✓" for this characteristic as it not only allows the selection of a subset of inspectors but also to create customized inspectors by means of SPARQL queries according to the users' needs. OOPS! allows the user to choose subset of pitfalls according to the dimensions and aspects detailed in Section 4.4 and also to choose particular pitfalls.

OOPS! as well as OQuaRE do not provide the means to be **executed offline**. For the OOPS! case, this issue is a consequence of choosing no installation process required and a system independent of ontology editors to the detriment of offline availability, as described in Chapter 5. Addressing this shortcoming, for example by means of providing plug-ins for ontology editors, opens new lines of work.

Finally, regarding the characteristic **offline use**, it should be mentioned that Moki, which is a web-based system can be executed offline. This is due to the fact that the full functionalities are provided to the user in an installer in order to set up a wiki instance, therefore user could install and run it locally even though it is based on web technologies.

6.4 Software validation

In order to know which are the most frequent errors in ontology development, we have recorded the number of pitfalls detected in each ontology diagnosed with OOPS!. To carry out this task we used the 33 pitfalls implemented (See Table 5.1 for the list of implemented pitfalls) and 969 ontologies diagnosed up to August 2015.

When analysing OOPS! execution logs we noticed that we could find ontologies identified by their URIs as well as anonymous ontologies, that is, ontologies that are not identified by any URI. In addition, a given ontology can appear in the log several times. A graphical analysis of the different ontologies diagnosed is shown in Figure 6.1. Such drilldown report of the registered ontologies was derived as follows:

- Between November 14th, 2011 and August 24th, 2015, 2,753 executions were carried out.[58] During these executions, the ontology being analysed was identified by its URI in 2,532 cases, whereas the ontology was "anonymous" (its URI was not defined or it was "null") in 221 cases.

- From these 2,532 ontologies identified, some URIs indicate that the same ontology has been evaluated several times. We have filtered duplicated URIs, keeping only the first execution per URI. As a result, we counted 852 unique ontologies. Further studies will take into account all the executions per URI and analyse the evolution of the pitfalls appearing.

- With regard to the 221 anonymous ontologies, we have removed executions with equal results, assuming that they belong to the same ontology, thus avoiding duplications. As a result, we counted 117 different anonymous ontologies.

- Overall, OOPS! has analysed 969 ontologies (852 with URI and 117 anonymous). This set of random ontologies submitted by OOPS! users contains upper level ontologies, as well as domain ontologies. These ontologies were developed either by domain experts, students, newcomers or ontology experts.

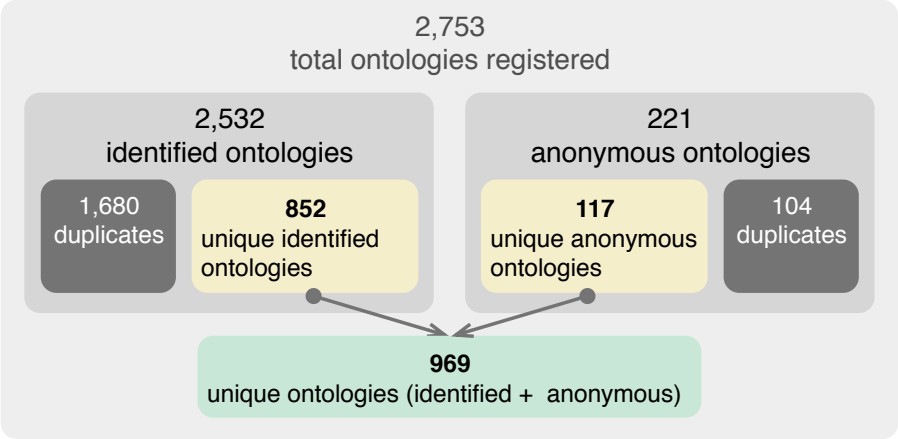


Figure 6.1: Ontologies registered drilldown report

[58]It should be mentioned that these executions correspond to an OOPS!'s registry, where the system registers information about the executions, instead of the server log or the Google Analytics' one. As already commented, there was a lost of such server logs, therefore this registry contains executions results not countered in the server logs.

Figure 6.2 shows the number of ontologies in which each pitfall have been detected. This information is split according to the updates on the number of pitfalls implemented in OOPS! and temporarily contextualized since the pitfalls detection is available:

- Figure 6.2 a) presents in how many ontologies have been detected the first 21 implemented pitfalls since November 2011. These pitfalls have been evaluated over 969 ontologies.
- Figure 6.2 b) presents in how many ontologies have been detected the 11 pitfalls (from P30 to P40) added in September 2013. These pitfalls have been evaluated over 410 ontologies.
- Figure 6.2 c) presents in how many ontologies has been detected the new pitfall added in March 2015. This pitfall has been evaluated over 329 ontologies.

Finally, Figure 6.3 shows in how many ontologies each implemented pitfall has been diagnosed. It shows an overview of the pitfalls detected regardless the updates on the number of pitfall implemented. Therefore, the number of pitfalls implemented along the table is not uniform. As already mentioned, pitfalls from P30 to P40 (marked with a * in Figure 6.3) were added in September 2013 and P41 (marked with ** in Figure 6.3) was added in March 2015.

Figure 6.3 reveals that most common pitfalls in ontologies are those related to the lack of explicit human and machine-readable information. However, these pitfalls do not correspond to those defined as critical by ontology practitioners but to those defined as "important" or "minor".

It is worth noting that "P41. No license declared", has been placed 7^{th} in the ranking. This is a very significative fact as it represents one of the most common pitfalls even though it has been the last pitfall added, being evaluated only over 329 ontologies. In a similar way, several of the pitfalls added in September 2013 (namely, P35, P40, P30, P36 and P38) have reached positions in the top half of the ranking. Finally, Table 6.3 shows the list of pitfalls ordered by percentage of ontologies in which a given pitfall is found regarding the number of ontologies in which the pitfall has been evaluated. For example, P41 is in the 4^{th} position as it has been found in 66.57% of the ontologies in which it has been tested (219 out of 329).

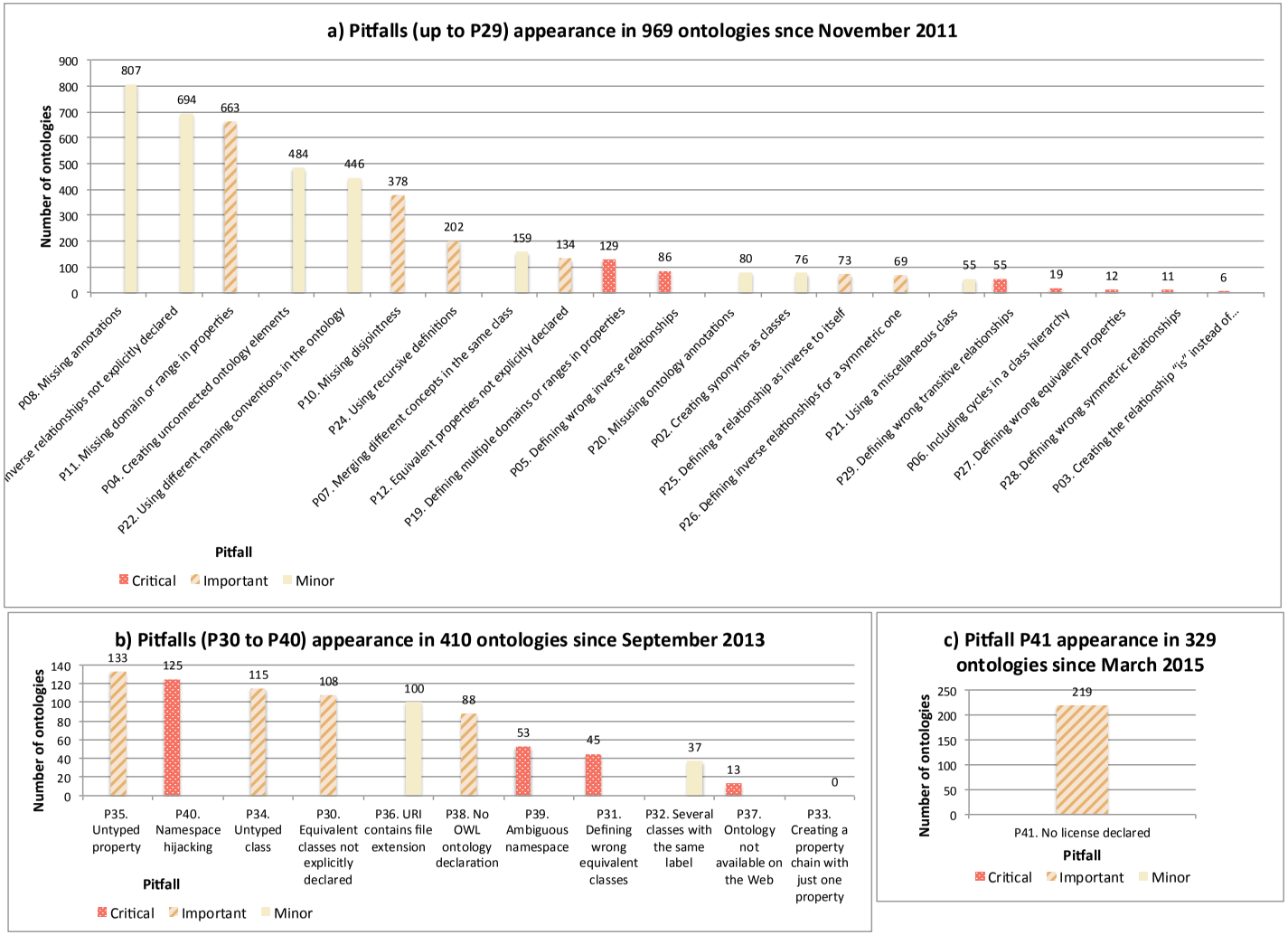


Figure 6.2: Most frequent pitfalls diagnosed by OOPS! split by number of pitfalls implemented.

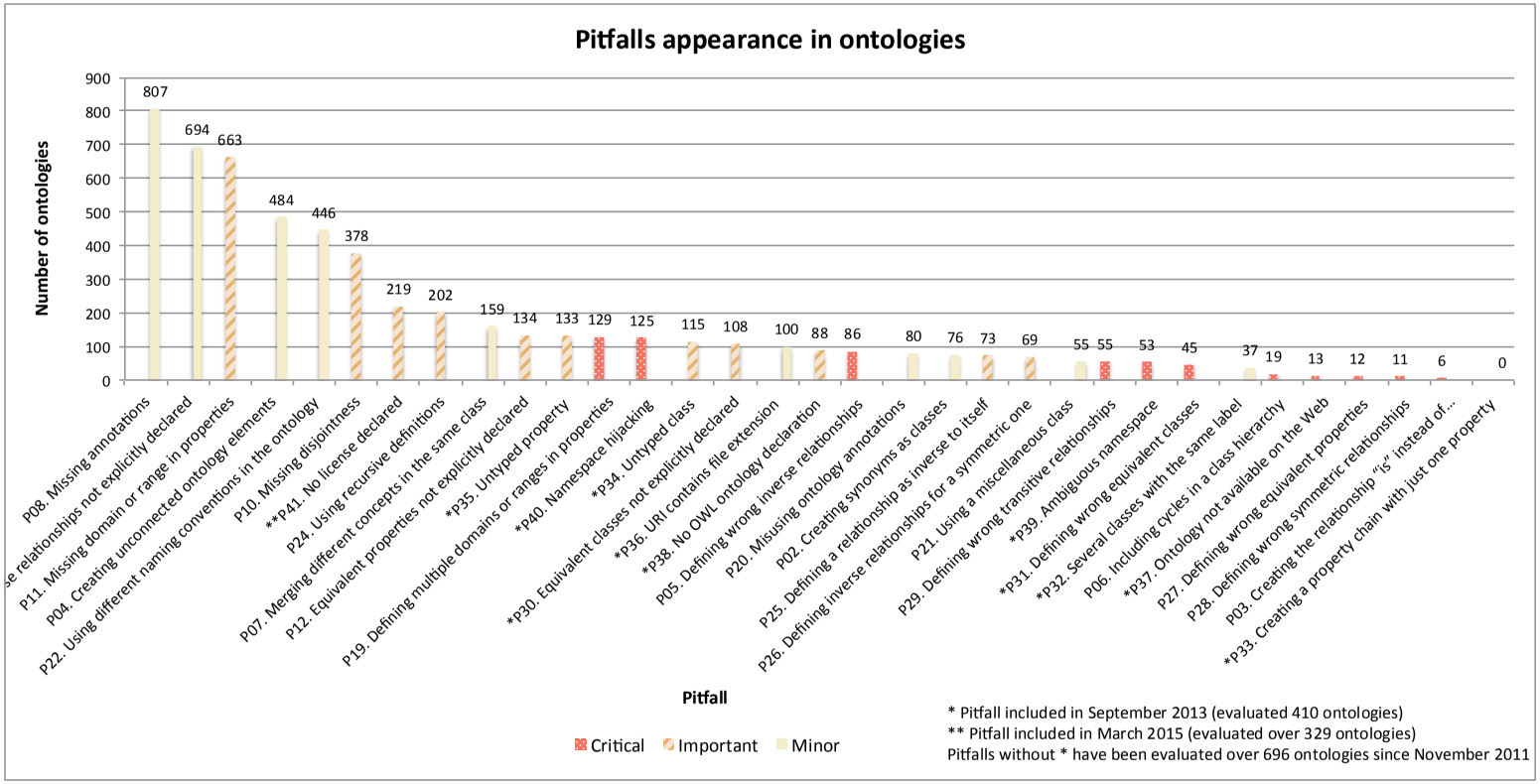


Figure 6.3: Most frequent pitfalls diagnosed by OOPS! in a set of 969 ontologies

Pitfall	# detected	# ontologies	%
P08. Missing annotations	807	969	83.28
P13. Inverse relationships not explicitly declared	694	969	71.62
P11. Missing domain or range in properties	663	969	68.42
*P41. No license declared	219	329	66.57
P04. Creating unconnected ontology elements	484	969	49.95
P22. Using different naming conventions in the ontology	446	969	46.03
P10. Missing disjointness	378	969	39.01
*P35. Untyped property	133	410	32.44
*P40. Namespace hijacking	125	410	30.49
*P34. Untyped class	115	410	28.05
*P30. Equivalent classes not explicitly declared	108	410	26.34
*P36. URI contains file extension	100	410	24.39
*P38. No OWL ontology declaration	88	410	21.46
P24. Using recursive definitions	202	969	20.85
P07. Merging different concepts in the same class	159	969	16.41
P12. Equivalent properties not explicitly declared	134	969	13.83
P19. Defining multiple domains or ranges in properties	129	969	13.31
*P39. Ambiguous namespace	53	410	12.93
*P31. Defining wrong equivalent classes	45	410	10.98
*P32. Several classes with the same label	37	410	9.02
P05. Defining wrong inverse relationships	86	969	8.88
P20. Misusing ontology annotations	80	969	8.26
P02. Creating synonyms as classes	76	969	7.84
P25. Defining a relationship as inverse to itself	73	969	7.53
P26. Defining inverse relationships for a symmetric one	69	969	7.12
P21. Using a miscellaneous class	55	969	5.68
P29. Defining wrong transitive relationships	55	969	5.68
*P37. Ontology not available on the Web	13	410	3.17
P06. Including cycles in a class hierarchy	19	969	1.96
P27. Defining wrong equivalent properties	12	969	1.24
P28. Defining wrong symmetric relationships	11	969	1.14
P03. Creating the relationship “is” instead of using “rdfs:subClassOf”, “rdf:type” or “owl:sameAs”	6	969	0.62
*P33. Creating a property chain with just one property	0	410	0.00

Table 6.3: Most frequent pitfalls ordered by percentage appearance. The percentage is relative to the number of ontologies in which each pitfall has been evaluated.

