Workshop on Ontology Patterns

WOP 2009

Papers and Patterns from the ISWC workshop

Introduction

High-quality and reusable ontologies are considered as key element of the Semantic Web and for successful semantic applications. Ontology Design Patterns (ODPs) are addressing these quality and reusability issues by providing different types of patterns supporting ontology designers. ODPs are collected in various repositories, such as the catalogue maintained by the University of Manchester and the ODP portal at ontologydesignpatterns.org. However, pattern catalogues are still small and do not cover all types of patterns and all domains. Semantic Web applications could also benefit from additional types of patterns, such as enterprise model patterns and specialized software patterns for semantic applications.

Patterns are an approach to knowledge reuse that proved feasible and very beneficial in various areas, such as software engineering and data modeling. Reuse has been an important research subject in ontology engineering and the semantic web community for many years. Patterns need to be shared by a community in order to provide a common language and stimulate pattern usage and development. Hence, the aim of this workshop was twofold

- providing an arena for proposing and discussing good practices, patterns, pattern-based ontologies, systems etc.,
- broadening the pattern community that will develop its own "language" for discussing and describing relevant problems and their solutions..

The workshop included a track for research papers addressing various aspects of ontology patterns and investigating application areas, and a pattern track focusing on presentation and discussion of actual ontology patterns.

We received 21 submissions for the paper and poster track of the workshop. The program committee selected 7 submissions for oral presentation and 8 submissions as short papers for presentation in the poster session. 13 ontology patterns were submitted to the workshop, of which 3 patterns were selected for discussion in the pattern writing session and 5 patterns for presentation in the poster session. Further information about the Workshop on Ontology Patterns can be found at: http://ontologydesignpatterns.org/wiki/WOP2009.

Acknowledgments

We thank all members of the steering committee, program committee, authors and local organizers for their efforts and support. We appreciate support from the FP6 NeON Integrated Project (IST-2005-027595) and the DEON project (STINT IG 2008-2011).

Eva Blomqvist
Kurt Sandkuhl
François Scharffe
Vojtěch Svátek

October 2009

Organization

WOP2009 Chairs

Paper chair: Kurt Sandkuhl, Jönköping University (SE)
Poster chair: Vojtěch Svátek, University of Economics (CZ)
Pattern chairs: Eva Blomqvist, ISTC-CNR (IT) and François Scharffe, INRIA (FR)

Steering Committee

Eva Blomqvist, ISTC-CNR (IT)
Aldo Gangemi, ISTC-CNR (IT)
Natasha Noy, Stanford University (US)
Valentina Presutti, ISTC-CNR (IT)
Alan Rector, University of Manchester (UK)
Francois Scharffe, INRIA (FR)
Steffen Staab, University of Koblenz (DE)
Chris Welty, IBM Watson Research Center (US)

Program Committee

Alessandro Adamou, ISTC-CNR (IT)
Marie-Aude Aufaure, Ecole Centrale Paris (FR)
Fabio Ciravegna, University of Sheffield (UK)
Mathieu D'Aquin, Open University (UK)
Enrico Daga, ISTC-CNR (IT)
Violeta Damjanovic, Salzburg Research (AT)
Rim Djedidi, Paris-Sud University (FR)
Henrik Eriksson, Linköping University (SE)
Aldo Gangemi, ISTC-CNR (IT)
Jose-Manuel Gomez, Universidad Politécnica de Madrid (ES)
Gerd Groener, University of Koblenz (DE)
Luigi Iannone, University of Manchester (UK)
Holger Lewen, AIFB University of Karlsruhe (DE)
Pierluigi Miraglia, Gerson Lehrman Group (US)
Mark Musen, Stanford University (US)
Natasha Noy, Stanford University (US)
Wim Peters, University of Sheffield (UK)
Valentina Presutti, ISTC-CNR (IT)
Alan Rector, University of Manchester (UK)
Alan Ruttenberg, Science Commons Cambridge, MA (US)
Marta Sabou, Open University (UK)
Guus Schreiber, VU University Amsterdam (NL)

Steffen Staab, University of Koblenz (DE)
Mari Carmen Suárez-Figueroa. Universidad Politécnica de Madrid (ES)
Tania Tudorache. Stanford University (US)
Boris Villazón-Terrazas, Universidad Politécnica de Madrid (ES)
Holger Wache, University of Applied Sciences Northwestern Switzerland (CH)
Chris Welty, IBM Watson Research Center (US)

Additional Reviewers

Guillermo Alvaro, Universidad Politécnica de Madrid (ES)
Andreas Billig, Fraunhofer ISST Berlin (DE)
Carlos Ruiz, Universidad Politécnica de Madrid (ES)

Table of Contents

Part 1:

Research Papers

A Pattern-based Ontology Building Method for Ambient Environments

Wolfgang Maass[1,2] and Sabine Janzen[1]

[1] Research Center for Intelligent Media (RCIM)
Furtwangen University, Robert-Gerwig-Platz 1
D-78120 Furtwangen, Germany
{wolfgang.maass,sabine.janzen}@hs-furtwangen.de
http://im.dm.hs-furtwangen.de

[2] Institute of Technology Management, University of St. Gallen
Dufourstrasse 40a, CH-9000 St. Gallen, Switzerland
wolfgang.maass@unisg.ch
http://www.item.unisg.ch

Abstract. Ambient environments are characterized by an ever increasing amount of information that needs to be selected and organized in order to make correct assumptions about users, entities, etc. within a specific context. This issue can be addressed by using ontologies that meet the specific requirements of such environments. In this paper, we survey ontology engineering methods that represent an adequate approach to creating adequate ontologies. Because unprecedented, we introduce a Pattern-based Ontology Building Method for Ambient Environments (POnA) and exemplify this method through the development of a domain-specific ontology for cosmetic products within ambient shopping environments.

Key words: Ambient environments, design patterns, ontology engineering methods

1 Introduction

In recent years, research in intelligent environments and entities has increased. The embedding of adaptive information and communication services into everyday physical things characterizes ambient environments (Ambient Intelligence (AmI)). A number of prototype systems for ambient environments have been designed and implemented, such as [1–4]. The challenge is to interpret sensor data so that adaptive behavior of ambient environments can be generated which naturally supports users in, for instance, situations of problem solving, relaxing, informing, and communicating with other users. Available information needs to be selected and organized so that ambient environments are able to make correct assumptions about users and physical entities within a specific context [5]. However, a major shortcoming of these AmI systems is weak support of knowledge sharing [6], i.e., AmI systems are typically based on proprietary and fixed sets of representations that are designed for particular AmI applications. This, in turn, poses hurdles for integrating external information that is stored in the infosphere of digital environments, such as the World Wide Web (WWW). Bridging the gap between sensor-data-driven applications and infospheres requires interoperable knowledge representations. Through this, information can be reused in both directions. Because both AmI environments and infospheres can become quite complex, a more principled, ontology-driven approach for the creation of appropriate knowledge representations is required [6]. Initial approaches were top-down approaches based on foundational ontologies that drilled down conceptual structures until they reached a conceptual level compatible with information derived from sensor data (e.g., [7]). For complexity reasons, these systems neglected most of the ontological structure that are given by the overarching ontology, which meant that the ontology was only helpful at design time. In these systems, semantics coded into ontological structures were typically not used at run-time.

Some applications that used ontologies indicated the value of smaller pieces of more abstract ontological structures, called design patterns, that could reused. The idea of design patterns has several origins, such as Christopher Alexander's work on design patterns in architecture, Kevin Lynch's work on mental models of city structures, and computational knowledge patterns [8, 9]. Ontology design patterns are conceptual macros that are repeatedly used by experts for solving problems in their particular domains (cf. [10]). They are a means for extracting knowledge from experts that can be used for designing applications with a holistic understanding of a domain. Furthermore, machine-processible representations of design patterns can be used for the generation of a system behavior itself.

AmI environments try to support natural behavior. Therefore ontology design patterns must combine domain knowledge and enhancements using information services grounded in sensor data and Web-based infospheres. In general AmI environments are required to be context-oriented, user-centered, and network-enabled [7]. Context-orientation is achieved by adaptating to particular situations, to objects within these situations, and to domain constraints such as contractual restrictions. Furthermore, AmI environments adapt to users within situations in a personalized and proactive manner. When an AmI environment is network-enabled, any object can in principle establish relationships with any other object or service within and beyond a particular situation. In summary, ontology design patterns for AmI environments can be defined on various layers: (1) users, (2) objects, (3) services, (4) physical space, (5) infosphere (information space), and (6) social space [11]. (4) through (6) establish networks among entities from (1) through (3). A detailled discussion is beyond the scope of this article.

Here, we discuss a tentative design method for ontology design patterns (POnA) and its application to the design of AmI environments with a focus on Natural Language Processing (NLP). A review of existing methods has revealed (cf. Section 2&3) that a dedicated method suitable for ambient environments is not yet available. Therefore we introduce a *Pattern-based Ontology Building Method for Ambient Environments (POnA)* and exemplify this method through the development of a pattern-based ontology for an AmI shopping environment. Next, we will discuss existing ontology building approaches and their applicability in AmI environments. In Section 3, we illustrate POnA through the development of an exemplary ontology. We then discuss some findings (Section 4) that exemplify patterns (Section 5). Finally, we conclude with a summary and an outlook on future work.