An analysis that complements this study is presented in (Keet et al., 2013). In their work, the authors examine the pitfalls detected in three different sets of ontologies, namely: (1) 23 ontologies in different domains (furniture, tennis, bakery, cars, soccer, poker, birds, and plants) developed by novices; (2) 21 existing well-known ontologies that may be considered "mature" in the sense of being a stable release of a real OWL ontology; and (3) 362 ontologies analysed by OOPS! selected from the 614 times that ontologies were submitted between 14th of November, 2011 and 19th of October, 2012. The pitfalls implemented by the time of developing the study were: P02, P03, P04, P05, P06, P07, P08, P10, P11, P12, P13, P19, P20, P21, P22, P24, P25, P26, P27, P28 and P29. This study concludes that in most of the cases there is no clear evidence of noteworthy differences between the ontologies extracted from OOPS! log, the ones developed by novices and the well-known ontologies. Therefore, even though the lack or appearance of pitfalls is considered a sign of quality, it could not be considered a measure of maturity in ontologies. According to the study, the maturity of an ontology would not be characterised by absence of pitfalls at all, but instead it would be defined by something else. Such other characteristic to define an ontology as mature could be its usefulness for its purpose (or at least meeting the requirements) or more abstract notions as the precision and coverage.

6.5 User based evaluation

OOPS! main goal is to get the outcomes of the ontology diagnose closer to ontology developers, mainly newcomers and domain experts who are not familiar with description logics and ontology implementation languages. In order to proof that OOPS! really helps users, the system has been used in different research and educational projects in which positive feedback and interesting comments from the ontology developers involved have been gathered.

In the following sections we describe different user-based evaluation settings, presenting qualitative results and quantitate results whenever it is possible. The different scenarios where OOPS! has been used by users are represented chronologically in Figure 6.4. This figure summarizes the three scenarios including the number and the characteristics of the participants, the status of development of OOPS! and the type of feedback received. As it can be observed in the figure, we tried a first version of

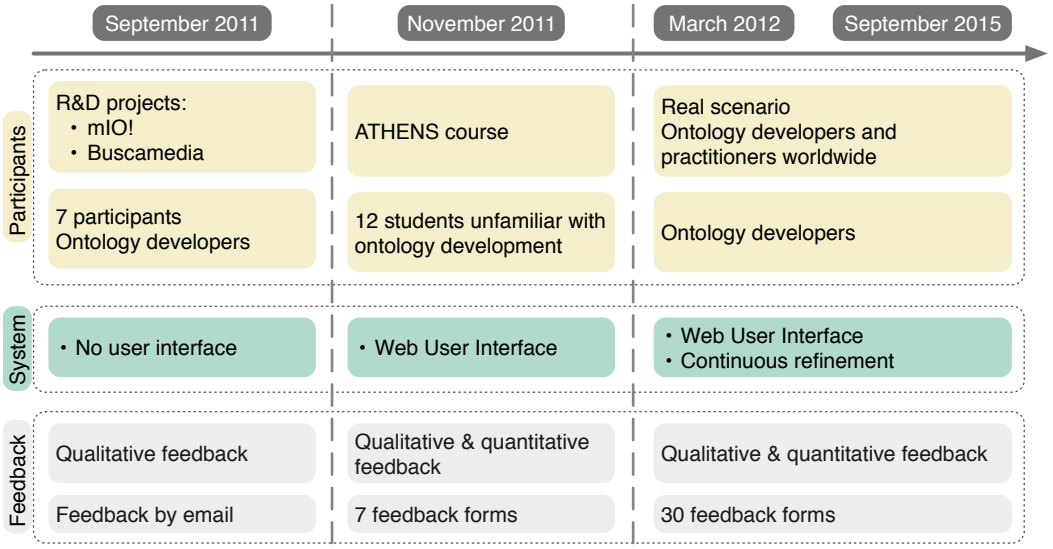


Figure 6.4: User test scenarios

the tool without providing graphical interface in the context of two research and development projects with Spanish industry partners, namely mIO! (CENIT-2008-1019) and Buscamedia (CENIT-2009-1026), which results are provided in Section 6.5.1. Next, we carried out an experiment in a controlled environment during an ATHENS course[59] providing the first user interface as detailed in Section 6.5.2. Finally, Section 6.5.3 shows the feedback gathered from real users after OOPS! was announced to the semantic web community in March 2012.

6.5.1 R&D projects application case

The first case is the use of OOPS! in the context of two Spanish research and development projects called mIO![60] and Buscamedia.[61] Both projects involved the development of two ontologies about user context in mobile environments and multimedia information objects respectively with Spanish industry partners. In both projects the ontology evaluation activity was carried out by ontology engineers from the industry partners using OOPS!. After the diagnosis activity, the ontologies were repaired accordingly to OOPS! evaluation results. It is worth mentioning that by the time when the ontology

[59] http://www.athensprogramme.com/main (last visited on the 8th January, 2016)
[60] http://www.cenitmio.es/ (last visited on the 24th November, 2015)
[61] http://www.cenitbuscamedia.es/ (last visited on the 24th November, 2015)

evaluation activity was carried out (September 2011), OOPS! did not provide a graphical user interface. OOPS! was provided to ontology developers involved in the projects as a ".jar" file and its output was given in a ".txt" file.

A total of seven ontology engineers were involved in the ontology evaluation activity within mIO! and Buscamedia use cases. Such ontology engineers provided by email qualitative and positive feedback about:

- OOPS! usefulness
- the advantages of being IDE independent
- the broader coverage of pitfalls detected by OOPS! in comparison with other tools

Participants of the projects application use case also provided very valuable feedback about aspects that could be improved in OOPS! as for example:

- providing a graphical user interface
- providing more information about what the pitfalls consist in
- considering the imported ontologies during the analysis
- to allow the evaluation of subsets of pitfalls

All these comments were taking into account and implemented in subsequent releases of OOPS!. Other suggestions such as to provide some recommendations to repair the pitfalls found within the ontology are currently included in the pitfall catalogue (See Section 4.2) although they are not yet implemented.

6.5.2 ATHENS course case

The second case refers to a controlled experiment to test the benefits of using OOPS! during the ontology evaluation activity that was carried out with undergraduate students. This experiment was performed during the ATHENS course that took place in November 2011 at Universidad Politécnica de Madrid. Twelve undergraduate students working in pairs executed the experiment. In this case, the users were not experts in Ontology Engineering and the ATHENS course was their first contact with semantic technologies. Before the experiment, students were provided with:

- theoretical lessons about knowledge engineering and OWL language

- some explanations about OOPS! and ontology evaluation concepts
- the detailed tasks to be performed during the experiment using OOPS! through the first prototype of the GUI
- two different ontologies to be evaluated prepared for the experiment

After the experiment, we gathered students' feedback using questionnaires (See Annex B). Most of the students considered that OOPS! was very useful to evaluate the ontologies at hand and that its output shows clearly which problems were detected and in which elements. Within the feedback gathered in this experiment, the students suggested the following features:

- to include guidelines about how to solve the detected pitfalls
- to associate colours to the outputs indicating how critical the pitfalls are, like errors and warnings
- to provide a way to see the lines of the file that the error considered is originated from

The first suggestion has been included in the pitfall catalogue. However, it has not been implemented in OOPS! yet. The second suggestion has been included both in OOPS! and in the pitfall catalogue by means of assigning importance levels to each pitfall (see Section 4.3) and assigning a colour to each importance level in the user interface. However, the third suggestion has not been implemented yet, as not always a specific single line of the source code could be pointed out. For example, in those cases where the pitfall refers to lack of information in the ontology. In summary, the parallelism between ontology code and programming code can not always be done. Detecting a pitfall is not equivalent to pointing a line of code that makes the compilation process to fail.

Regarding quantitate results extracted from the student questionnaires, we do not consider including graphics as they are not too representative due to the low number of participants.

6.5.3 Real scenario case

OOPS! was announced on the 6th of March, 2012 thorough several mailing lists[62] related to the Semantic Web and Ontological Engineering. The goal was to invite the ontology developers community to diagnose their ontologies and to collect their feedback through the questionnaire included in Annex B. Up to now 2753 ontology evaluations have been registered (see Figure 6.1) and we have received 30 feedback questionnaires from users not related to any project or any controlled experiment.

Along these questionnaires some strengths of the tool were explicitly pointed out by users as:[63] *"easy to use"*, *"The main contribution is in providing a valuable tool for improving the quality of ontologies"*, *"quick results"* and *"the capability of find many different types of errors in the ontology, to show what errors were found, explain what those errors mean or how it could affect the ontology and suggest a way to fix them"*. In that questionnaire, users also indicated how the system effectively improved their ontologies and helped them in the process of ontology evaluation. In this regard, users mainly pointed out that OOPS! was useful for (a) discovering potential missing statements (e.g. human readable annotations, domain and range declarations and property characterization as inverse, among others), (b) detecting incorrect pairs of inverse properties, (c) enriching property definitions (e.g. by adding the symmetric or transitive characteristic).

However, the richest side of this feedback is the set of proposals to improve the tool. The most important feedback in this regard refers to:

- show which pitfalls do not appear in the ontology
- include reasoning processes so that OOPS! would not complain when a property inherits domain/range from its superproperty (this issue is related to restriction R2 of this thesis (see Section 3.5))
- allow the evaluation of subsets of pitfalls
- consider other natural language annotation properties apart from `rdfs:label` and `rdfs:comment`

[62] For example http://lists.w3.org/Archives/Public/semantic-web/2012Mar/0064.html
[63] The following comments have been taken literally from the feedback questionnaires.

As already mentioned (see Section 6.5.1) the evaluation of subsets of pitfalls is included in the system. In further software extensions we plan: (a) to include reasoning features as an option; (b) to implement the list of pitfalls that has not been included yet; and (c) to suggest the user to manually look for non implemented pitfalls when needed.

In the following we provide some quantitative results. Within the questionnaire we included four rating scale questions. For this type of questions participants specify their level of (dis)agreement on a symmetric agree-disagree scale. These questions are:

a) The output generated by OOPS! shows clearly what is the problem detected and in which elements.
   ▫ Strongly Agree ▫ Agree ▫ Undecided ▫ Disagree ▫ Strongly Disagree

b) In your ontology developments, do you think you will use again OOPS! for validating your ontology?
   ▫ Yes, always ▫ Yes, sometimes ▫ No

c) In general, do you think OOPS! is:
   ▫ Very useful ▫ A nice gadget ▫ Not good enough

d) Would you recommend OOPS! to other colleagues involved in ontology development projects?
   ▫ Yes, always ▫ Yes, sometimes ▫ No

The questionnaire asked participants how easy it was for them to understand OOPS!'s output (question "a"). Figure 6.5 a) shows the results for this question. It can be observed that the result is quite positive as the majority consider the output clear (agree 40%) or very clear (strongly agree 47 %) while only 2 out of 30 respondents could not decide about it and other 2 think that the output is not clear enough.

Figure 6.5 b) shows the responses to whether the users would use again the system (question "b"). In this regard, we also obtained positive results as only 1 out of 30 respondents answered "No", while 64% would always use it again and 33% would sometimes use it again.

Regarding the usefulness of the system (question "c"), Figure 6.5 c) displays that 80% of the users think that the system is very useful.

Finally, Figure 6.5 d) shows the response to whether the users would recommend OOPS! to their colleagues (question "d"). In this regard, we also obtained very positive

results as 73% of the users would always recommend the system and 23% of them only sometimes, while just 1 user would not recommend it.

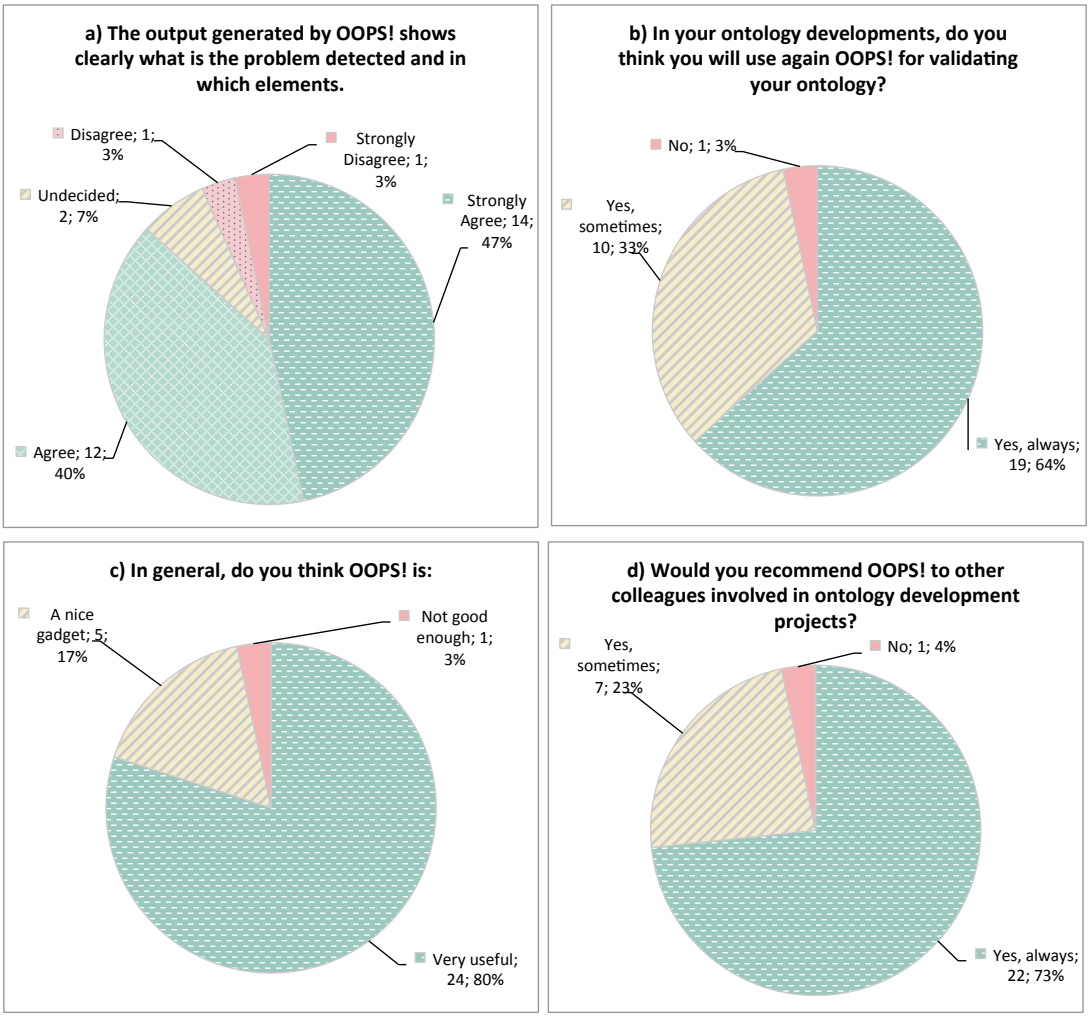


Figure 6.5: Quantitative results from feedback questionnaires

It is worth noting than 13 of the participants said that they had never used an ontology evaluation system before while, 11 claimed to have done so, having 6 empty answers to this question. Those users having used other ontology evaluation systems, they mentioned the use of reasoners for inconsistency checking and syntax validators. A few users mentioned other systems as Eyeball, Vapour or OWL Lint[64] among others. In general, this group of users gave positive answers to the four questions showed in

[64]http://protegewiki.stanford.edu/wiki/OWL_Lint (last visited on the 24th November, 2015)

Figure 6.5. In addition, we can highlight the following comments from these users when being asked about previous experience with ontology evaluation tools:[65] *"this is the first that was easy to use and makes sense"*; *"not really evaluation tools, but syntax checkers and reasoners for consistency and to identify undesired inferences"*; or *"In the past I used reasoners to have an idea of the ontology design. However after using oops I realized that it was a bad idea."*

In addition to the feedback received through the web questionnaire, we have also received comments and questions about OOPS! by email, what reveals users willingness to adopt this type of tools within their ontology developments. Users also pointed out the need of developing systems like OOPS! in the context of Ontological Engineering. In addition, the following improvements were suggested:

- to discard pitfalls involving terms properly marked as DEPRECATED following the OWL 2 deprecation pattern
- to take into account the different namespaces used within the ontology
- to look for URI misuse, for example when using the same URI as two different types of ontology elements
- to look for non-standard characters in natural language annotations.

6.6 System adoption

This section shows the acceptance of OOPS! from the Semantic Web community. To do so, this section provides evidence of OOPS! use worldwide, including specific ontology development projects in which the use of OOPS! has been reported in scientific publications, and six examples of integration of OOPS! functionality within third-party software.

6.6.1 User adoption worldwide

Next, we present some evidence of how OOPS! has been used and adopted worldwide up to August 24$^{\text{th}}$, 2015. To do so, we have analysed the log files from the server and the

[65] The following comments have been taken literally from the feedback questionnaires.

reports from Google Analytics service.[66] From these logs we state that OOPS! website has been visited more than 4,000 times from around 90 different countries, and that the system has been executed from around 60 different countries.[67] Figure 6.6 displays from how many counties OOPS! have been run. It also indicates the top ten countries from where it is used more often, including the number of executions registered.

Other uses of OOPS are reported in scientific publications. For example, in the biomedicine domain (see Beißwanger's PhD dissertation (Beißwanger, 2013) page 79) where it is literally said: "*In the context of this thesis, the pitfall scanning tool Oops! has been tested in practice by running it on GRO, MaHCO and BioTop. OOOPS! was able to detect different types of modeling mistakes in all three ontologies, which indicates that it is effective in practice.*" The GRO ontology contains more than 500 terms. The MaHCO ontology in its complete version is composed of more than 7,900 terms, while its core defines around 100 terms. Finally, BioTop contains more than 180 terms. This thesis work has been developed in Stuttgart, Germany.

OOPS! has also been used during the ontology assessment process in the context of ontologies for human behaviour recognition, as explained in (Rodríguez et al., 2014b) in order to select ontologies to be reused. In addition, same authors, developed afterwards the *Fuzzy Human Behaviour Ontology* making use of OOPS! to detect and correct pitfalls (Rodríguez et al., 2014a) during the ontology evaluation activity. This ontology consists of 228 classes, 133 object properties and 62 data properties. These works have been carried out in collaboration between the Åbo Akademi Univeristy, in Turku, Finland and the University of Granada, Spain.

Finally, OOPS! has also been used as part of the Ontology Engineering course in the Artificial Intelligence Master at *Universidad Poliécnica de Madrid* (UPM) for evaluating ontologies developed by more than 50 students. It has also been used for ontology evaluation within some enterprises like SemanticArts[68] in U.S.A and inova8[69] in the

[66]The server logs analysed contained access information from March 1st, 2012 to February 17th, 2014 and the Google Analytics one from November 1st, 2014 to 24th August, 2015. The period from February to November 2014 from the server log is lost due to technical problems and changes on the storage and configuration. Therefore, the access and executions reported here are actually less than the real traffic of the website as we could not recover the log for those eight months.

[67]These executions are those registered in the server log plus google Analytics. This log is different from the OOPS! log of executions that gathers ontologies and which results are shown in Section 6.4.

[68]http://semanticarts.com (last visited on the 26th August, 2015)

[69]http://inova8.com (last visited on the 26th August, 2015)

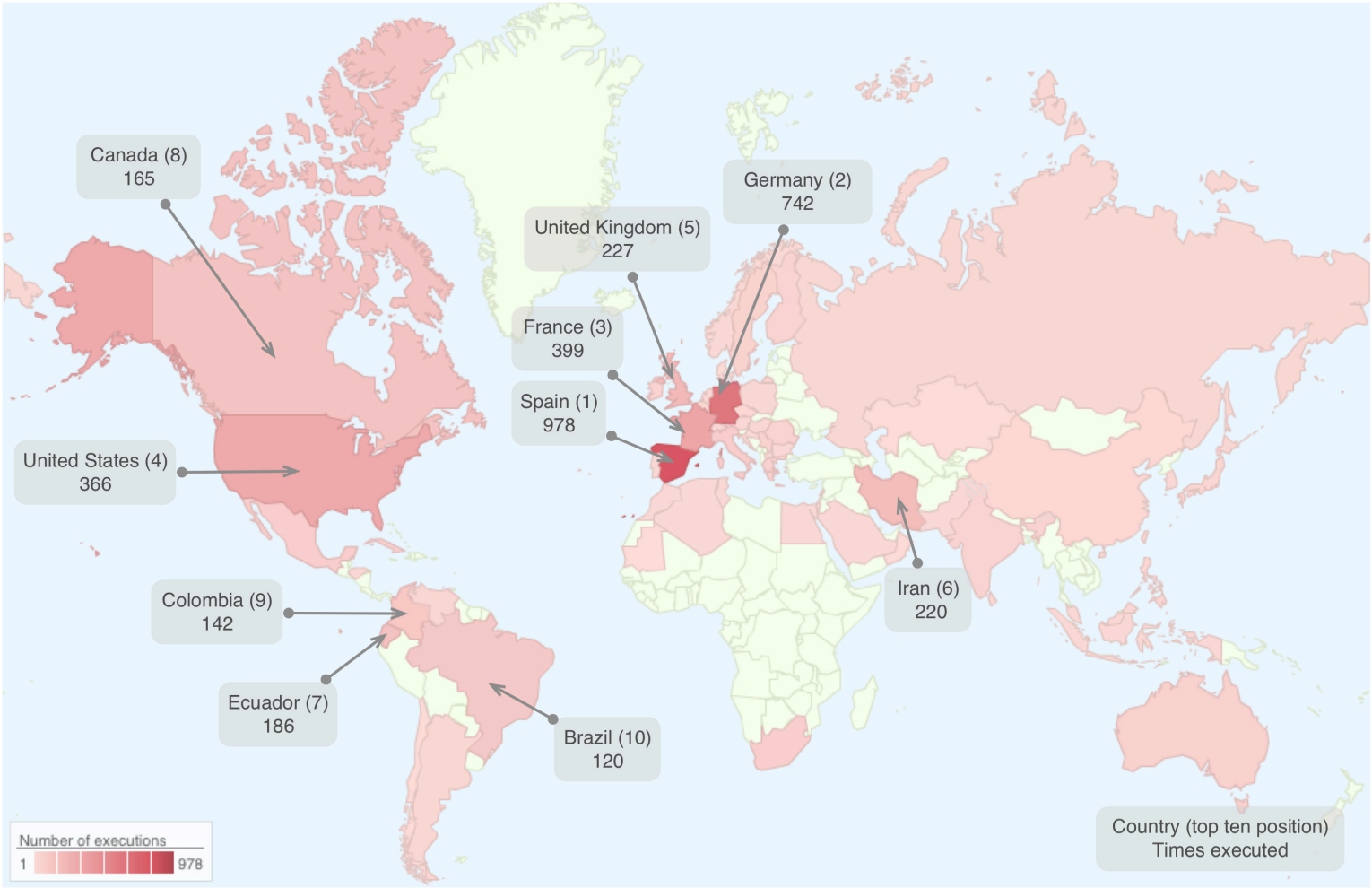


Figure 6.6: Map with the top 10 countries executing OOPS!

United Kingdom. Other companies, like Raytheon,[70] have used OOPS! both for ontology evaluation and for training courses.

6.6.2 Integration in third-party software

Some third-party software, mainly ontology registries and catalogues, already incorporate the ontology evaluation features provided by OOPS!. In total OOPS! have been integrated in six systems. Three of them are developments outside the Ontology Engineering Group (OEG), namely: Linked Open Vocabularies[71] (LOV), Ontohub[72] and DrOntoAPI.[73] The other three systems have been developed by OEG colleagues at UPM, namely: Widoco,[74] the SmartCity ontology catalogue[75] and OnToology.[76] The integration within each system could be done in two ways: (a) by establishing a link from the given system to OOPS!'s interface and (b) by invoking the web service provided by OOPS!.

Linked Open Vocabularies is an ontology index developed by Mondeca,[77] in Paris, as part of the Datalift project.[78] In this case the integration consists in providing for each vocabulary a link to OOPS! website as shown in Figure 6.7. Such link points to a web page that provides OOPS!'s output for the given vocabulary. This integration was carried out by the LOV team. We provided support and help to LOV developers when needed.

Ontohub (Mossakowski et al., 2014) is an open ontology repository, which supports ontology collection, retrieval, development, mapping, translation, among others. Ontohub is developed in collaboration between the University of Magdeburg and the University of Bremen, both in Germany. In this case, for each ontology registered, Ontohub provides to the user the functionality of diagnosing the given ontology with OOPS!, as shown in Figure 6.8. When the user activates the diagnose, the system calls OOPS! web service and render the output within their own web interface. In this

[70] http://www.raytheon.com (last visited on the 26th August, 2015)
[71] http://lov.okfn.org (last visited on the 27th November, 2015)
[72] https://ontohub.org/ (last visited on the 27th November, 2015)
[73] http://sourceforge.net/projects/drontoapi/ (last visited on the 27th November, 2015)
[74] https://github.com/dgarijo/Widoco (last visited on the 27th November, 2015)
[75] http://smartcity.linkeddata.es (last visited on the 27th November, 2015)
[76] http://ontoology.linkeddata.es(last visited on the 27th November, 2015)
[77] http://www.mondeca.com (last visited on the 27th November, 2015)
[78] http://datalift.org/ (last visited on the 27th November, 2015)

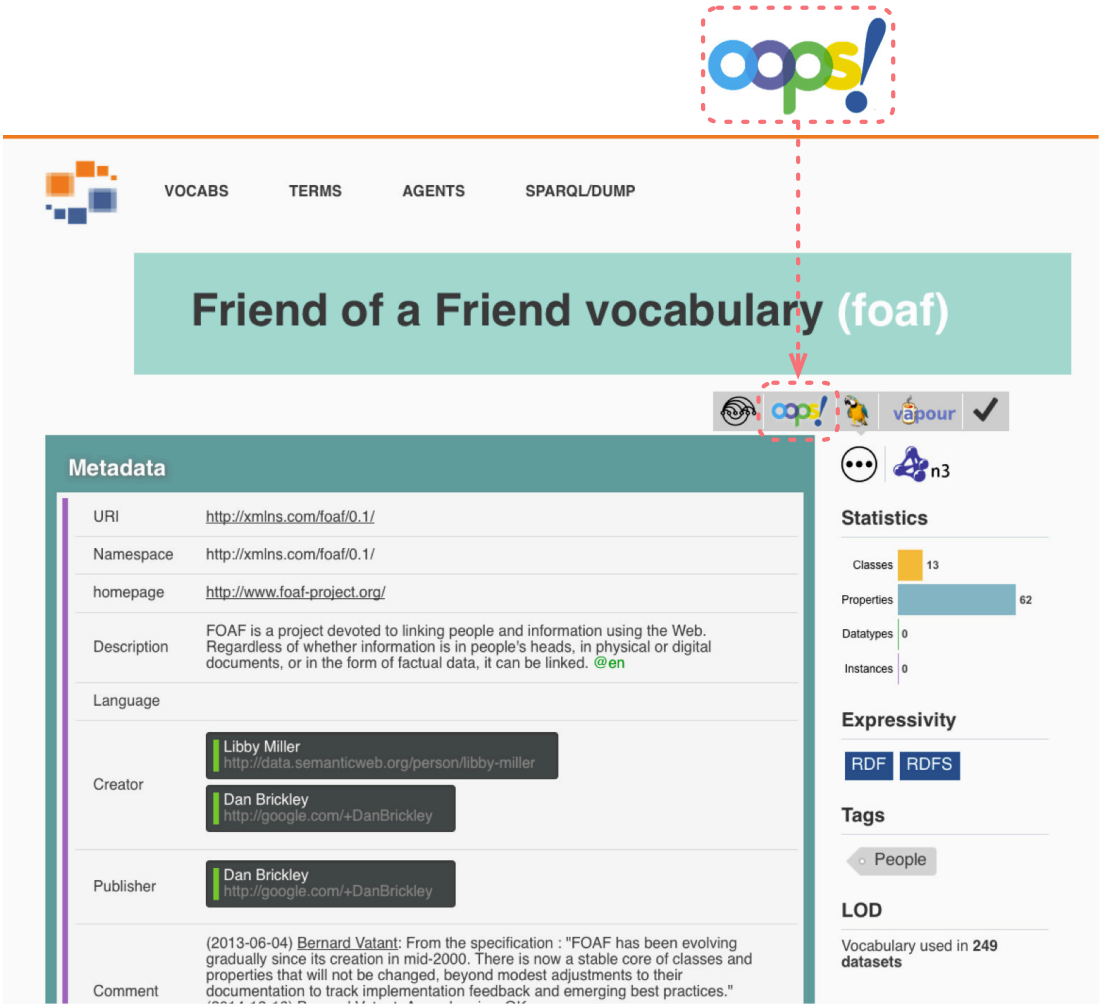


Figure 6.7: OOPS! integration within Linked Open Vocabularies (LOV)

particular case, the system shows for each ontology element a list of pitfalls that affect the given element, as shown in Figure 6.9. This integration was done by the Ontohub developers team in collaboration with OEG. More precisely, Ontohub team proposed a hackathon project during OntologySummit2013 called *Clinics Ontohub OOR OOPS Integration*.[79] This project was awarded with the IAOA prize.[80]

The **DrOntoAPI** is the last example of systems not developed at OEG that integrate OOPS!. It has been developed at Babes-Bolyai University, Romania. This API

[79]See http://goo.gl/GPe2Or for more details about the integration project carried out during the hackathon organized as part of the OntologySummit2013.