2 Related Work

In general, there are diverse approaches for the design and development of ontologies [12]. Systematic methodologies concentrate on the ontology development process and are independent of particular languages. Patterns are light-weight versions of methodologies that include useful hints. Svatek argues that repositories of such reusable patterns might improve the accuracy, transparency and reasonability of an ontology [12]. In the following, systematic methodologies and pattern-based approaches for ontology engineering are briefly described.

2.1 Systematic Ontology Building Methods

There are diverse ontology building methods for designing ontologies (for a review of earlier methodologies cf. [13] NeOn 5.4.1.). For instance, Uschold and King describe a methodology for building ontologies anew [14] while METHONTOLOGY considers reuse of other ontologies [15]. The more recent NeOn methodology focusses on the reuse and combination of distributed ontologies with a special emphasis on ontology design patterns [16]. The Unified Process for Ontology building (UPON) [17] applies a phase-structured software engineering viewpoint by following the Unified Process model. All these methodologies follow a basic pattern. First, a scope is defined by a domain of interest. Second, scenarios are defined, which are used to identify competency questions (CQ) to be answered by the ontology [18]. Thereafter, terms are gathered and translated into formal concepts that are connected. The final result is called a formal ontology. The NeOn methodology defines ontology design patterns as best practices for more efficient ontology engineering processes. Additionally it describes generic processes for ontology evaluation, evolution, and localization [16].

Even though the NeOn methodology is rich with respect to complete ontology life cycle, it lacks depth with regard to the application of design patterns. At the moment, matching problems with ontology design patterns and reusing and composing of ontology design patterns are complex tasks still left to the knowledge engineer's expertise [16]. In the following, we describe a methodology that combines early phases of the NeOn methodology and the UPON methodology for designing ontologies for AmI environments. Special emphasis is given to a problem solving viewpoint often found in AmI scenarios.

2.2 Building Pattern-based Ontologies

Christopher Alexander introduced the term “design pattern” for shared guidelines that help to solve architectural design problems [19]. He argued that a good design can be achieved using rule sets, i.e. patterns. The potential for reusing ontological structures through a pattern-based approach was first developed by Clark et al. [20]. They emphasized the importance of combining concepts within a vocabulary using “knowledge patterns”, i.e., frequently recurring, structurally similar patterns of axioms. The notion of ontology design patterns was used by Gangemi [10] when presenting Conceptual Ontology Design Patterns (CODePs) as a useful resource for engineering ontologies for Semantic web infrastructures. CODePs are represented by textual, semiformal, and formal descriptions similar to Alexander’s initial approach.

3 Pattern-based Ontology Building Method for Ambient Environments (POnA)

Our goal is the development of an ontology design pattern library specific to AmI environments that considers all six layers (cf. Section 1). These patterns are grounded in the patterns of the Ontology Design Pattern ODP library.[3] The PoNA methodology reuses UPON’s detailed engineering approach and combines it with an approach proposed by the NeOn methodology that is centered on ontology design patterns. Currently, we focus on early development phases of ontology engineering. Rigorous ontology evaluation and evolution phases will be considered in our future work.

Ontological design patterns have several advantages. They improve explicit modularizations of knowledge bases and enable separation of abstract theories from real-world phenomena [20]. Their usage ensures that better explications concerning structure and modeling decisions are made when constructing a formal axiom-rich ontology [20, 10]. Furthermore, ontology design patterns provide new opportunities for ontology integration [21]. Contrary to systematic methodologies, pattern-based approaches do not dispose of structured methodologies that consist of specific activities and outcomes. Our hypothesis is that a combination of systematic methodologies (cf. Section 2.1) and ontology design patterns (cf. Section 2.2) constitute a more detailled and thus efficient approach to designing ontologies for ambient environments.

In the following, we present the ***P****attern-based* ***On****tology Building Method for* ***A****mbient Environments (POnA).* Following UPON [17], POnA consists of four engineering phases: *Requirements, Design, Implementation* and *Test.* Each phase is subdivided into activities, contains decision points, and provides clearly defined outcomes. The re-use of design patterns is integrated into the design phase. We will exemplify POnA for an AmI application in the cosmetics domain [22].

[3] http://ontologydesignpatterns.org

3.1 Requirements Phase

The requirements phase aims at the identification of business needs for modeling a domain of interest and specification of the environmental aspects of the ontology, e.g., users and scenarios. This phase consists of the following activities: *(1) defining the domain of interest and scope, (2) identifying objectives, (3) defining scenarios, (4) defining terminology* and *(5) identifying competency questions.*

Defining Domain of Interest & Scope According to [14] and [17], defining a domain of interest is an important step in focusing on a particular fragment of the world to be modeled. Therefore, prospective users and the type of ontology to be used are circumscribed. In line with the scope of a particular ontology, the most important concepts and their characteristics are identified. Consequently, some parts of a domain of interest are brought into focus. Within our cosmetics domain, we focus on ontological representations of the following product concepts: *perfume and fragrances, make-up for eyes, makeup for lips, hand care, nail care, foot care, make-up for facial skin, liquid hair care, hair spray, hair color, dental care, shower gel, skin care, and sunscreen.* Target groups of resulting ontologies are manufacturers, retailers and customers, which might use the ontology for two different real-world situations:

– **In-store purchase situation** - The customer communicates with a cosmetic product. She might search for an individual solution, e.g.”I have dry skin. Which vanishing creme suits me?”
– **Usage situation at home** - A product advises a customer on correct application procedure and initiates re-purchases through communication with appropriate web-based shopping services.

Identifying Objectives In this step, motivations for an ontology are collected together with associated problems. This step is important for later reuse of ontologies. It indicates to other knowledge engineers the importance of this ontology and the problem types that are targeted. We found that physical products are mainly described in a non-semantic way; their descriptions exist in terms of static databases or XML structures (e.g., BMEcat®, ETIM/eCl@ss and GS1). Modeling of enterprises or processes is generally sophisticated, but the description of products rarely exceeds the scope of classification. We intend to integrate physical products into communicative situations in ambient shopping environments. Therefore, semantically annotated product information is required to realize personalized communications between different stakeholders and products.

Defining Situations Situations are textual descriptions of integrated performances of a particular interaction type. Situations conceive different entities (objects, subjects, information, and services) and their interactions (for a discussion cf. [11]). Situations are prototypes that reflect characteristic features of a corresponding class of situations, e.g., shopping situations. Thus, situations resemble frames [8], schemas [9], and use cases. For instance:

"Anna has dry skin and searches for a vanishing creme matching her skin in a shop. She wants to take a look at the cremes that are right for her, so she asks all products in the store for a solution to her specific skin problem. Six vanishing cremes give notice that they want to solve her problem because they are suitable for dry skin. Anna goes to the creme closest to her and initiates a dialogue with the intelligent product. She asks whether the creme is a gel-based moisturizer. Furthermore, she wants to know about the ingredients. Then, Anna asks for the price as well as current discount campaigns and matching products. The creme informs Anna about a bundle price with a 5% discount for the vanishing creme and the corresponding eye care. Anna decides to buy both products."

Defining Terminology The terminology was extracted from situation reports gathered in workshops with experts and was automatically extracted based on Web-based product descriptions. For the cosmetics domain, 130 terms were extracted and categorized into five term categories (for an excerpt see Table 1): *Actors and Roles, Products and Features, Environment and Situation, Problems* and *Solutions.*

Actors and Roles	**Products and Features**	**Environment and Situation**	**Problems**	**Solutions**
User	Name	Shop	Colored hair	Protecting lips
Manufacturer	Price	Services	Oily skin	Treating skin
Father	Bundle	Retailer	Dry skin	Coloring hair
Desire	Perfume	Time	Oily hair	Painting nails

Table 1. Extract of the cosmetics terminology

Identifying Competency Questions Competency questions (CQs) are conceptual questions that the ontology must be able to answer [23]. They were identified through analysis of scenarios and terminology as well as through brainstorming with domain experts. CQs were used during the test phase of POnA to evaluate the quality of resulting ontologies [17]. Some examples of CQs are

CQ1	Does the product fit to me?
CQ2	Does the product solve my problem?
CQ3	How long does the product last?
CQ4	Is my purchase decision correct?
CQ5	Which product can I use for protecting my lips (in winter)?
CQ6	How can I apply the product?

Table 2. Examples of CQs for the cosmetics domain

presented in Table 2. The output of the requirements phase consists of CQs, scenarios, and the terminology with corresponding term categories.