[80]http://ontolog.cim3.net/cgi-bin/wiki.pl?OntologySummit2013_Hackathon_Clinics

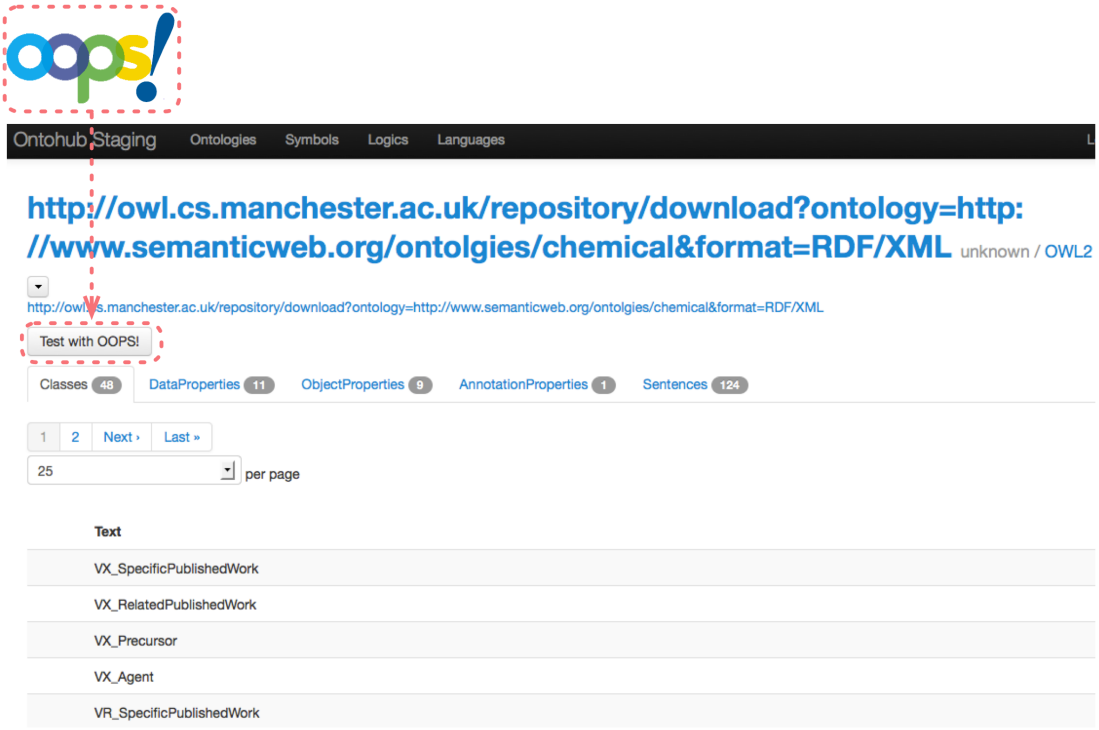


Figure 6.8: OOPS! integration within Ontohub

provides ontology evaluation functionality based on OOPS! web service. Even though it is supposed to gather different evaluation services under a unique JAVA interface, at the moment of writing this thesis, this system provides a JAVA API only for OOPS! web service. This integration has been done by DrOntoAPI developers while OOPS! team provided support by email when requested.

Within the OEG developments, the first collaboration to integrate OOPS! was carried out together with the developers of **Widoco**, a wizard for generating HTML documentation for ontologies. This integration consists in rendering the OOPS! web service output in a HTML document. Once the user has generated the ontology documentation, the system offers the option of generating an ontology evaluation report as shown in Figure 6.10.

In the context of the READY4SmartCities European project (FP7-608711),[81] an ontology catalogue for smart cities and related domains has been developed by OEG

[81] http://www.ready4smartcities.eu (last visited on the 27th November, 2015)

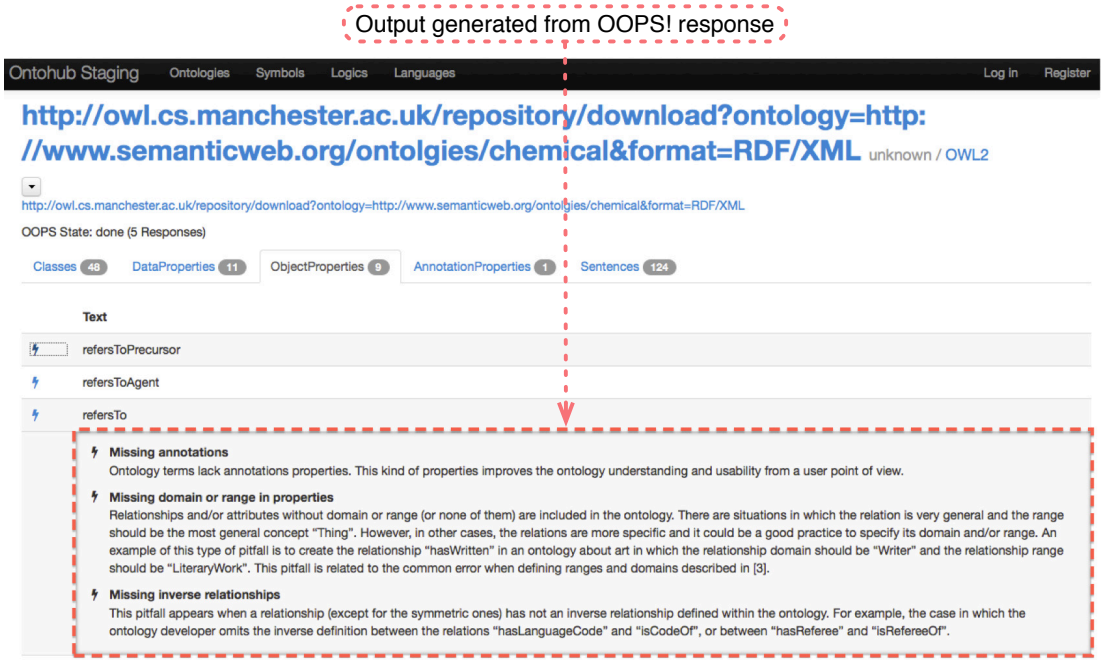


Figure 6.9: Example of pitfalls that affect a given object property in Ontohub interface

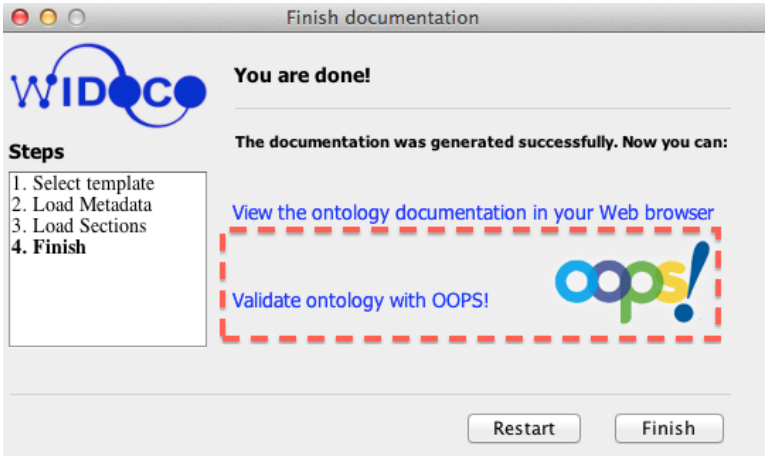


Figure 6.10: OOPS! integration within Widoco

members. As part of its functionality, the **SmartCity ontology catalogue**[82] offers evaluation results for each ontology provided by the OOPS! web service as shown in Figure 6.11. In this case the system includes the evaluation results provided by OOPS!

[82]http://smartcity.linkeddata.es (last visited on the 27th November, 2015)

within the ontology information registered in the catalogue for each ontology.[83]

Last example of OOPS! integration, in collaboration with other OEG members, is the case of **OnToology**,[84] a tool for developing ontologies collaboratively using Github, where ontology developers keep track of their ontology projects. OntToology addresses the integration of several ontology development activities such as ontology documentation, the generation of diagrams and ontology diagnosis. OnToology detects new versions of ontology documents in Github repositories and it generates the documentation, diagrams and evaluation reports. Once the evaluation of the new ontology version is done, OnToology raises an issue on Github containing a summary of the evaluation results and a preview of such results in HTML. Figure 6.12 shows an example of an issue opened in a Github repository summarizing an ontology evaluation report.

6.7 Summary

This chapter has shown the use adoption of the system world wide, including use statistics, and examples of integration of the system within third-party software. **This chapter clearly shows that OOPS! has been widely adopted and accepted by the semantic web community, including researchers and ontology engineers.**

A conceptual comparison between existing tools for ontology diagnosis and OOPS! have been presented, showing that OOPS! addresses the detection of a broader number of pitfalls than the other tools. The tools' software characteristics have also been compared. It can be stated that OOPS! outperforms the current state of the ontology evaluation tools addressing the detection of 33 pitfalls.

In the course of the chapter evidence of wide adoption of OOPS! through the Semantic Web community has been provided. OOPS! has been used in more than 60 countries, has been executed more than 2,700 times over 969 different ontologies. It has been integrated within three systems developed outside the OEG and within three systems developed within the OEG. In addition, OOPS! is being used in PhD thesis and within enterprises during their ontology development projects as well as during training activities.

[83] Example taken from http://smartcity.linkeddata.es/ontologies/personal.us.esaparedesTrade.owl.html (last visited on the 27th November, 2015)

[84] http://ontoology.linkeddata.es (last visited on the 27th November, 2015)

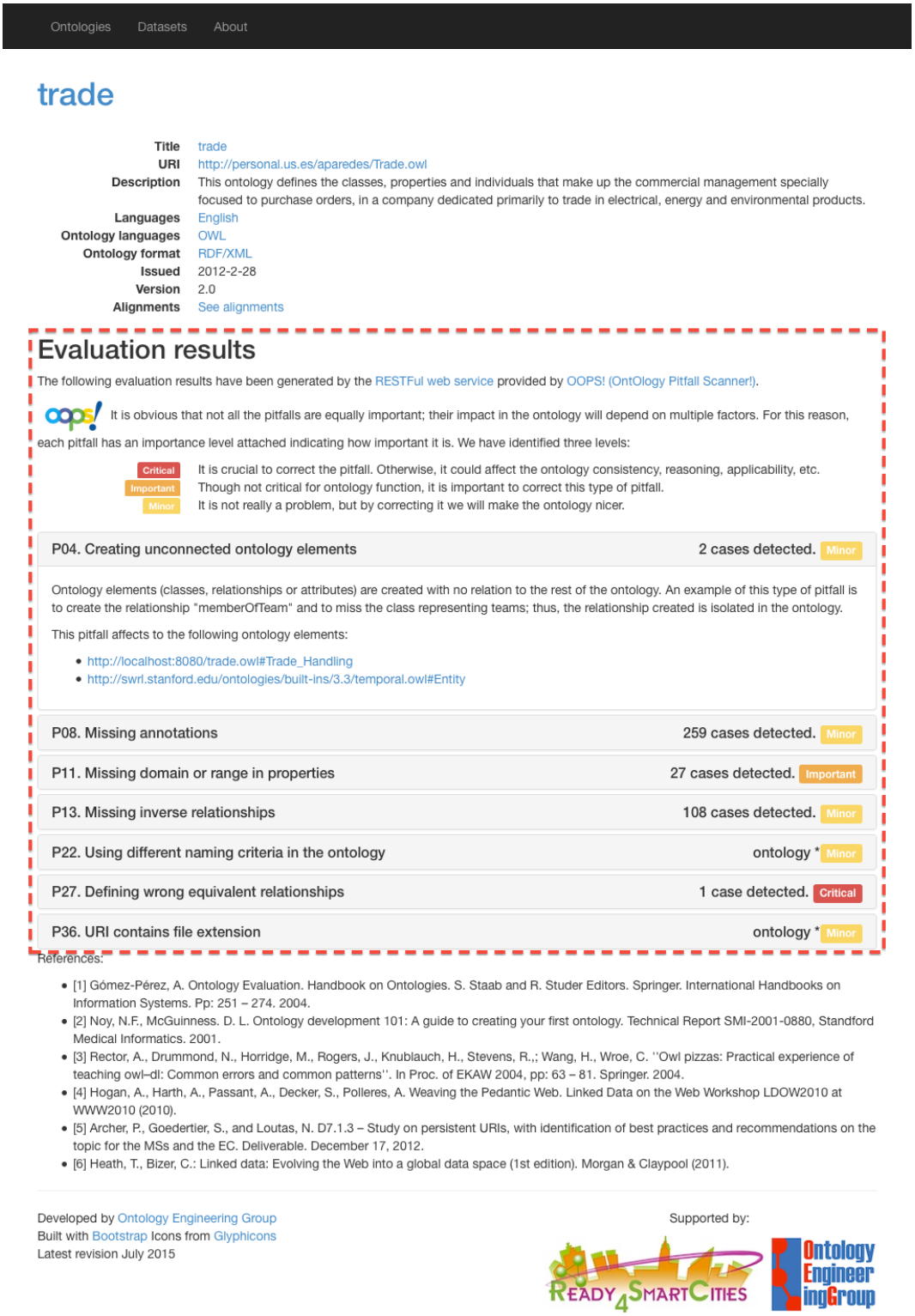


Figure 6.11: OOPS! integration within a SmartCity ontology catalogue available at http://smartcity.linkeddata.es

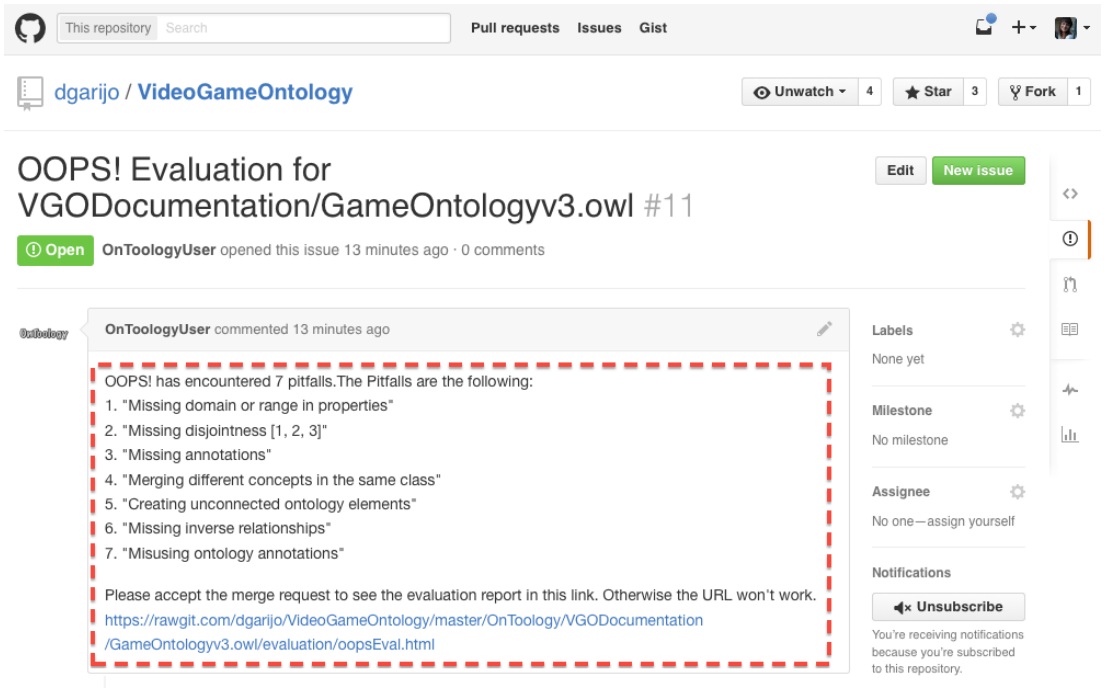


Figure 6.12: Screenshot of a Github issue generated by OnToology summarizing the ontology report

Finally, the validation of OOPS! in terms of users' satisfaction has proved that the system effectively helped them to diagnose their ontologies. In this sense, users can improve their ontologies quality carrying out the ontology repair activity triggered by the diagnosis results. Users claimed that such results are clear (40%) or very clear (47%) and that OOPS! has been useful for them (80%).

Chapter 7

Conclusions and future work

Every ontology development project should involve the ontology evaluation activity, including ontology diagnosis and repair, to check the technical quality of the ontology under development. Due to its importance, numerous research contributions have emerged in this field. However, this activity is still largely neglected by ontology developers as there is an important gap between the methodological approaches for ontology evaluation, how to apply such approaches and the tools that support them. In addition, not many initiatives provide clear guidance about how to diagnose ontologies and how to repair them accordingly.

In this thesis we have provided ontology engineers with support to evaluate their ontologies, and more precisely, to assist them during the ontology diagnosis activity, lessening thus the effort required during such activity. To achieve this goal, we have pursued the following objectives: (O1) to help ontology engineers to diagnose their ontologies in order to find common pitfalls and (O2) to ease the ontology diagnosis by providing suitable technological support.

This chapter presents the conclusions of this work and summarizes the main contributions provided to solve the relevant open research problems identified in Chapter 2. Then, we discuss the verification of our hypothesis, and finally we give an outlook of the future lines of work.

7.1 Review of main contributions

This section presents the main contributions of this thesis and their relation to the open research problems identified in the state of the art.

The first of these open research problems refers to the fact that most of the methods for ontology evaluation (i) only deal with taxonomical knowledge; (ii) address a narrow range of ontology evaluation aspects; or (iii) provide a set of measurements but no concrete ontology diagnosis output are offered. This is addressed by the first contribution of this thesis, which consists in **a pitfall catalogue (C1)**. This catalogue describes a total of 41 pitfalls that developers might fall into, in terms of several aspects such as: modelling decisions, usability issues, metadata problems, and Linked Data specific mistakes, among others. The catalogue of pitfalls is not only useful for ontology diagnosis purposes but also for ontology conceptualization, since wrong modelling decisions are pointed out and correct alternatives are offered as suggestions in the pitfall descriptions. For the description of the pitfalls in this catalogue, a general template has been proposed, which includes: general description of pitfalls, particular examples, tips about how the pitfall could be solved and references to previous works, among other fields. This contribution also includes a **classification of pitfalls in terms of ontology evaluation dimensions and aspects**. Such classification allows ontology evaluators to focus on those of the given dimensions that they might be interested in evaluating. This contribution compiles and extends the common errors identified in the ontology evaluation field.

Pitfalls have also been **classified according to their importance level** in regard to the possible negative consequences that each pitfall might cause to the ontology. The indications provided by the importance levels allow users to address in first place those pitfalls that might have worse consequences in contrast to the lists of common errors that do not establish any priority system. In addition, this contribution does not reflect a particular author's view. Instead, the importance levels have been assigned following a community-based approach by taking inputs from 54 ontology experts, ontology practitioners and OOPS! users.

We have also provided a **quality model for ontology diagnosis (C2)** to annotate ontology evaluation results according to the QMO and EVAL models. Such quality model not only provides a guide on how to evaluate ontologies, but also a reference for

researchers and practitioners for producing consistent, easily integrated and comparable ontology evaluations. In this sense, this contribution could be used by recommendation systems to rank a set of ontologies according to evaluation results, user criteria and preferences. To the best of the author's knowledge, there are no other open implementations of quality models for ontology diagnose in OWL available online. However, this implementation has not been used in actual evaluation yet. It is planned to be included into OOPS! evaluation results.

The second open research problem identified in the state of the art is the lack of systems that (semi)automatically diagnose ontologies by addressing several aspects and pointing to specific problems to be corrected. This situation might lead to ontologies evaluated superficially and ontology engineers spending too much time and resources in ontology diagnosis. In this thesis we have **designed and implemented 48 methods to (semi)automatically diagnose (C3)** 33 out of the 41 common pitfalls included in the catalogue. These detection methods are **classified according to the type of technique used during their implementations**, namely, structural pattern search, linguistic analysis and specific characteristic search. The design of the methods provides specific guides to users in order to search for common pitfalls, and their implementation lightens the tedious task of manually exploring the ontologies. This contribution reduces the effort needed from ontology engineers to diagnose ontologies. It also represents a step forward in the number of detection methods implemented and the ontology evaluation aspects covered.

In order to bridge the existing gap between ontology developers, and specially newcomers to the field, and the ontology evaluation activity, we provide the last contribution of this thesis, consisting in the **technological support OOPS! (C4)**, which allows users to diagnose their ontologies according to the designed and implemented detection methods. OOPS! has been proved to be useful for final users, as illustrated in Chapter 6. It was also shown that our approach brings ontology developers to start evaluating ontologies in a (semi)automatic way. It is worth noting that only 11 out of the 30 users providing feedback forms claimed to have previously used any other system to evaluate their ontologies.

The potential of having the implemented detection methods working jointly with other Ontological Engineering tools and systems lead us to also provide a web service

offering access to the ontology diagnosis methods. This service allows third-party software such as ontology registries and editors to make use of the common pitfall detection methods shown in Chapter 5. Indeed, it has been integrated in a number of tools as detailed in Chapter 6.

As presented in Chapter 6, it can be stated that the approach developed and presented throughout this thesis has been widely accepted by the Semantic Web community and experts in other areas. The impact of our approach is supported by the following facts:

- OOPS! has been broadly accepted by a high number of users worldwide and has been executed around 3000 times from 60 different countries.
- It has been continuously used from very different geographical locations.
- It is integrated with third-party software.
- It is locally installed in private enterprises and used both for ontology development activities and for training courses.
- It has been used in third-parties ontology developments and studies being reported in scientific publications.

Therefore, it could be stated that the approach proposed in this work represents a valuable contribution to the field, since it has become useful for ontology practitioners and for newcomers for different use cases. In this thesis we have shown that both the catalogue and OOPS! are maintained and evolve according to users' feedback and research results. It has also been presented how the catalogue and OOPS! contribute to the state of the art in ontology diagnosis by extending the number of common errors described and the detection methods automated until now.

7.2 Hypothesis verification

We have verified the hypotheses of thesis by different means. In the following, we introduce each of these hypotheses and elaborate on how the work presented in this thesis led to its verification.

H1: *Systematic approaches to evaluate ontologies based on a list of common pitfalls improve the quality of ontologies.*

This thesis provides a systematic approach for ontology diagnosis based on a list of common pitfalls that also includes preliminary ontology repair guidelines. As defined in the NeOn Methodology (Suárez-Figueroa et al., 2015), ontology repair refers to the activity of solving errors in the ontology and it is triggered by the ontology diagnosis activity. Therefore, by detecting the pitfalls described in the catalogue and repairing the ontology accordingly, the developer is able to improve the quality of a given ontology.

As shown in Section 6.4, a significant number of ontologies have been submitted to OOPS! more than once (we found 1.680 duplicated ontologies). When analysing OOPS! 's logs, we observed that in cases in which an ontology is submitted more than once, usually more pitfalls are detected during the first diagnose than in subsequent submissions. This fact also contributes to the verification of H1 even though further analysis of ontologies being submitted several times to OOPS! is planned for future work.

Regarding whether the list of pitfalls described in the catalogue are actually useful for ontology diagnosis and therefore ontology repair, we relay on the results of the survey conducted on the list of pitfalls in order to assign them importance levels (see Section 4.3). In this sense, we observe that most of the pitfalls are considered useful for the ontology diagnosis activity by ontology experts. Only a few pitfalls have been marked as irrelevant and, in such cases, with less than 10 votes (among 54 participants).

H2: *A collection of methods for detecting pitfalls in a (semi)automatic way facilitates the ontology diagnosis activity.*

The list of designed and implemented detection methods (see Section 5.3) together with the positive results obtained from the users' satisfaction survey, and the engagement with the system OOPS! that the community showed (see Section 6.5 and Section 6.6) support H2.

It is worth noting that there are still eight remaining pitfalls for which no detection methods have been implemented. This fact supports the claim by Gangemi and colleagues already mentioned in Section 5.3 that reads: *"This seems to imply that no automatized method will ever suffice and that intellectual judgement will always be needed.*

However, automatic and semi-automatic techniques can be applied that make evaluation easier, less subjective, more complete and faster." (taken literarily from (Gangemi et al., 2006) page 6). Identification, design and implementation of new methods and techniques are open lines to continue with research work presented in this thesis.

7.3 Future work

For the current and **future lines of work regarding the pitfall catalogue** proposed in this thesis, we would like to present the following lines of research and development:

- We pretend to keep a living catalogue in which not only the catalogue maintainers but also users will include pitfalls. In a first step, we plan **to include pitfalls related to multilingualism, metadata and annotation aspects** related to the patterns[85] presented in (Labra-Gayo et al., 2015).

- Another line of work related to the catalogue update is **to adapt the current catalogue to OWL 2**. This would involve a detailed analysis of both the catalogue and the OWL 2 language to detect those pitfalls that should be updated or extended to consider OWL 2 features. A further analysis would also point to new pitfalls that might occur in OWL 2 due to OWL 2 primitives. For example, it should be considered whether the `owl:propertyDisjointWith` construct could led to new pitfalls.

- During the development of this thesis the pitfalls have been classified by different criteria and some pitfalls include in their descriptions references to other pitfalls. However, the catalogue might also be seen mostly as a plain list. We plan **to analyse and establish relationships between pitfalls**. For example, hierarchical relationships between pitfalls could be stablished, as a common mistake could be an specialization of another error.

- Following the analysis presented in (Poveda-Villalón et al., 2013), **providing a catalogue of good practices** would be very useful in order to assess an ontology's quality. Once the catalogues of common pitfalls and good practices are

[85] http://www.weso.es/MLODPatterns/catalog.html (last visited on the 23rd September, 2015)

developed, it could be analysed whether the lack of a given good practice implies the occurrence of an error or, if otherwise, it is just neutral in terms of quality.

Furthermore, we intend to extend and improve the technological support proposed in this thesis. **Future lines of development and innovation regarding OOPS!** are:

- Immediate future work will concentrate on the **automation of the remaining pitfalls and the enhancement of some of the already implemented methods**. These extensions might require increasing the users' interaction with the system by keeping them on the loop and using natural language processing techniques as proposed in (Suárez-Figueroa et al., 2013b).

- Another future line of work would be **to incorporate guidelines for ontology repair into OOPS!**. This feature would also require users' interaction with the system. The system should be able to provide guidance and relevant information in order to offer the user choices for repairing or improving the ontology.

- Focusing on the Linked Data scenario, in which a vast amount of data is annotated by making use of ontologies, an immediate line of work is **to consider such data during the evaluation with the purpose of enhancing the results.** As a first step, mismatches between the defined model and the instantiated data could be detected as well as inconsistencies. An integration of this future work with Loupe (Mihindukulasooriya et al., 2015), an online system for inspecting datasets, is also planned. This enhancement can also be applied to local knowledge bases.

- In addition, and also related to the Linked Data scenario, future work would involve **making OOPS! scalable for ontologies that contain a high number of terms**. At the moment of writing this thesis, the system leads to a high response time for big ontologies, as for example the DBpedia ontology, being the main bottleneck the number of object properties defined in them.

- To continue with the evaluation of the catalogue of pitfalls and OOPS! we aim **to perform quantitative assessments** as part of the future work in order to measure the time and effort saved by developers in contrast with (a) the lack of a pitfall catalogue or (b) the lack of an automated system to detect pitfalls.

However, given the scale of the current web experiment, we have opted for a continuous cycle of quantitative assessment to progressively refine the system, the catalogue and the detection methods.