3.2 Design Phase

The design phase aims at the identification of semantic structures, more precisely the definition of design patterns for answering CQs. First, relations between terms are identified, which results in coarse term structures. Based on descriptions of situation types, CQs and term structures, prototypical ontology design patterns (PODPs) are derived and formally modeled by reusing ODPs grounded in DOLCE [10]. Complex prototypical ontology patterns require the combination and adaptation of ODPs. Resulting ontology design patterns are discussed with domain experts again. PODPs are informal conceptual structures that are derived from analysis of situations, terms, and CQs. Therefore, PODPs resemble more mental models of real world perceptions [24] than formal logic representations and fit very well to the requirements of AmI environments. PODPs consist of conceptual entities, called scopes, and relations. Scopes are themes that frequently occur in situations, terms and CQs and share a common meaning. The design phase consists of the following activities: *(1) identification of prototypical ontology design patterns, (2) terminology setup*, and *(3) mapping PODPs onto ODPs.*

Identification of Prototypical Ontology Design Patterns In contrast to UPON [17], CQs are mapped onto PODPs that in turn are mapped onto formal ODPs [19, 20, 10]. PODPs and ODPs provide levels of granularity that support discussions with experts much better than discussions of isolated concepts and relationships. Generally, it is proposed that an accurate domain ontology specifies all and only those conceptualizations that are required for answering all the CQs formulated within the requirements phase [10]. The most general CQ for product-centered situations is classified as a problem-solution situation, i.e., “Which solutions exist for this problem?” Through analysis of the 55 CQs, seven POSPs are derived: *Product-Pattern (P), Product-Product-Pattern (PP), Context-Pattern (C), Product-User-Pattern (PU), Product-Information-Pattern (PI), Product-User-Context-Pattern (PUC)*, and *Product-Context-User-Information-Pattern (PUIC)*. Prototypical design patterns conceive a general conceptualization of a set of situations, dominant terms, and corresponding CQs. Furthermore, they highlight candidates for key concepts, called scopes. PODPs have a conceptual or rather architectural nature [25] and arrange the answering of the CQs by scopes according to the categories of the terminology (Actors and Roles, Products and Features, Environment and Situation, Problems and Solutions) (cf. Fig. 1).

The P-Pattern represents solutions to problems concerning the product itself, e.g., its price, whereas the PP-Pattern covers semantics on product bundles and their features. The C-Pattern semantically describes the current context, e.g., time and space of a situation, present objects, and individuals. The PU-Pattern describes information that relates products with user attributes. The

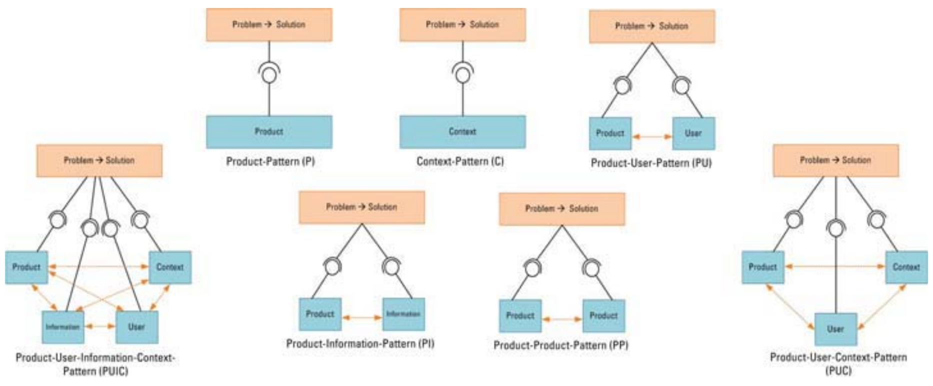


Fig. 1. Design patterns for ambient environments

PUC-Pattern enhances the PU-Pattern with contextual information. Solutions for problems concerning additional information about the product, e.g., user reviews, images, videos, etc. are represented by the PI-Pattern. The most complex pattern, the PUIC-Pattern, represents solutions to problems concerning the matching of information about products to user-specific attributes and needs within a context. All PODPs combine scopes that were extracted from the terminology. The product scope covers the term category *Product and Features* whereas the user scope represents the term category *Actors and Roles*. The context scopes contain terms subsumed by the category *Environment and Situation*. The lollipop relationship between pattern P and pattern C indicate that they conceptually contribute to what is a problem solving situation (cf. Figure 2). More complex PODPs are derived from simpler ones, e.g., pattern PI is an extension of pattern P. Patterns can be composed, which creates new patterns, e.g., the PUIC pattern. By discussion of PODPs, the requirement for an independent information scope appeared that supports CQs such as *"How can I apply this product?"* The information scope refers to information *about* a product, e.g. product images, application videos, and user-generated or professional product reviews.

PODPs dispose a template-based and compact visualization [10], which consists of the following slots:

(a) Graphical visualization of the pattern [19, 10]
(b) Name of the pattern [19, 10]
(c) Description of the intention of the pattern [19, 26]
(d) CQs that are addressed by the pattern [26]
(e) Terms that characterize the pattern
(f) Situations that exemplify a pattern [26]
(g) Consequences, side effects, references to other patterns [19, 10, 26]
(h) Components of the pattern, e.g., product scope and information scope [19, 26]

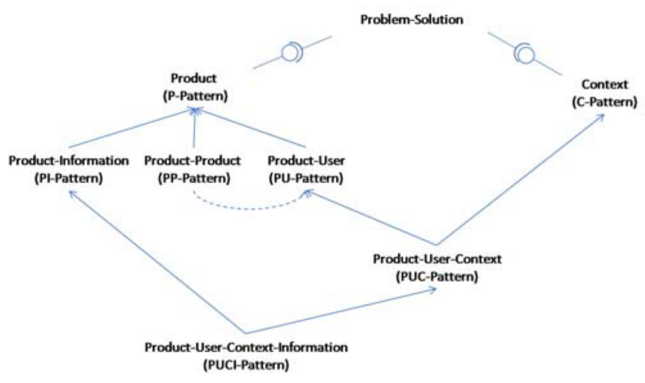


Fig. 2. Tree of design patterns for ambient environments

Slot (e) represents the linkage of the different patterns. For example, the PP-Pattern evolves from the P-Pattern when more than one product is part of a communicative situation within an ambient environment. On the other hand, the PU-Pattern has a strong reference to the P-Pattern representing one product as well as the PP-Pattern focusing on solutions to problems concerning product bundles (cf. Fig. 2).

Terminology Set-up Next, scopes within patterns are refined and structured by arranging relevant terms (cf. Section 3.1) and further analyzing the CQs. For concept identification, a middle-out approach is used that starts with salient concepts and proceeds with generalization and specialization [27]. This seems to be the most effective approach because concepts "in the middle" are more informative about the domain. Table 3 shows the four scopes and an extract of their corresponding terms. The product scope covers all necessary information

Product Scope	**User Scope**	**Context Scope**	**Information Scope**
Name	Characteristic	Space	Animation
Price	Problem	Time	Video
Ingredients	Feeling	Temperature	Image

Table 3. Pattern scopes and exemplary terms

that is part of the product itself, e.g., information about price and material. In contrast, external product information, such as manuals or brochures, is covered by the information scope. The user scope covers terms and questions related to a user whereas the context scope conceives characterizations of a general context.

Mapping PODPs onto ODPs Operations that support the reuse of formal ODPs try to find patterns that optimally cover CQs [16]. In PoNA, PODPs are mapped onto ODPs. Within the cosmetics domain, ODPs proposed by [26] fit well with requirements given by PODPs (cf. Fig. 2). Typically several ODPs are required, which leads to pattern composition. For instance, the ODP mapping of the PI-pattern uses the ODP mapping of the P-pattern based on the *Description and Situation* ODP in conjunction with the information-object ODP [10]. Currently, mapping tasks are executed manually. Our goal is to reduce the complexity of ODP mapping tasks by automatically mapping situations, terminologies, and CQs onto PODPs. With a library of predefined PODP-ODP mappings, we will test ways in which the reuse of formal ontologies can be improved.

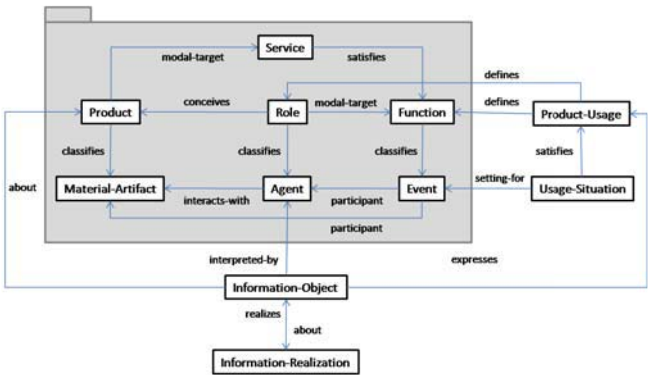


Fig. 3. Merging product and information scopes based on the *Description and Situation* ODP and information-objects ODP

3.3 Implementation Phase

The ODP-based P-pattern and PP-pattern represented in OWL-DL has been integrated into the Smart Product Description Object (SPDO) model that is used in AmI environments [7, 22]. Instances of SPDO models are used as product-centered knowledge bases for Natural Language Processing modules [28] and product reasoning [7]. Currently, we are working on the integration of the other patterns within the EU-project *Interactive Knowledge Stack* (IKS). The resulting SPDO ontology covers all information concerning the product itself. It consists of 25 classes, 86 properties, and 104 restrictions. SPDO answers all the CQs of the P-Pattern and the PP-Pattern. The linkage of two or more product scopes within the PP-Pattern is realized via reasoning based on standardized Web-based rules (SWRL[4]). Statements about alternative or matching products are generated by processing certain concepts of SPDO instantiations while each SPDO describes one particular product.