- Regarding current technological drawbacks or possible improvements of OOPS!, we can mention the following enhancements to be implemented in future versions of the tool: (a) **to include other vocabularies** for the pitfalls related to **natural language annotations** and let the user choose or propose annotation properties; (b) **to generate diagrams** representing the ontology structure and indicating the elements involved in the detected pitfalls; (c) **to update the web interface** to newer web technologies and increase user experience; (d) **to allow reasoning features** so that the user can choose between analysing the inferred statements as well or only the explicit knowledge; (e) **to adapt the web service's response** to the quality model presented in Chapter 4.

- For those detection **methods based on structural pattern** search, we propose to analyse whether they can be **translated into SPARQL queries** and to which extent this approach improves the system performance. This analysis would align and combine efforts with the approach presented in (Roussey et al., 2012).

- More ambitious plans include **providing a formal language for defining pitfalls and the mechanisms to interpret and process the pitfall definitions without manual encoding**. This feature would represent a major breakthrough, as the manual encoding of the detection methods is the main bottleneck in the scalability of the system.

- Finally, the **integration of OOPS! within existing ontology editors**, such as Protégé, WebProtégé and the NeOn Toolkit, would substantially benefit users, since they would not need to change platforms in order to repair their ontologies after the diagnosis activity.

This thesis has advanced the state of the art in ontology evaluation by helping developers to diagnose and consequently repair their ontologies, increasing in this way their quality. Even though human intervention is still needed to completely evaluate ontologies, our work has significantly contributed to the automation of this activity.

Bibliography

Aguado-De Cea, G., Montiel-Ponsoda, E., Poveda-Villalón, M., and Giraldo-Pasmin, O. X. (2015). Lexicalizing Ontologies: The issues behind the labels. In *Multimodal communication in the 21st century: Professional and academic challenges. 33rd Conference of the Spanish Association of Applied Linguistics (AESLA), XXXIII AESLA conferences*. Elsevier. 53

Archer, P., Goedertier, S., and Loutas, N. (2012). D7.1.3–study on persistent URIs, with identification of best practices and recommendations on the topic for the MSs and the EC. *Interoperability Solutions for European Public Administrations*. 54

Arpírez, J. C., Corcho, Ó., Fernández-López, M., and Gómez-Pérez, A. (2003). WEBODE in a nutshell. *AI Magazine*, 24(3):37–48. 3, 36

Atemezing, G., Corcho, O., Garijo, D., Mora, J., Poveda-Villalón, M., Rozas, P., Vila-Suero, D., and Villazón-Terrazas, B. (2013). Transforming meteorological data into linked data. *Semantic Web*, 4(3):285–290. 54

Auer, S. (2006). The rapidowl methodology–towards agile knowledge engineering. In *15th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE 2006), 26-28 June 2006, Manchester, United Kingdom*, pages 352–357. IEEE Computer Society. xv, 26, 27

Barron, F. H. and Barrett, B. E. (1996). Decision quality using ranked attribute weights. *Management Science*, 42(11):1515–1523. 110

Beißwanger, A. E. (2013). Developing ontological background knowledge for biomedicine. 197

Berrueta, D., Fernández, S., and Frade, I. (2008). Cooking HTTP content negotiation with Vapour. In Bizer, C., Auer, S., Grimnes, G. A., and Heath, T., editors, *Worshop on Scripting for the Semantic Web 2008*, number 368 in CEUR Workshop Proceedings. 54

Brank, J., Grobelnik, M., and Mladenić, D. (2005). A survey of ontology evaluation techniques. In *Proceedings of 8th Int. multi-conf. Information Society*, pages 166–169. 28

Burton-Jones, A., Storey, V. C., Sugumaran, V., and Ahluwalia, P. (2005). A semiotic metrics suite for assessing the quality of ontologies. *Data & Knowledge Engineering*, 55(1):84–102. 2, 31

Corcho, O., Gómez-Pérez, A., González-Cabero, R., and Suárez-Figueroa, M. C. (2004). ODEval: a tool for evaluating RDF (S), DAML+ OIL, and OWL concept taxonomies. In Bramer, M. and Devedzic, V., editors, *Artificial Intelligence Applications and Innovations, IFIP 18th World Computer Congress, TC12 First International Conference on Artificial Intelligence Applications and Innovations (AIAI-2004), 22-27 August 2004, Toulouse, France*, volume 154 of *IFIP*, pages 369–382. Kluwer/Springer. 3, 37

Corcho, O., Roussey, C., Blázquez, L. M. V., and Pérez, I. (2009). Pattern-based OWL ontology debugging guidelines. In Blomqvist, E., Sandkuhl, K., Scharffe, F., and Svátek, V., editors, *Proceedings of the Workshop on Ontology Patterns (WOP 2009) , collocated with the 8th International Semantic Web Conference (ISWC-2009), Washington D.C., USA, 25 October, 2009.*, volume 516 of *CEUR Workshop Proceedings*. CEUR-WS.org. 33, 45

Daga, E., Blomqvist, E., Gangemi, A., Montiel-Ponsoda, E., Nikitina, N., Presutti, V., and Villazón-Terrazas, B. (2010). D2.5.2 Pattern based ontology design: methodology and software support. Technical report, National Research Council (CNR). NeOn Project. http://www. neon-project. org. 3, 41

Djedidi, R. and Aufaure, M. (2010). *ONTO-EVO*^{a}*L* an ontology evolution approach guided by pattern modeling and quality evaluation. In Link, S. and Prade, H., editors, *Foundations of Information and Knowledge Systems, 6th International Symposium,*

FoIKS 2010, Sofia, Bulgaria, February 15-19, 2010. Proceedings, volume 5956 of *Lecture Notes in Computer Science*, pages 286–305. Springer. 2

Duque-Ramos, A., Fernández-Breis, J. T., Stevens, R., and Aussenac-Gilles, N. (2011). OQuaRE: A SQuaRE-based approach for evaluating the quality of ontologies. *Journal of Research and Practice in Information Technology*, 43(2):159–176. 34, 42, 45

Fernández-López, M. and Gómez-Pérez, A. (2002). The integration of OntoClean in WebODE. In *Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02) Evaluation of Ontology-based Tools Workshop (EON2002)*, volume 62 of *CEUR Workshop Proceedings*, pages 38–52. CEUR-WS.org. xv, 2, 36, 37

Fernández-López, M., Gómez-Pérez, A., and Juristo, N. (1997). METHONTOLOGY: from ontological art towards ontological engineering. In *Proceedings of the Ontological Engineering AAAI97 Spring Symposium Series*. American Asociation for Artificial Intelligence. 1, 15, 18, 53

Fernández-López, M., Gómez-Pérez, A., Pazos-Sierra, J., and Pazos-Sierra, A. (1999). Building a chemical ontology using methontology and the ontology design environment. *IEEE Intelligent Systems*, 14(1):37–46. 1, 18, 53

Gangemi, A., Catenacci, C., Ciaramita, M., and Lehmann, J. (2006). Modelling ontology evaluation and validation. In Sure, Y. and Domingue, J., editors, *The Semantic Web: Research and Applications, 3rd European Semantic Web Conference, ESWC 2006, Budva, Montenegro, June 11-14, 2006, Proceedings*, volume 4011 of *Lecture Notes in Computer Science*, pages 140–154. Springer. 2, 31, 53, 57, 115, 116, 132, 212

García-Ramos, S., Otero, A., and Fernández-López, M. (2009). Ontologytest: A tool to evaluate ontologies through tests defined by the user. In Omatu, S., Rocha, M., Bravo, J., Riverola, F. F., Corchado, E., Bustillo, A., and Corchado, J. M., editors, *Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living, 10th International Work-Conference on Artificial Neural Networks, IWANN 2009 Workshops, Salamanca, Spain, June 10-12, 2009. Proceedings, Part II*, volume 5518 of *Lecture Notes in Computer Science*, pages 91–98. Springer. 35

Gómez-Pérez, A. (1995). Some ideas and examples to evaluate ontologies. In *Proceedings of the Eleventh Conference on Artificial Intelligence for Applications*, pages 299–305. IEEE Computer Society Press. 29, 43

Gómez-Pérez, A. (1996). Towards a framework to verify knowledge sharing technology. *Expert Systems with Applications*, 11(4):519–529. 2

Gómez-Pérez, A. (1999). Evaluation of taxonomic knowledge in ontologies and knowledge bases. In *Banff Knowledge Acquisition for Knowledge-Based Systems, KAW'99*, volume 2, pages 6.1.1–6.1.18, Banff, Alberta, Canada. University of Calgary, Alberta, Canada. 2

Gómez-Pérez, A. (2004). Ontology evaluation. In Staab, S. and Studer, R., editors, *Handbook on ontologies*, International Handbooks on Information Systems, pages 251–274. Springer. 2, 53, 57, 228

Gómez-Pérez, A., Corcho, O., and Fernández-López, M. (2004). *Ontological Engineering: with examples from the areas of Knowledge Management, e-Commerce and the Semantic Web*. Advanced Information and Knowledge Processing. Springer. xv, 1, 19, 20, 37

Gómez-Pérez, A., Juristo, N., and Pazos, J. (1995). Evaluation and assessment of knowledge sharing technology. In Mars, N. J., editor, *Towards Very Large Knowledge Bases*, pages 289–296. IOS Press. 2, 14

Grüninger, M. and Fox, M. S. (1995). Methodology for the design and evaluation of ontologies. In *IJCAI'95, Workshop on Basic Ontological Issues in Knowledge Sharing*. xv, 1, 2, 17

Guarino, N. (2004). Toward a formal evaluation of ontology quality. *IEEE intelligent Systems*, 19(4):78–79. 2

Guarino, N. and Welty, C. A. (2009). An overview of OntoClean. In Staab, S. and Studer, R., editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 201–220. Springer. 2, 29, 45

Haase, P., Brockmans, S., Palma, R., Euzenat, J., and d'Aquin, M. (2009). D1.1.2 updated version of the networked ontology model. Technical report, Universität Karlsruhe. NeOn Project. http://www. neon-project. org. xvi, 60, 62, 63, 64, 65, 67, 135

Heath, T. and Bizer, C. (2011). *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 1st edition. 53

Hogan, A., Harth, A., Passant, A., Decker, S., and Polleres, A. (2010). Weaving the pedantic web. In Bizer, C., Heath, T., Berners-Lee, T., and Hausenblas, M., editors, *Proceedings of the WWW2010 Workshop on Linked Data on the Web, LDOW 2010, Raleigh, USA, April 27, 2010*, volume 628 of *CEUR Workshop Proceedings*. CEUR-WS.org. 54

Hristozova, M. and Sterling, L. (2002). An extreme method for developing lightweight ontologies. In *Workshop on Ontologies in Agent Systems, 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems*. 25

IEEE (1996). IEEE Standard for Developing Software Life Cycle Processes. *IEEE Std 1074-1995*, pages i–. 18

ISO (2001). ISO/IEC 9126-1:2001, Software engineering – Product quality – Part 1: Quality model. Technical report, International Organization for Standardization.

ISO (2011a). ISO/IEC 25010:2011, Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. Technical report, International Organization for Standardization. 34, 120

ISO (2011b). ISO/IEC 25040:2011, Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Evaluation process. Technical report, International Organization for Standardization. 14, 16

Ji, Q., Haase, P., Qi, G., Hitzler, P., and Stadtmüller, S. (2009). RaDON - Repair and Diagnosis in Ontology Networks. In Aroyo, L., Traverso, P., Ciravegna, F., Cimiano, P., Heath, T., Hyvönen, E., Mizoguchi, R., Oren, E., Sabou, M., and Simperl, E. P. B., editors, *The Semantic Web: Research and Applications, 6th European Semantic Web*

Conference, ESWC 2009, Heraklion, Crete, Greece, May 31-June 4, 2009, Proceedings, volume 5554 of *Lecture Notes in Computer Science*, pages 863–867. Springer. 36

Keet, C. M., Suárez-Figueroa, M. C., and Poveda-Villalón, M. (2013). The current landscape of pitfalls in ontologies. In Filipe, J. and Dietz, J. L. G., editors, *KEOD 2013 - Proceedings of the International Conference on Knowledge Engineering and Ontology Development, Vilamoura, Algarve, Portugal, 19-22 September, 2013*, pages 132–139. SciTePress. 189

Labra-Gayo, J. E., Kontokostas, D., and Auer, S. (2015). Multilingual linked data patterns. *Semantic Web Journal*, 6(4):319–337. 212

Mader, C., Haslhofer, B., and Isaac, A. (2012). Finding Quality Issues in SKOS Vocabularies. In Zaphiris, P., Buchanan, G., Rasmussen, E., and Loizides, F., editors, *Theory and Practice of Digital Libraries - Second International Conference, TPDL 2012, Paphos, Cyprus, September 23-27, 2012. Proceedings*, volume 7489 of *Lecture Notes in Computer Science*, pages 222–233. Springer. 35

Miettinen, K. (1999). Nonlinear multiobjective optimization. 110

Mihindukulasooriya, N., Poveda-Villalón, M., García-Castro, R., and Gómez-Pérez, A. (2015). Loupe - An Online Tool for Inspecting Datasets in the Linked Data Cloud. In Villata, S., Pan, J. Z., and Dragoni, M., editors, *Proceedings of the ISWC 2015 Posters & Demonstrations Track co-located with the 14th International Semantic Web Conference (ISWC-2015), Bethlehem, PA, USA, October 11, 2015.*, volume 1486 of *CEUR Workshop Proceedings*. CEUR-WS.org. 213

Miller, G. and Fellbaum, C. (1998). WordNet: An electronic lexical database. 133

Montiel-Ponsoda, E., Vila Suero, D., Villazón-Terrazas, B., Dunsire, G., Escolano Rodríguez, E., and Gómez-Pérez, A. (2011). Style guidelines for naming and labeling ontologies in the multilingual web. In *Proceedings of International Conference on Dublin Core and Metadata Applications*. Informatica. 53

Mossakowski, T., Kutz, O., and Codescu, M. (2014). Ontohub: A semantic repository for heterogeneous ontologies. In *Proc. of the Theory Day in Computer Science (DACS-2014), Satellite workshop of ICTAC-2014, University of Bucharest.* 199

Noy, N. F. and McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology. 24, 53, 57

Pammer, V. (2010). *Automatic support for ontology evaluation: review of entailed statements and assertional effects for OWL ontologies.* PhD thesis, Graz University of Technology, Faculty of Computer Science. 39, 180

Pinto, H. S., Staab, S., and Tempich, C. (2004). DILIGENT: Towards a fine-grained methodology for DIstributed, Loosely-controlled and evolvInG Engineering of oNTologies. In de Mántaras, R. L. and Saitta, L., editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 393–397. IOS Press. xv, 1, 20, 22