[4] http://www.w3.org/Submission/SWRL/

3.4 Test Phase

The quality of the resulting ontology is tested with respect to four characteristics: syntactic, semantic, pragmatic, and social quality [17]. First, the semantic quality is evaluated by checking the consistency of the ontology using the Pellet reasoner. Validation of pragmatic quality consists of verifying the coverage of an ontology over a domain and answering CQs. For the cosmetic domain, initial semantic checking of the SPDO showed promising results. Testing the pragmatic quality by answering CQs associated with the P-pattern and PP-pattern was straightforward because users can use the NLP component of SPDO instances for direct communications. Nonetheless, more detailed user studies are required. The syntactic quality is verified within the implementation workflow whereas the social quality can be checked only after application of an ontology in real environments, which is part of the EU-project IKS.

4 Discussion of POnA

When defining the terminology, we found a huge amount of terms with completely different origins, e.g., user or content-specific terms. We therefore decided to structure terms concerning their particular content categories. This additional step was very helpful for creating prototypical ontology design patterns based on scopes. Furthermore, we were able to clearly separate product-centered knowledge from other ontological parts, which is important for AmI environments. Thus, each product could be labled by dedicated semantic product information. Through modularization by scopes and PODPs, we were able to set up a clearly defined ontology engineering process that fits very well with the requirements of AmI environments. To ensure a better portability to other domains and a general representation of the structure of ambient environments, we decided on general scopes within patterns. Domain specifications can be realized by conceptualizations of scopes and PODPs. Additionally, we found that not all scopes can be directly derived from analysis of situations, CQs, and terminologies. At the moment, scope completion requires careful discussions with domain experts. In general, we found that joint consideration of situations, CQs and terminologies is an efficient approach for identifying requirements for ontologies because they help to “pursue the path”.

5 Example

Within this section, the *Product-User-Context-Pattern* is exemplified in detail. This PODP was chosen because product, user and context scopes are quite advanced. Table 4 shows the representation of the pattern in the form of a PODP.

The PUC-Pattern represents solutions to problems concerning the matching of products to user-specific attributes and needs within a context, such as whether a vanishing creme should be matched with a specific skin type in a

Graphical visualization	*cf. Fig. 1*
Name	*Product-User-Pattern (PU)*
Description of Intention	Representation of solutions for problems concerning the matching of products to user-specific attributes and needs
Competency Questions	Extract: (a) Does the product /the application of the product solve my problem? (b) Does the product fit to me? (c) Which product can I use for protecting my lips? (d) Will I be happy applying the product?
Characterizing terms	product: price, ingredients, user: emotions, context: temperature, day of week
Situations	Anna is interested in a vanishing creme and wants to know whether it is right for her skin in humid environments.
Consequences / Side Effects / References	PP-Pattern; PUC-Pattern
Components	product scope; user scope: context scope

Table 4. Template of Product-User-Context-Pattern

humid environment. Furthermore, the PUC-Pattern disposes of cross references to other patterns. For instance, when product bundles (more than one product) have to be matched with user-specific aspects, the PUC-Pattern is extended by the PP-pattern. A mapping of the PUC-PODP onto ODPs is illustrated in Fig. 4.

6 Conclusion and Future Work

Ambient Intelligence implies modularized environments of computing and specific interfaces. The characteristic of such an environment requires systems to deal with large amounts of unstructured information from heterogeneous sources and to support dynamic knowledge sharing and reasoning. These issues imply the application of appropriate knowledge representations. We start from the outset that ontologies are an efficient means for building ambient environments because they enable efficient sharing, adding and changing of information, and inference generation [2]. By leveraging capabilities of different ontology design methodologies, we investigated *Pattern-based Ontology Building Method for Ambient Environments (POnA)* as a method with particular focus on ambient environments. We found that a combination of systematic methodologies and different types of design patterns can be an efficient approach to designing ontologies for ambient environments. The POnA process focuses on the main ontology development processes and enables an explicit pattern-based modularization and abstraction of scopes. In our future work, we will proceed with several tasks: (1)

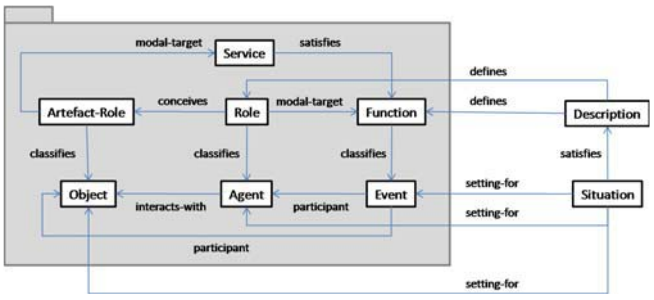


Fig. 4. ODP-based PUC-pattern

evaluation of PODPs and their formalizations based on ODPs, (2) extension of PODPs to more AmI-specific dimensions, i.e., representation of spatio-temporal information, frames of reference, and sensoric perceptions, and (3) automatic mapping of CQs, situations, and terms onto PODPs.

7 Acknowledgements

We would like to thank Andreas Filler and Tobias Kowatsch for their help on this paper. This work is part of the project SmaProN that is being funded by the Federal Ministry of Education and Research (BMBF), Germany, under the number FKZ 17 53X 07.

References

1. Dey, A.K.: Providing architectural support for building context-aware applications. PhD thesis, Georgia Tech College of Computing, Atlanta, GA, USA (2000) Director-Abowd, Gregory D.
2. Coen, M., Phillips, B., Warshawsky, N., Weisman, L., Peters, S., Finin, P.: Meeting the computational needs of intelligent environments: The metaglue system. In: Proceedings of MANSE99, Springer-Verlag (1999) 201–212
3. Schilit, W.N.: A system architecture for context-aware mobile computing (1995)
4. Want, R., Hopper, A., Falcao, V., Gibbons, J.: The active badge location system. ACM Trans. Inf. Syst. **10**(1) (1992) 91–102
5. Peters, S., Shrobe, H.E.: Using semantic networks for knowledge representation in an intelligent environment. In: PERCOM '03: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, Washington, DC, USA, IEEE Computer Society (2003) 323
6. Chen, H., Finin, T., Joshi, A.: The SOUPA ontology for pervasive computing. In Tamma, V., Cranefield, S., eds.: Ontologies for Agents: Theory and Experiences. Springer (2005) 233–258
7. Maass, W., Filler, A.: Towards an infrastructure for semantically annotated physical products. In Hochberger, C., Liskowsky, R., eds.: Informatik 2006. Volume P-94 of Lecture Notes in Informatics., Berlin, Springer (2006) 544–549

8. Minsky, M.: A framework for representing knowledge. In Winston, P.H., ed.: The Psychology of Computer Vision. McGraw-Hill, New York (1975) incollection.
9. Schank, R.C., Abelson, R.P.: Scripts, Plans, Goals and Understanding. Erlbaum, Hillsdale, NJ (1977)
10. Gangemi, A.: Ontology design patterns for semantic web content. In: M. Musen et al. (eds.): Proceedings of the Fourth International Semantic Web Conference, Berlin, Springer (2005)
11. Maass, W., Varshney, U.: A framework for smart healthcare situations and smart drugs. In: SIG-Health Pre-AMCIS Workshop at the 15th Americas Conference on Information Systems (AMCIS 2009), San Francisco, USA (2009)
12. Svátek, V.: Design patterns for semantic web ontologies: Motivation and discussion. In: Proceedings of the 7th Conference on Business Information Systems, Poznan (2004)
13. Corcho, O., Fernández-López, M., Gómez-Pérez, A.: Methodologies, tools and languages for building ontologies: where is their meeting point? Data Knowl. Eng. **46**(1) (2003) 41–64
14. Uschold, M., King, M.: Towards a methodology for building ontologies. In: In Workshop on Basic Ontological Issues in Knowledge Sharing, held in conjunction with IJCAI-95. (1995)
15. Fernández-López, M., Gómez-Pérez, A., Juristo, N.: METHONTOLOGY: from ontological art towards ontological engineering. In: Proceedings of the AAAI97 Spring Symposium Series on Ontological Engineering, Stanford, USA (March 1997) 33–40
16. Suárez-Figueroa, M.C., Blomqvist, E., D´Aquin, M., Espinoza, M., et al., A.G.P.: D5.4.2. revision and extension of the neon methodology for building contextualized ontology networks. Technical report, NeOn: Lifecycle Support for Networked Ontologies (2009)
17. De Nicola, A., Missikoff, M., Navigli, R.: A software engineering approach to ontology building. Inf. Syst. **34**(2) (2009) 258–275
18. Grueninger, M., Fox, M.S.: Methodology for the design and evaluation of ontologies. In: Proceedings of the Workshop on Basic Ontological Issues in Knowledge Sharing held in conjunction with IJCAI-95. (1995)
19. Alexander, C.: The timeless way of building. Oxford University Press, New York (1979)
20. Clark, P., Thompson, J., Porter, B.W.: Knowledge patterns. In Staab, S., Studer, R., eds.: Handbook on Ontologies. International Handbooks on Information Systems. Springer (2004) 191–208
21. Gangemi, A., Guarino, N., Masolo, C., Oltramari, A., Oltramari, R., Schneider, L.: Sweetening ontologies with DOLCE. In: Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web, Springer (2002) 166–181
22. Janzen, S., Maass, W.: Smart product description object (SPDO). In: Poster Proceedings of the 5th International Conference on Formal Ontology in Information Systems (FOIS2008). IOS Press, Saarbrücken, Germany (2008)
23. Grueninger, M., Fox, M.S.: The role of competency questions in enterprise engineering. In: Proceedings of the IFIP WG5.7 Workshop on Benchmarking - Theory and Practice. (1994)
24. Johnson-Laird, P.N.: Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness. Cambridge University Press (1983) book.
25. Gangemi, A., Presutti, V.: Ontology design patterns. http://ontologydesignpatterns.org (2008)

26. Presutti, V., Gangemi, A.: Content ontology design patterns as practical building blocks for web ontologies. In: ER '08: Proceedings of the 27th International Conference on Conceptual Modeling, Berlin, Heidelberg, Springer-Verlag (2008) 128–141
27. Uschold, M., Gruninger, M.: Ontologies: Principles, methods and applications. Knowledge Engineering Review **11** (1996) 93–136
28. Maass, W., Janzen, S.: Dynamic product interfaces: A key element for ambient shopping environments. In: Proc. of 20th Bled eConference, Bled, Slovenia (2007)

Using Semantic Relations for Content-based Recommender Systems in Cultural Heritage

Yiwen Wang[1], Natalia Stash[1], Lora Aroyo[12], Laura Hollink[2], and Guus Schreiber[2]

[1] Eindhoven University of Technology, Computer Science
{y.wang,n.v.stash}@tue.nl
[2] VU University Amsterdam, Computer Science
{l.m.aroyo@cs.vu.nl,hollink,schreiber}

Abstract. Metadata vocabularies provide various semantic relations between concepts. For content-based recommender systems, these relations enable a wide range of concepts to be recommended. However, not all semantically related concepts are interesting for end users. In this paper, we identified a number of semantic relations, which are within one vocabulary (e.g. a concept has a broader/narrower concept) and across multiple vocabularies (e.g. an artist is associated to an art style). Our goal is to investigate which semantic relations are useful for recommendations of art concepts and to look at the combined use of artwork features and semantic relations in sequence. These sequences of ratings allow us to derive some navigation patterns from users, which might enhance the accuracy of recommendations and be reused for other recommender systems in similar domains. We tested the CHIP demonstrator, called the Art Recommender with end users by recommending both semantically-related concepts and artworks features (e.g.*creator, material, subject*).

1 Introduction and Problem Statement

The main objective of the CHIP (Cultural Heritage Information Personalization) project is to demonstrate how Semantic Web and personalization technologies can be deployed to enhance access to digital collections of museums. In collaboration with the Rijksmuseum Amsterdam[3], we have developed the CHIP Art Recommender: a content-based recommender system that recommend art-related concepts based on user ratings of artworks. For example, if a user gives the famous painting "Night watch" a high rating, the user will get its *creator* "Rembrandt" recommended.

The semantic enrichment of Rijksmuseum InterActief (ARIA)[4] database [1] enables the opportunity to recommend a wide range of concepts via different semantic relations. These relations link concepts not only within one vocabulary (e.g. *teacher/studentOf, broader/narrower*), but also across two different

[3] http://www.rijksmuseum.nl
[4] http://www.rijksmuseum.nl/collectie/ontdekdecollectie

vocabularies (e.g. *hasStyle, birth/deathPlace*). For example, if a user likes the artist "Rembrandt", the system could recommend his *teacher* "Pieter Lastman" and his art *style* "Baroque", or even its *narrower* concept "Renaissance-Baroque styles and periods" and its *broader* concept "European styles and periods".

However, for recommender systems, the use of semantic relations also poses a problem. Not all related items are useful or interesting for end users. If the user likes the artist "Rembrandt", besides his teacher and art style, the system could also recommend his *death place* "Amsterdam" or even the broader geographic location "Noord-Holland", which might not be of interest for users. Thus, our main challenge is to find which semantic relations are generally useful for content-based recommendations. Furthermore, we aim to derive the navigation patterns in order to improve the accuracy of recommendations. Our hypothesis is that by choosing specific semantic relations, the recommender system could retrieve more related items without decreasing the accuracy and interestingness. In the experiment, we tested the Art Recommender with end users by applying both artwork features and semantic relations to recommend related concepts. Using artwork features as a baseline, we compared the recommendations via different semantic relations in terms of accuracy and interestingness.

The paper is organized as follows: Section 2 presents related work about the use of semantic relations for recommender systems. Section 3 briefly introduces the metadata vocabularies and identifies a number of semantic relations as well as artwork features. In Section 4 we describe our demonstrator, a content-based art recommender system and explains the design of the experiment. Section 5 discusses the results. We conclude and discuss the future work in Section 6.

2 Related Work

In recent years, many recommender systems have appeared that use Semantic Web technologies, where information is well-defined in an open standard format that can be read, shared and exchanged by machines across the Web [2]. Peis et al [3] classified semantic recommender systems into three different types: (i) vocabulary or ontology based systems; (ii) trust network based systems constructed with FOAF[5]; and (iii) context-adaptable systems that use additional ontologies about e.g. the current time, place of the user. In this paper, we focus on the first type (vocabulary-based recommender systems) and discuss how various semantic relations to enhance recommendations.

Metadata vocabularies or domain ontologies are so far mainly used for content-based recommender systems. the CULTURESAMPO portal [4] recommends images based on semantic relations between selected images and other images in the repository. In particular, they used the *has-part/part-of* relations with a fixed weight to determine the ontological relevance of recommendations. A similar approach is adopted in the ConTag project [5], which extracts similar topics using the *broader/narrower* relations for recommendations. By knowing user

[5] Friend of A Friend: http://www.foaf-project.org/

preferences, Blanco-Fernández [6] inferred semantic associations between user preferences and relevant instances from the domain ontology in order to provide personalized recommendations of TV programs.

In CHIP we have developed a content-based recommender system, the Art Recommender. Compared with the content-based recommender systems mentioned above, the Art Recommender works with four different semantic metadata vocabularies (see Section 3), which provide richer semantic relations: not only hierarchical relations such as *broader/narrower* within one vocabulary, but also more sophisticated relations across two different vocabularies, e.g. *hasStyle* and *birth/deathPlace*. These semantic relations might be helpful to partially solve the cold-start and over-specialization problems for content-based recommender systems. For example, (i) when there are few ratings, the system could use semantic relations to provide additional concepts; (ii) the use of semantic relations within one vocabulary or across multiple vocabularies might retrieve new concepts, which might be surprising or interesting for users.

3 Metadata vocabularies and Semantic Relations

The CHIP Art Recommender currently works with the Rijksmuseum ARIA database, containing images and metadata descriptions of artworks. The mapping of metadata from ARIA to Iconclass[6] and to the three Getty thesauri[7] (the Art and Architecture thesaurus (AAT), the Union List of Artists Names (ULAN) and the thesaurus of geographic Names (TGN)) [1] brings rich semantic structure to the Rijksmuseum collection and creates the opportunity to recommend a wide range of art concepts via various semantic relations. As shown in Fig. 1, we listed 4 basic artwork features (Relations 1-4) which link an artwork and its associated concepts, as well as 11 semantic relations (Relations 5-15), which link concepts within one vocabulary and across two different vocabularies.

Relations 1-4 are artwork features, which have already been implemented in the original Art Recommender for the inference of recommended concepts. As an example, if a user likes the artwork "Night watch", we could recommend the *creator* "Rembrandt" from ULAN, the *creation site* "Amsterdam" from TGN, the *material* "Oil painting" from AAT, the *subjects* "Cloth" from Iconclass and "Wealth in the Republic" from ARIA.

Relations 5-15 are semantic relations linking concepts within one vocabulary and across two different vocabularies. We applied these semantic relations in the experiment in order to get insights in which relations are useful for content-based recommendations. In more detail, Relation 5 (*link:hasStyle*) links an artist to his/her art style(s), across the ULAN and AAT vocabularies, e.g. "Rembrandt" in ULAN has an art style "Baroque" in AAT. Relations 6 and 7 are the *ulan:teacher/studentOf* relations linking two concepts within the ULAN vocabulary. For example, "Rembrandt" is the teacher of "Gerrit Dou" and the student

[6] http://www.iconclass.nl/libertas/ic?style=index.xsl
[7] http://www.getty.edu/research/conductingresearch/

of “Pieter Lastman”. Relations 8 and 9 are the *birth/deathPlace* relations between artists and geographical locations where she was born or died, across the ULAN and TGN vocabularies, e.g. “Rembrandt” in ULAN was born in “Leiden” in TGN, and died in “Amsterdam” in TGN. Relations 10-15 are the general *broader/narrower* relations within the AAT, Iconclass and TGN vocabularies. Although the relations are the same, the types of concepts mapped to the three vocabularies are different: (i) concepts mapped to AAT are mainly art styles, e.g. “Rococo revival” has a broader concept “Modern European revival styles”, (ii) concepts mapped to Iconclass are general subjects, e.g. “Musical” has a narrower concept “Music instruments” and, (iii) concepts mapped to TGN are geographic locations, e.g. “Amsterdam” has a broader concept “Noord-Holland”.