Poveda-Villalón, M., Gómez-Pérez, A., and Suárez-Figueroa, M. C. (2014). OOPS! (OntOlogy Pitfall Scanner!): An On-line Tool for Ontology Evaluation. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 10(2):7–34. xvi, 57, 117, 118

Poveda-Villalón, M., Suárez-Figueroa, M. C., and Gómez-Pérez, A. (2009). Common pitfalls in ontology development. In Meseguer, P., Mandow, L., and Gasca, R. M., editors, *Current Topics in Artificial Intelligence, 13th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2009, Seville, Spain, November 9-13, 2009. Selected Papers*, volume 5988 of *Lecture Notes in Computer Science*, pages 91–100. Springer. 55

Poveda-Villalón, M., Suárez-Figueroa, M. C., and Gómez-Pérez, A. (2010). A double classification of common pitfalls in ontologies. In Cimiano, P. and Pinto, H. S., editors, *Workshop on Ontology Quality at the 17th International Conference on Knowledge Engineering and Knowledge Management*, volume 6317 of *Lecture Notes in Computer Science*, pages 1–12. Springer. 57, 115, 116

Poveda-Villalón, M., Suárez-Figueroa, M. C., and Gómez-Pérez, A. (2012). Validating ontologies with OOPS! In ten Teije, A., Völker, J., Handschuh, S., Stuckenschmidt, H., d'Aquin, M., Nikolov, A., Aussenac-Gilles, N., and Hernandez, N., editors, *Knowledge Engineering and Knowledge Management - 18th International Con-*

ference, EKAW 2012, Galway City, Ireland, October 8-12, 2012. Proceedings, volume 7603 of *Lecture Notes in Computer Science*, pages 267–281. Springer. 57

Poveda-Villalón, M., Vatant, B., Suárez-Figueroa, M. C., and Gómez-Pérez, A. (2013). Detecting good practices and pitfalls when publishing vocabularies on the web. In Gangemi, A., Gruninger, M., Hammar, K., Lefort, L., Presutti, V., and Scherp, A., editors, *Proceedings of the 4th Workshop on Ontology and Semantic Web Patterns co-located with 12th International Semantic Web Conference (ISWC 2013), Sydney, Australia, October 21, 2013.*, volume 1188 of *CEUR Workshop Proceedings*. CEUR-WS.org. 212

Presutti, V., Blomqvist, E., Daga, E., and Gangemi, A. (2012). Pattern-Based Ontology Design. In Suárez-Figueroa, M. d. C., Gómez-Pérez, A., Motta, E., and Gangemi, A., editors, *Ontology Engineering in a Networked World.*, pages 35–64. Springer. xv, 27, 28, 41

Presutti, V., Gangemi, A., David, S., Aguado de Cea, G., Suárez-Figueroa, M. C., Montiel-Ponsoda, E., and Poveda-Villalón, M. (2008). D2.5.1 A Library of Ontology Design Patterns: reusable solutions for collaborative design of networked ontologies. Technical report, National Research Council (CNR). NeOn Project. http://www.neon-project. org. 2, 55

Radulovic, F. (2016). *RIDER - A Recommendation Framework for Exploiting Evaluation Results and User Quality Requirements.* PhD thesis, Universidad Politécnica de Madrid, Spain. (To appear). xvi, 119, 120, 121

Radulovic, F., García-Castro, R., and Gómez-Pérez, A. (2013). A Recommendation Framework Based on the Analytic Network Process and its Application in the Semantic Technology Domain. In *Proceedings of The 15th Conference of the Spanish Association for Artificial Intelligence (CAEPIA2013), Doctoral Consortium. Madrid, Spain.* 120

Radulovic, F., García-Castro, R., and Gómez-Pérez, A. (2015). SemQuaRE - An extension of the SQuaRE quality model for the evaluation of semantic technologies. *Computer Standards & Interfaces*, 38:101–112. 120

Rector, A. L., Drummond, N., Horridge, M., Rogers, J., Knublauch, H., Stevens, R., Wang, H., and Wroe, C. (2004). OWL pizzas: Practical experience of teaching OWL-DL: common errors & common patterns. In Motta, E., Shadbolt, N., Stutt, A., and Gibbins, N., editors, *Engineering Knowledge in the Age of the Semantic Web, 14th International Conference, EKAW 2004, Whittlebury Hall, UK, October 5-8, 2004, Proceedings*, volume 3257 of *Lecture Notes in Computer Science*, pages 63–81. Springer. 30, 45, 53, 57

Rodríguez, N. D., Cuéllar, M. P., Lilius, J., and Delgado Calvo-Flores, M. (2014a). A fuzzy ontology for semantic modelling and recognition of human behaviour. *Knowl.-Based Syst.*, 66:46–60. 197

Rodríguez, N. D., Cuéllar, M. P., Lilius, J., and Delgado Calvo-Flores, M. (2014b). A survey on ontologies for human behavior recognition. *ACM Comput. Surv.*, 46(4):43:1–43:33. 197

Rodríguez-Doncel, V., Gómez-Pérez, A., and Mihindukulasooriya, N. (2013). Rights declaration in linked data. In Hartig, O., Sequeda, J., Hogan, A., and Matsutsuka, T., editors, *Proceedings of the Fourth International Workshop on Consuming Linked Data, COLD 2013, Sydney, Australia, October 22, 2013*, volume 1034 of *CEUR Workshop Proceedings*. CEUR-WS.org. 53

Roussey, C., Corcho, O., Sváb-Zamazal, O., Scharffe, F., and Bernard, S. (2012). SPARQL-DL queries for Antipattern Detection. In Blomqvist, E., Gangemi, A., Hammar, K., and del Carmen Suárez-Figueroa, M., editors, *Proceedings of the 3rd Workshop on Ontology Patterns, Boston, USA, November 12, 2012*, volume 929 of *CEUR Workshop Proceedings*. CEUR-WS.org. 214

Sabou, M. and Fernández, M. (2012). Ontology (network) evaluation. In Suárez-Figueroa, M. d. C., Gómez-Pérez, A., Motta, E., and Gangemi, A., editors, *Ontology Engineering in a Networked World.*, pages 193–212. Springer. 24, 28

Schober, D., Tudose, I., Svatek, V., and Boeker, M. (2012). OntoCheck: verifying ontology naming conventions and metadata completeness in protégé 4. *Journal of biomedical semantics*, 3(S-2):S4. 42

Shore, J. and Warden, S. (2007). *The art of agile development*. O'Reilly. 27

Staab, S., Studer, R., Schnurr, H.-P., and Sure, Y. (2001). Knowledge processes and ontologies. *IEEE Intelligent systems*, 16(1):26–34. xv, 1, 19, 21

Studer, R., Benjamins, V. R., and Fensel, D. (1998). Knowledge engineering: Principles and methods. *Data & Knowledge Engineering*, 25(1-2):161–197. 1

Suárez-Figueroa, M. C. (2010). *NeOn Methodology for Building Ontology Networks: Specification, Scheduling and Reuse*. PhD thesis, Universidad Politécnica de Madrid, Spain. 2, 16, 22

Suárez-Figueroa, M. C., Blomqvist, E., d'Aquin, M., Espinoza, M., Gómez-Pérez, A., Lewen, H., Mozetic, I., Palma, R., Poveda, M., Sini, M., Villazón-Terrazas, B., Zablith, F., and Dzbor, M. (2009). D5.4.2 Revision and Extension of the NeOn Methodology for Building Contextualized Ontology Networks. Technical report, Universidad Politécnica de Madrid (UPM). NeOn Project. http://www. neon-project. org. 27

Suárez-Figueroa, M. C., Cea, G. A. d., and Gómez-Pérez, A. (2013a). Lights and shadows in creating a glossary about ontology engineering. *Terminology*, 19(2):202–236. 2, 15

Suárez-Figueroa, M. C., Gómez-Pérez, A., and Fernández-López, M. (2015). The NeOn Methodology framework: A scenario-based methodology for ontology development. *Applied Ontology*, 10(2):107–145. xv, 2, 22, 23, 53, 211

Suárez-Figueroa, M. C., Kamel, M., and Poveda-Villalón, M. (2013b). Benefits of natural language techniques in ontology evaluation: the OOPS! Case. *10th International Conference on Terminology and Artificial Intelligence (TIA 2013)*, pages 107–110. 134, 213

Suominen, O. and Hyvönen, E. (2012). Improving the quality of skos vocabularies with Skosify. In ten Teije, A., Völker, J., Handschuh, S., Stuckenschmidt, H., d'Aquin, M., Nikolov, A., Aussenac-Gilles, N., and Hernandez, N., editors, *Knowledge Engineering and Knowledge Management - 18th International Conference, EKAW 2012, Galway City, Ireland, October 8-12, 2012. Proceedings*, volume 7603 of *Lecture Notes in Computer Science*, pages 383–397. Springer. 35

Tartir, S., Arpinar, I. B., Moore, M., Sheth, A. P., and Aleman-Meza, B. (2005). OntoQA: Metric-based ontology quality analysis. In *IEEE Workshop on Knowledge Acquisition from Distributed, Autonomous, Semantically Heterogeneous Data and Knowledge Sources*. 31

Vandenbussche, P.-Y. and Vatant, B. (2012). Metadata Recommendations For Linked Open Vocabularies. OKFN. `http://lov.okfn.org/dataset/lov/Recommendations_Vocabulary_Design.pdf`. 53

Völker, J., Vrandečić, D., Sure, Y., and Hotho, A. (2008). AEON - an approach to the automatic evaluation of ontologies. *Applied Ontology*, 3(1-2):41–62. xv, 3, 37, 38

Vrandečić, D. (2010). *Ontology Evaluation*. PhD thesis, KIT, Fakultät für Wirtschaftswissenschaften, Karlsruhe. xv, 2, 15, 32, 33, 34, 45, 53, 119

Vrandečić, D. and Gangemi, A. (2006). Unit tests for ontologies. In Meersman, R., Tari, Z., and Herrero, P., editors, *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops. Montpellier, France, October 29 - November 3, 2006. Proceedings, Part II*, volume 4278 of *Lecture Notes in Computer Science*, pages 1012–1020. Springer. 32

Welty, C. A. and Guarino, N. (2001). Supporting ontological analysis of taxonomic relationships. *Data & Knowledge Engineering*, 39(1):51–74. 2, 3, 29, 45

Winkler, R. and Hays, W. (1985). Statistics: probability, inference, and decision. *Holt, Rinehart and Winston*. 111

ANNEX A

OOPS! user interface and web service

A.1 Web user interface

Figure A.1 shows OOPS! first prototype that aimed at providing free and online ontology diagnosis features for ontology developers by means of a web application[86]. Such web application provides a user interface based on HTML[87], jQuery[88], JSP[89] and CSS[90] technologies.

The main page of the web application allows the user to enter the ontology to be analysed by indicating its URI or by pasting its OWL code as indicating in Figure A.1. By pressing the evaluation button, the user will get back the evaluation results for all the pitfalls for which a detection method have been implemented (see Section 5.3).

If the user would like to analyse only the appearance of a subset of pitfalls, he/she should go to "advanced evaluation"[91]. In this case, the user might choose between the following options:

- selecting a number of specific pitfalls as shown in Figure A.2 option a).

[86]The web user interface is currently available at http://oops.linkeddata.es

[87]http://www.w3.org/html/wg/ (last visited on the 14th October, 2015)

[88]http://jquery.com/ (last visited on the 14th October, 2015)

[89]http://www.oracle.com/technetwork/java/javaee/jsp/index.html (last visited on the 14th October, 2015)

[90]http://www.w3.org/Style/CSS/ (last visited on the 14th October, 2015)

[91]The advance evaluation feature is available at http://oops.linkeddata.es/advanced

- selecting groups of pitfalls according to the ontology evaluation dimensions and aspects presented in Section 4.4 as shown in Figure A.2 option b).
- selecting groups of pitfalls according to the evaluation criteria according to (Gómez-Pérez, 2004) as shown in Figure A.2 option c). It is worth noting that not all pitfalls can be classified under this criteria.

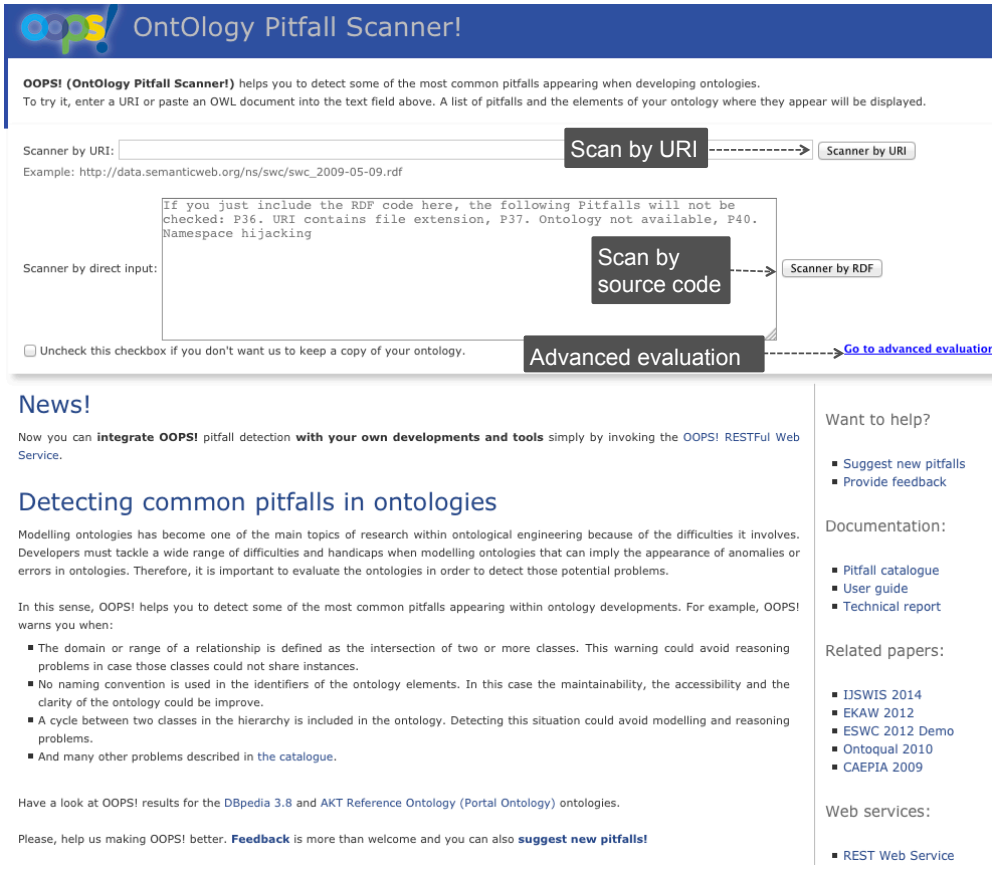


Figure A.1: OOPS! main page

Once the ontology has been diagnosed, the results are displayed in the web user interface as shown in Figure A.3. It can be observed that for each detected pitfall the following information is displayed:

(a) the code and title of the pitfall: (b) how many times the pitfall has been detected; (c) the importance level; (d) the pitfall description and example; and (e) the list of ontology elements affected by the pitfall or whether the pitfall affects the ontology itself instead of particular elements.