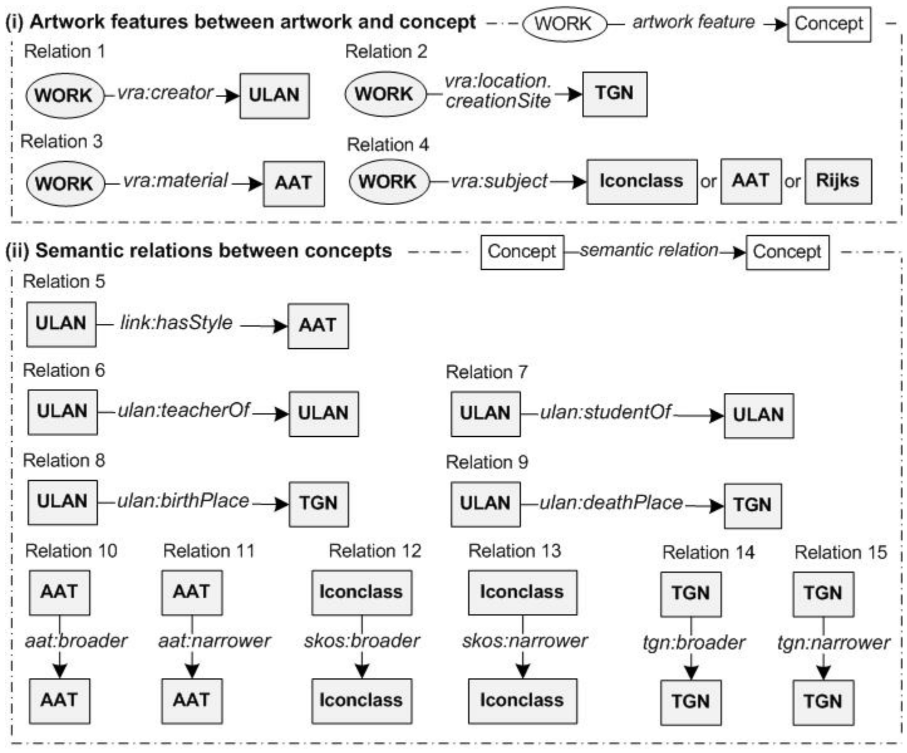


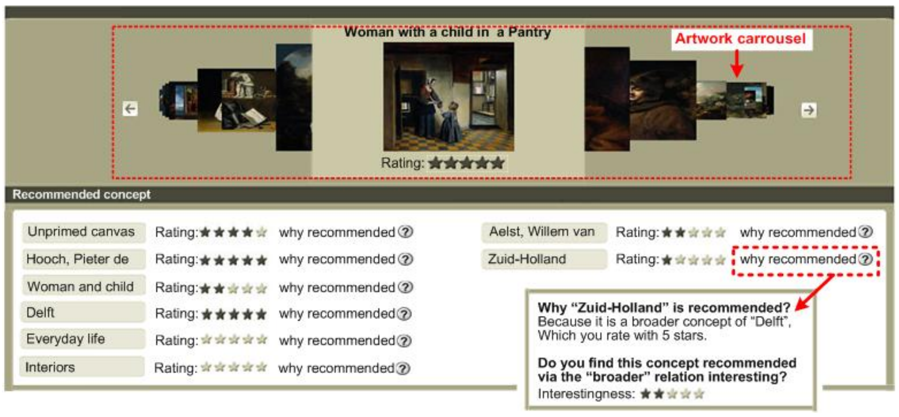
Fig. 1. Overview of artwork features and semantic relations based on the metadata vocabularies

4 Experiment

Our goal is (i) to investigate which semantic relations are useful for content-based recommendations in comparison with standard artwork features, and (ii) to look at the combined use of semantic relations and artwork features in sequence, which might derive some navigation patterns from users in order to enhance the accuracy of recommendations and to be reused for other recommender systems.

4.1 Target System: The Art Recommender

To address these goals, we applied both artwork features and semantic relations for content-based recommendations of art concepts in the Art Recommender[8]. Considering artworks are recommended based on related/recommended art concepts, in order to get a clear insights, we only looked at how semantic relations and artwork features influence related/recommended art concepts in this experiment. We leave the exploration of how they affect related artworks for recommendations as a next step in future work.



Why "Zuid-Holland" is recommended?
Because it is a broader concept of "Delft",
Which you rate with 5 stars.

Do you find this concept recommended via the "broader" relation interesting?
Interestingness:

Fig. 2. User interface of the Art Recommender in the experiment

The user interface of the Art Recommender (see Fig. 2) was split in two parts: the upper part is the rating dialog with a slide show of artworks, which allows the user to browse artworks from the collection and give ratings to them with 1-5 stars (i.e. I hate it, I dislike it, neutral, I like it, and I like it very much). In the bottom part recommended concepts are shown, based on the ratings given by users to the artworks in the upper part. Then the user rates (with 1-5 stars) the recommended concepts shown in the bottom part to express how much she likes each recommendation. The list of recommended concepts will be dynamically updated based on the last rating given for an artwork or concept.

[8] http://www.chip-project.org/demoUserStudy3/

In addition, in the "Why recommended" option (see Fig. 2), an explanation is provided about which feature or relation was used for each recommendation. The user is then asked to give 1-5 stars indicating how interesting she finds the concept recommended via this feature or relation (*interestingness*). This dimension of interestingness puts the recommended concept back in context, which helps user to understand the inference of recommendations by using particular artwork feature(s) or semantic relation(s).

4.2 Method

At the beginning of each session, participants were asked to fill out a questionnaire about: (i) their age, (ii) whether they are familiar with the Rijksmuseum collection, (iii) experience with recommender systems in general, (iv) expectation from art recommendations, and, (v) for what purpose they will use art recommendations.

After completing the questionnaire, we briefly introduced the Art Recommender and explained the recommendation process. Using the Art Recommender, users were asked to follow two steps:

Step 1 (Pre-task): to find an artwork that she likes from the artwork slide show (to start the process the user needs to give a rating of either 4 or 5 stars; the recommender does not start-up with negative ratings). As a baseline, it will recommend the first set of related art concepts by applying artwork features based on the rated artwork.

Step 2 (Main task): to rate the first set of recommended concepts. Based on the ratings of concepts, the system will produce a second/new set of recommended concepts by applying semantic relations, which also allows users to rate. At any point for each recommendation the user can click on "Why recommended" and give her feedback on whether she finds this recommendation via the particular artworks feature or semantic relation interesting or not on a 5-degree scale. Step 2 gave us an insight in the performance of the concepts recommended via semantic relations in comparison with the concepts recommended directly via artwork features.

Users were asked to repeat this process for at least 5 times in order to rate enough recommended concepts via either artwork features or semantic relations. At any point, the user could stop rating recommended concepts and go to select another artwork from the slide show. Then the same process is repeated for each rated artwork.

4.3 Dimensions and Metrics

Using artwork features as a baseline, we tested the results of recommended concepts via semantic relations in terms of two dimensions: *accuracy* and *interestingness*.

- *Accuracy*: by directly asking the user whether she likes this recommended concept, which is shown as "Ratings" in the Art Recommender in Fig. 2.

– *Interestingness*: by giving the explanations of “Why recommended”, it asks the user whether she finds the concept recommended via the particular artwork feature or semantic relation interesting.

Precision, Recall and Mean Absolute Error (MAE) are most popular metrics to evaluate recommender systems [7, 8] and to measure the usefulness of semantic relations in query expansion for information retrieval systems [9–11]. *Precision* represents the probability that a recommended item is relevant, *Recall* represents the probability that a relevant item will be recommended, and *MAE* measures the average absolute deviation between a predicted rating and the users true rating [8].

However, in our case, we could only apply *precision*, but not *recall* and *MAE*. Because it is difficult to determine the total number of relevant items. As Burke discussed in [7], relevance is subjective from an end user's standpoint and it often changes when the user gets explanations for recommendations. As Herlocker discussed in [8], it is also not appropriate in our case to use *MAE*, where a list of recommended concepts is returned but users often only view concepts that she is interested and cares about errors in concepts that are recommended. Thus in the experiment we only use precision to measure accuracy and interestingness for recommended art concepts. To divide the concepts into relevant or irrelevant concepts, we defined a threshold value on the used 5-star scale, which converts 4 and 5 stars to “relevant” and 1-3 stars to “not relevant”. In terms of accuracy, relevant concepts refer to the recommended concepts that the user likes with 4 and 5 stars, and in terms of interestingness, relevant concepts refer to the recommended concepts that the user finds interesting with 4 and 5 stars. Below we explain how we calculate it:

$$Precision = \frac{Correct\ Hits}{Total\ Rec.Rated}$$

Correct Hits is the total number of relevant concepts that are recommended by the system and have been rated by the user with 4 and 5 stars in terms of accuracy and interestingness respectively.

Total Rec.Rated is the total number of concepts that are recommended by the system and have been rated by the user with 1 to 5 stars in terms of accuracy and interestingness respectively. *Total Rec.* is the number of all recommended concepts with or without user ratings. To avoid the data sparsity problem [7] (i.e. the number of recommended items far exceeds what a user can rate), we only use the number of “Total Rec.Rated” to compute the precision and we do not include the number of “Total Rec.”, because we do not have user feedback on concepts without ratings [8]. However, we will provide the number of “Total Rec.” (in Table 1) to get an idea of how many concepts could be recommended via an artwork feature or a semantic relation.

5 Results

In a period of three weeks, in total 48 users participated. The experiment took about 20-35 minutes per person. Each user gave on average 53 ratings for artworks and concepts. Below we describe the participants characteristics collected with the questionnaire.

– *Age*: in the categories of 20-30 years old (65%) and 30-40 years old (21%)
– *Familiar with the Rijksmuseum collection*: not familar with the collection (27%) and a bit familiar with the collection (46%)
– *Experience with recommender systems in general*: every few months using recommender systems, such as Amazon.com (68%)
– *Expectation from art recommendations*: want to get accurate art recommendations that match their art preferences (79%) and interests (83%)
– *For what purpose will use art recommendations*: want to keep up-to-date with new information about artworks/concepts (93%), to reflect on what has been seen in the museum (75%), and to understand her art interests better (79%)

Table 1. Experiment results for artworks features and semantic relations

Nr.	Artwork features/ Semantic relations	Total Rec.	Accuracy			Interestingness		
			Total Rec.Rated	Correct Hits	Precision	Total Rec.Rated	Correct Hits	Precision
Artwork features								
1	vra:creator	332	111	74	0.67	97	80	0.82
2	vra:location.creation Site	182	83	33	0.40	61	34	0.56
3	vra:material	159	92	39	0.43	47	21	0.45
4	vra:subject	3245	1054	528	0.50	768	453	0.59
1-4	all artwork features	3918	1340	674	0.50	973	588	0.60
Semantic relations								
5	link:hasStyle	82	38	24	0.63	46	34	0.73
6	ulan:teacherOf	291	135	57	0.43	127	90	0.71
7	ulan:studentOf	92	55	24	0.44	67	46	0.68
8	ulan:birthPlace	184	44	14	0.32	48	21	0.43
9	ulan:deathPlace	130	42	11	0.26	55	14	0.25
10	aat:broader	69	23	12	0.53	19	11	0.60
11	aat:narrower	125	31	17	0.55	26	16	0.62
12	skos:broader	404	224	112	0.50	131	67	0.51
13	skos:narrower	1198	506	263	0.52	425	213	0.50
14	tgn:broader	82	22	5	0.22	15	2	0.15
15	tgn:narrower	1204	35	6	0.16	23	3	0.13
5-15	all semantic relations	3861	1155	524	0.45	1007	533	0.53

Table 1 gives an overview for artwork features and semantic relations. We calculated: (i) Total number of recommended concepts, (ii) total number of recommended and rated concepts, (iii) correct Hits (recommended and rated with 4 or 5 stars); and, (iv) precision for *accuracy* and *interestingness* respectively.

As a baseline, artwork features provide in total 3918 recommended concepts and reach an average precision of 0.50 for accuracy and 0.60 for interestingness. In comparison, semantic relations bring 3861 new recommended concepts and reach an average precision of 0.46 for accuracy and 0.53 for interestingness, which are only slightly lower than artwork features. For the individual artwork features and semantic relations, we found that:

(i) Artwork feature *vra:creator* and semantic relations *link:hasStyle* and *aat:broader/narrower* produce the most accurate recommendations and they are also the most interesting relations from the users' point of view. This could be explained by observing that artist and art style (concepts in ULAN and AAT) are intrinsically related to the artworks and an important reason why people might like an artwork or related artworks.

(ii) Semantic relations *ulan:birth/deathPlace* and *tgn: broader/narrower* that recommend geographic locations perform very badly. In particular, the *tgn:broader/narrower* relations have the least values for accuracy and interestingness. To understand why *tgn:broader/narrower* and *ulan:birth/deathPlace* relations perform "so badly", we looked at the experiment data in detail. For example, many users like the artist "Rembrandt", however, in most cases they found his birth place "Leiden" and his death place "Amsterdam" not relevant. In comparison, users like recommended concepts such as his art styles, his teacher(s) and students(s). Another example, "Utrecht" is also a popular concept often rated with high scores, but its narrower location "Vianen" is always rated as a not-relevant concept, since it is unfamiliar to most users. This suggests that, for art recommendations, semantic relations *tng:broader/narrower* and *ulan:birth/deathPlace* might not be useful or interesting for users because they are not intrinsically related to artworks but only to locations or artists. This might also explain why users rarely rated locations recommended via these relations (with a low number of *Total Rec.Rated*). In comparison, artwork feature *vra:creationSite* gives much better results, probably it is more related to artworks.

(iii) Artwork feature *vra:subject* and semantic relations about subjects *skos:broader/narrower* produce the largest number of recommended concepts and correspondingly resulted in most user ratings. With respect to accuracy and interestingness, they score on the average.

To explore potential correlations between accuracy and interestingness, in Fig. 3, we plotted these two dimensions for artworks features and semantic relations. Interestingly, there is a strong positive correlation between accuracy and interestingness (Peason's R=0.89, p value<0.01). This means that for an artwork feature or semantic relation, the more accurate recommendations it produces, the more interesting users find the recommendations, and vice-versa. An exception here is the semantic relation *ulan:teacher/studentOf.* As shown in Table 1, although the accuracy precisions for these two relations are slightly lower (0.43, 0.44) and the interesting precisions for them are very high (0.71, 0.68). This explains why semantic relations could partially solve the over-specialization problem (see Section 2) by recommending surprising or interesting items, even though the recommendations are not always quite accurate.

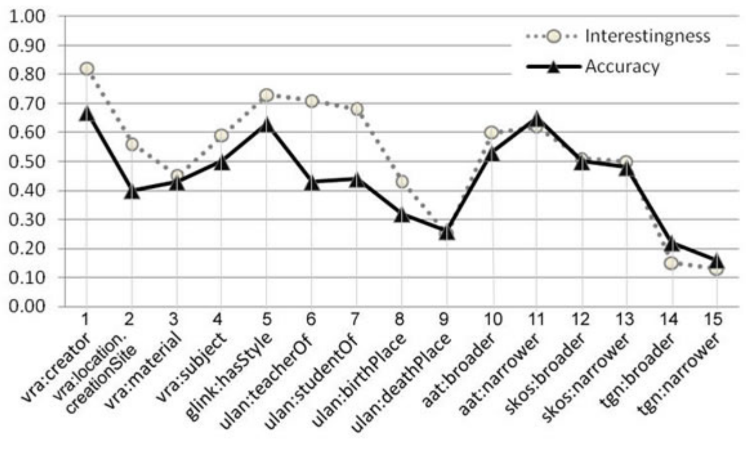


Fig. 3. Correlation between accuracy and interestingness

The setup of the experiment gives us an opportunity to look at the combined use of artwork features and semantic relations in sequence. As explained in Section 5, every positively rated artwork/concept resulted in a new set of recommended concepts that the user could rate. In theory this process can go on until no new recommendations are found, but in practice most users stopped after three or four steps [9]. These sequences of ratings allow us to examine the quality of recommendations based on sequences of semantic relations and artwork features.

We first removed all sequences for which we have less than 10 user ratings. From our previous user studies [12], 10 ratings seems to be a minimum to get a reliable estimate of the quality of recommendations. We then calculated the mean of accuracy precision and interestingness precision (P_{mean}) for the remaining features and relations. Fig. 4 shows the sequences of recommended concepts that received more than 10 ratings, and their P_{mean} values at each step. From Table 1, we can calculate that the P_{mean} is 0.55 for all artwork features and 0.49 for all semantic relations. Using these two values as references, in Fig. 4 we highlighted artwork features (used in Step 2) that have a P_{mean} greater than 0.55 in black and semantic relations (used in Step 3 and 4) that have a P_{mean} greater than 0.49 in grey. Interestingly, we found three potentially useful navigation patterns of combined artwork feature and semantic relations:

– artwork -> creator -> style -> broader/narrower styles
– artwork -> creator -> teacher/student -> styles
– artwork -> subject -> broader/narrower subjects

We observe that all three patterns show a decrease of P_{mean} in each step, which might be due to the fact that the concepts are gradually more remote from the artwork. The only exception is Step 4 in Pattern 2 (from teachers and