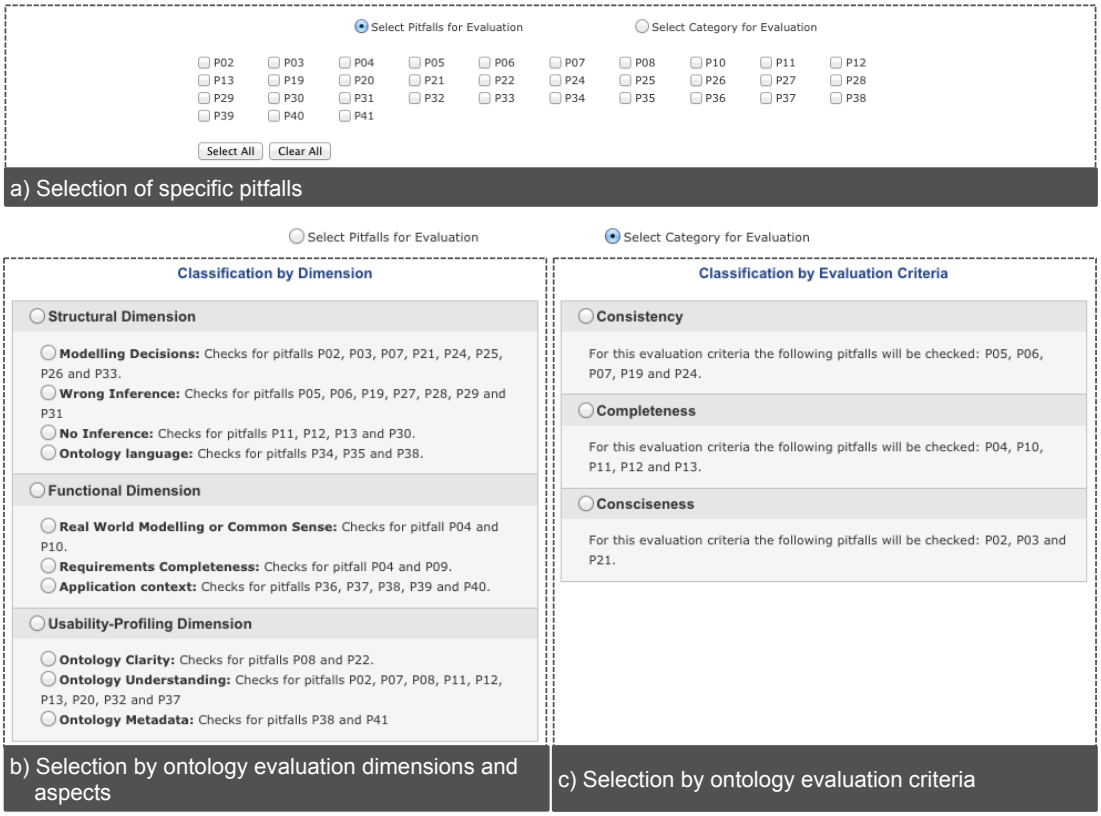


Figure A.2: Selecting particular pitfalls or groups to be analysed

In case any modelling suggestion have been detected by the suggestion module (see Figure 5.1), they are shown within the evaluation results following the last pitfall detected.

A.2 Web service

While the main goal was to provide ontology developers with a graphical user interface for diagnosing their ontologies, it is undoubtedly useful to allow third party software to run evaluations too. In this sense, not only ontology editors, registries or catalogues could include ontology evaluation features in their functionalities but also they could execute batch evaluations over a number of ontologies.

To allow third-party software to integrate OOPS! features, a REST web service has been developed[92]. The web service uses Jena API as RDF parser like the web user

[92]The REST web service is currently available at http://oops-ws.oeg-upm.net/rest while the related

Figure A.3: OOPS!'s output example through web user interface

interface (see Section A.1) plus Jersey framework[93] to build a system compliant with REST architecture. The request should be invoked by an HTTP POST following the XML body presented in Listing A.1, which have the following fields:

- **OntologyURI**: this field contains the ontology URI. It should be noted that this field is mandatory in case the OntologyContent field is empty.
- **OntologyContent**: this field contains the ontology RDF source code inside `<![CDATA[ RDF code ]]>`. This field is mandatory in case OntologyURI field is empty.
- **Pitfalls**: this field indicates the list of pitfalls to be scanned. If the user wants to

documentation can be found at http://oops-ws.oeg-upm.net/

[93]https://jersey.java.net/ (last visited on the 14th October, 2015)

analyse only a subset of pitfalls, the identification number of each pitfall (separated with comma if more than one pitfall is selected) should be indicated, for example: "P04,P11,P21". If no particular pitfalls are indicated, the web service will analyse all of them. This field is optional.

- **OutputFormat**: this field indicates which response format is demanded. The output formats available at the moment of writing this thesis "XML" and "RDF/XML". If any other value is entered, the default output will be RDF/XML. This field is optional.

```
<?xml version="1.0" encoding="UTF-8"?>
<OOPSRequest>
  <OntologyURI></OntologyURI>
  <OntologyContent></OntologyContent>
  <Pitfalls></Pitfalls>
  <OutputFormat></OutputFormat>
</OOPSRequest>
```

Listing A.1: XML request

It is worth noting that the user must enter either the OntologyURI or the OntologyContent value. If none of them is given, an RDF error message *"Invalid arguments. No ontology url or ontology rdf text found in the input parameters. Please check your request parameters."* will be received as response. If both of them are entered, the service will use the OntologyContent.

Listing A.2 shows an example of request in which the ontology to be analysed is indicated by its URI and the output is required to be in XML format. As no specific pitfalls are indicated, the system will analyse all the implemented pitfalls.

```
<?xml version="1.0" encoding="UTF-8"?>
<OOPSRequest>
  <OntologyURI>http://www.cc.uah.es/ie/ont/learning-resources.owl</OntologyURI>
  <OntologyContent></OntologyContent>
  <Pitfalls></Pitfalls>
  <OutputFormat>XML</OutputFormat>
</OOPSRequest>
```

Listing A.2: Example of XML request entering the ontology URI, evaluating all the pitfalls and asking for XML output

Listing A.3 shows an example of request in which the ontology code to be analysed is passed directly within the request (line 5 - line 38), only the pitfalls "P10" and "P11" are requested to be analysed and the output is required to be in RDF/XML format.

```
<?xml version="1.0" encoding="UTF-8"?>
<OOPSRequest>
  <OntologyURI></OntologyURI>
  <OntologyContent><![CDATA[
<?xml version="1.0"?>
<rdf:RDF
    xmlns="http://www.cc.uah.es/ie/ont/learning-resources#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://www.cc.uah.es/ie/ont/learning-resources">
  <owl:Ontology rdf:about="">
    <rdfs:comment xml:lang="en">An upper ontology for models of</rdfs:comment>
  </owl:Ontology>
  <owl:Class rdf:ID="LearningObject">
    <rdfs:comment xml:lang="en">"A digital learning resource"</rdfs:comment>
  </owl:Class>
  <owl:Class rdf:ID="ExerciseLO">
    <rdfs:comment xml:lang="en">"A task, problem, or other effort performed to
    develop or maintain fitness
    or increase skill:"</rdfs:comment>
    <rdfs:subClassOf rdf:resource="#LearningObject"/>
  </owl:Class>
  <owl:ObjectProperty rdf:ID="partOf">
    <owl:inverseOf>
      <owl:TransitiveProperty rdf:ID="hasPart"/>
    </owl:inverseOf>
    <rdfs:range rdf:resource="#LearningObject"/>
    <rdfs:domain rdf:resource="#LearningObject"/>
  </owl:ObjectProperty>
  <owl:TransitiveProperty rdf:about="#hasPart">
    <rdfs:range rdf:resource="#LearningObject"/>
    <rdfs:domain rdf:resource="#LearningObject"/>
    <rdfs:comment xml:lang="en">Specifies that a LO has as one of its
    constituent another LO.</rdfs:comment>
    <owl:inverseOf rdf:resource="#partOf"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
  </owl:TransitiveProperty>
</rdf:RDF>
]]></OntologyContent>
  <Pitfalls>P10,P11</Pitfalls>
  <OutputFormat>RDF/XML</</OutputFormat>
</OOPSRequest>
```

Listing A.3: Example of XML request entering the ontology code, evaluating a selection of pitfalls and asking for RDF/XML output

A the output provided by the service could be provided both in RDF/XML or XML format, and for the sake of simplicity we describe here the RDF/XML output that follows the structure:

- **Response element**: every response element represents a resource belonging to the class `oops:Response`. This is the main resource in the RDF file. This resource will contain as many elements as pitfalls or suggestions were detected during the diagnose of the ontology.
- **Pitfall element**: every pitfall element represents a resource belonging to the class `oops:Pitfall`. For every pitfall detected, a new resource of this type will be created and associated with the response element.
  - This element contains a set of fixed properties:
    - `oops:hasCode`: indicates the code of the pitfall.
    - `oops:hasName`: indicates the name of the pitfall.
    - `oops:hasDescription`: indicates the description of the pitfall.
    - `oops:hasImportanceLevel`: indicates the importance level of the pitfall.
    - `oops:hasNumberAffectedElements`: indicates how many times the pitfall has been detected in the ontology.
  - Depending on the pitfall, it could contain some of the following elements:
    - `oops:hasAffectedElement`: indicates the URI of the element (class, object property or datatype property) affected.
    - `oops:mightBeEquivalentProperty`: points to a resource that will contain the URIs of the elements (class, object property or datatype property) affected.
    - `oops:mightBeEquivalentAttribute`: points to a resource that will contain the URIs of the elements (class, object property or datatype property) affected.
    - `oops:mightNotBeInverseOf`: indicates the URI of the element (class, object property or datatype property) affected.
- **Suggestion element**: every suggestion element represents a resource belonging to the class `oops:Suggestion`. For possible suggestions detected, a new resource of this type will be created and associated with the response element. This element contains a set of fixed elements:
    - `oops:hasName`: indicates the name of the suggestion.

- `oops:hasDescription`: indicates the description of the suggestion value.
- `oops:hasAffectedElement`: this element will contain as many URIs as elements detected during the analysis.

Listing A.4 shows an example of response provided by the web service in format RDF/XML for the request shown in Listing A.3. It can be observed that the request indicated three pitfalls to be detected from which only two have been found.

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:oops="http://www.oeg-upm.net/oops#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" >
  <rdf:Description rdf:about="http://www.oeg-upm.net/oops#suggestion">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.oeg-upm.net/oops/fdea1aa6-71d6-4557-
    a17a-dc3244ff536b">
    <oops:hasCode rdf:datatype="http://www.w3.org/2001/XMLSchema#string">P10</
    oops:hasCode>
    <oops:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Missing
     disjointness [1, 2, 3]</oops:hasName>
    <oops:hasDescription rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    The ontology lacks disjoint axioms between classes or between properties
    that should be defined as disjoint.</oops:hasDescription>    <rdf:type
    rdf:resource="http://www.oeg-upm.net/oops#pitfall"/>
     <oops:hasImportanceLevel rdf:datatype="http://www.w3.org/2001/XMLSchema#
    string">Important</oops:hasImportanceLevel>
     <oops:hasNumberAffectedElements rdf:datatype="http://www.w3.org/2001/
    XMLSchema#integer">1</oops:hasNumberAffectedElements>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.oeg-upm.net/oops/496ae03d-48c6-406d-8
    d07-530bf05c9ac1">
    <oops:hasPitfall rdf:resource="http://www.oeg-upm.net/oops/fdea1aa6-71d6
    -4557-a17a-dc3244ff536b"/>
    <rdf:type rdf:resource="http://www.oeg-upm.net/oops#response"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.oeg-upm.net/oops#pitfall">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  </rdf:Description>
</rdf:RDF>
```

Listing A.4: Example of XML/RDF response from the web service

ANNEX B

OOPS! user questionnaire

OOPS! evaluation survey

This survey contains questions about OOPS! (OntOlogy Pitfalls Scanner!) and the ontology evaluated.

Information about the ontology evaluated:

1. Name of the ontology evaluated
2. URI of the ontology evaluated
3. Domain/domains modelled in the ontology
4. Brief description of the ontology

OOPS! - OntOlogy Pitfalls Scanner!
The following questions are about OOPS!

5. The output generated by OOPS! shows clearly what is the problem detected and in which elements.
   - ▫ Strongly Agree
   - ▫ Agree
   - ▫ Undecided
   - ▫ Disagree
   - ▫ Strongly Disagree
6. Please explain, if needed, your answer for the previous question.
7. The output generated by OOPS! has been useful for: (Please describe it and/or list the type of error in which detection OOPS! has been useful.)
8. The output generated by OOPS! has not been useful for: (Please describe it and/or list the type of error in which detection OOPS! has not been useful.)

9. In your ontology developments, do you think you will use again OOPS! for validating your ontology?
   - ▫ Yes, always
   - ▫ Yes, sometimes
   - ▫ No
10. Please explain, if needed, your answer for the previous question.
11. In general, do you think OOPS! is:
   - ▫ Very useful
   - ▫ A nice gadget
   - ▫ Not good enough
12. Please explain, if needed, your answer for the previous question.
13. Would you recommend OOPS! to other colleagues involved in ontology development projects?
   - ▫ Yes, always
   - ▫ Yes, sometimes
   - ▫ No
14. Please explain, if needed, your answer for the previous question.
15. Which features or improvements would you add to OOPS!?
16. From your point of view, what is the main contribution of OOPS!?
17. Have you ever used other ontology evaluation tools apart from OOPS!? (Please, list them here.)
18. In case you have used other ontology evaluation tools: Which features from the other ontology evaluation tools would you add to OOPS! (Please, explain them here, relating the features with the tool providing them if possible.)
19. In case you have used other ontology evaluation tools: Would you change them for OOPS or use all of them as complements? (Please, explain your answer here.)
20. If you have any other comments or suggestions about OOPS!, please write them here. (You can also add your name and email address if there is any particular question that you want us to answer personally or just if you want to let us know who is our user :-) We will be glad)