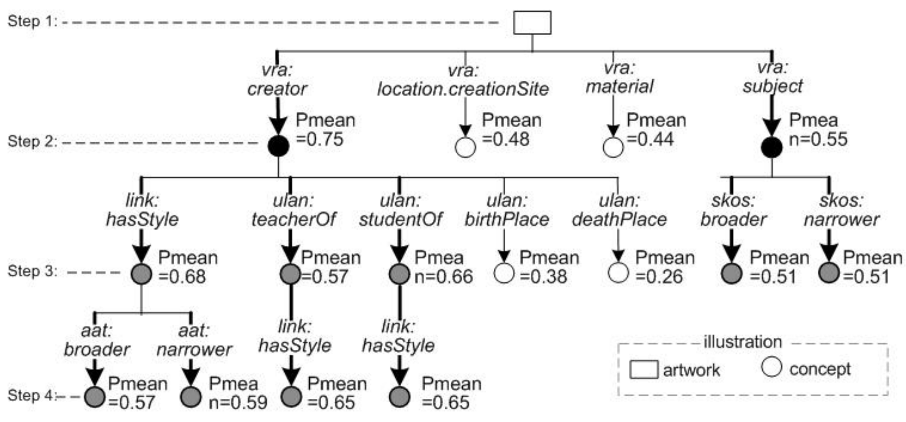


Fig. 4. Combining artwork features and semantic relations in sequence

students to art styles). Still, at each step in the patterns, the P_{mean} value remains relatively high above the average. The three patterns could potentially be used to recommend remotely linked concepts without asking users' explicit feedback/ratings on each step. For example, if a user likes the artwork "Night watch", following the second pattern, it could recommend concepts "Rembrandt" (*creator*), "Pieter Lastman" (*teacher*), "Renaissance" (*the teacher's art style*), "Gerrit Dou" (*student*), and "Baroque" (*the student's art style*), without explicitly asking the user to rate "Rembrandt", "Pieter Lastman" and "Gerrit Dou".

6 Discussion and Future Work

Metadata vocabularies provide rich semantic relations that can be used for recommendation purposes. We examined the performance of both semantic relations and artwork features with the content-based CHIP Art Recommender in terms of accuracy and interestingness. Our results demonstrated that artwork features (*vra:creator*) and semantic relations (*ulan:teacher/studentOf, link:hasStyle*) that recommend concepts in the ULAN and AAT vocabularies produce the most accurate recommendations and also give the most interesting recommendations from the users' point of view. This might be due to the fact that these artwork features and semantic relations which recommend concepts in domain-specific vocabularies are closely related to the domain of art. In comparison, semantic relations considering geographic locations in TGN (e.g. *tgn:broader/narrower, ulan:birth/ deathPlace*) score very low on both accuracy and interestingness. A similar observation applies to the TGN vocabulary, which is a relatively much more general vocabulary and not related to the art domain in comparison with the ULAN and AAT vocabularies.

Based on the performance of individual semantic relations and artwork features, we derived optimal navigation patterns of combined features and relations with multiple intermediate concepts. These patterns can potentially be used to effectively recommend indirectly linked concepts without asking the user's explicit feedback for the intermediate concepts.

Generalizing, we found that vocabularies which are relatively close to the domain are usually more useful for content-based recommendations than vocabularies, which are more general. In particular, for recommender systems in the domain of art, ULAN and AAT vocabularies which contain concepts about artists and art styles proved to be more useful for art recommendations than the TGN vocabulary which contains concepts about geographic locations. In summary, we may conclude that the use of specific semantic relations can enhance content-based recommendations by (i) retrieving more related concepts, which partially solves the cold-start problem; (ii) providing more interesting or surprising recommended concepts by using combinations of artwork feature and semantic relations, which partially solves the over-specialization problem.

As the preliminary results, the three navigation patterns we derived from the experiment might be very interesting for both users and recommender systems in similar domain of art. For future work, we are primarily interested in association rule mining and decision trees that may produce optimized results. For example, some internal nodes of the presented patterns may be pruned.

In addition, we plan to investigate the weights for different semantic relations based on the user ratings collected from the experiment. These weights can be used in computing predicted values for recommended concepts. For example, if a user likes "Rembrandt", recommendations of his *student* "Gerrit Dou", his *art style* "Baroque" or his *death place* "Amsterdam" would receive different predicted values based on the different weights of the semantic relations. The predicted values of recommended concepts can then be used to determine the predicted values for recommended artworks. In this way, we will gain insights about how the various semantic relations influence both recommended concepts and artworks. Inspired by the work from Mobasher [13], Ruotsalo and Hyvönen [4], the weight for each relation should not be a fixed value but a dynamic value which is calculated according to several factors, e.g. the relevance of a concept with respect to an artwork *TD-IDF* [14], the times of user ratings of a particular artwork or concept, the semantic distance or similarity between two concepts by using latent semantic index (LSI) [15], etc.

Our findings about which semantic relations are most beneficial to recommendations and our future work about applying weights for various relations could also be used for collaborative filtering recommender systems. For example, Mobasher's work [13] shows that well-selected semantic relations can be used to populate related items in order to compute the similarity between users for collaborative filtering recommender systems. This might be helpful to partially solve the cold-start and sparsity problems for recommender systems in general. Following this direction, we could apply the method of calculating the weights for various semantic relations in the recommender system and try different recommendation strategies (e.g. content-based, collaborative filtering and the hybrid approach) in order to compare the quality of recommendations in a large scale quantitative experiment.

7 Acknowledgments

The CHIP project is funded by the Dutch Science Foundation funded program CATCH[9] (Continuous Access to Cultural Heritage). We would further like to thank all the participants in the experiment.

References

1. Wang, Y., Stash, N., Aroyo, L., Gorgels, P., Rutledge, L., Schreiber, G.: Recommendations based on semantically-enriched museum collections. Journal of Web Semantics (2008)
2. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. (2001)
3. Peis, E., del Castillo, J.M.M., Delgado-Lopez, J.A.: Semantic recommender systems. analysis of the state of the topic. In: Proc. of Hipertext. (2008)
4. Ruotsalo, T., Hyvönen, E.: A method for determining ontology-based semantic relevance. In: DEXA. (2007)
5. Adrian, B., Sauermann, L., Roth-Berghofer, T.: Contag: A semantic tag recommendation system. In: Proc. of the New Media Technology and Semantic Systems. (2007)
6. Blanco-Fernández, Y., Arias, J.J.P., et al.: Semantic reasoning: A path to new possibilities of personalization. In: Proc. of ESWC. (2008) 720–735
7. Burke, R.: Hybrid recommender systems: Survey and experiments. Journal of User Model. User-Adapt. Interact. **12**(4) (2002) 331–370
8. Herlocker, J., Konstan, J., Borchers, A., Riedl, J.: Evaluating collaborative filtering recommender systems. Journal of ACM Transactions on Information Systems (2004)
9. Hollink, L., Schreiber, G., Wielinga, B.J.: Patterns of semantic relations to improve image content search. Journal of Web Semantics **5**(3) (2007) 195–203
10. Navigli, R., Velardi, P.: An analysis of ontology-based query expansion strategies. In: Proc. of Machine Learning Conference. (2003)
11. Tudhope, D., Binding, C., Blocks, D., Cunliffe, D.: Query expansion via conceptual distance in thesaurus indexed collections. Journal of Documentation (2006)
12. Wang, Y., Aroyo, L., Stash, N., Rutledge, L.: Interactive user modeling for personalized access to museum collections: The rijksmuseum case study. In: Proc. of User Modeling Conference. (2007)
13. Mobasher, B., Jin, X., Zhou, Y.: Semantically enhanced collaborative filtering on the web. Journal of Web Mining: From Web to Semantic Web (2004)
14. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley, ACM Press (1999)
15. M.W. Berry, S.D., Brien, G.O.: Using linear algebra for intelligent information retrieval. Journal of SIAM Review (1995)

[9] http://www.nwo.nl/catch

Refining Ontologies by Pattern-Based Completion

Nadejda Nikitina and Sebastian Rudolph and Sebastian Blohm

Institute AIFB, University of Karlsruhe
D-76128 Karlsruhe, Germany
{nikitina, rudolph, blohm}@aifb.uni-karlsruhe.de

Abstract. Constructing richly axiomatized ontologies for real-world knowledge-intensive applications is a time-consuming and difficult task. For this reason, the future relevance of ontologies in practice depends on the availability of advanced semi-automatic methods for ontology learning and refinement. In this paper we propose a method to enrich ontologies with complex axiomatic information by completing partial instantiations of ontology design patterns.

Keywords: Ontology design patterns, ontology refinement.

1 Introduction

Richly axiomatized ontologies are essential for powerful, knowledge-intensive applications, since they allow the application of advanced reasoning. However, ontology modeling and construction is a difficult and time-consuming task. Considering the fact that real world applications require large-scale knowledge bases, there is a need for automatic methods to support ontology construction. Unfortunately, automatically constructed ontologies tend to lack expressivity, e.g., axiomatic information about transitivity or symmetry of relations.[1] For example, the relations "before" and "after" found in DBPedia[2] are not specified to be transitive while transitivity would clearly be an expected characteristic of these relations. This kind of information is difficult to acquire from unstructured resources, since it often does not explicitly occur in text, but can only be deduced using external information sources. However, we think that exploiting additional information sources can help to bring forward the ontology enrichment.

As pointed out in [1], typical conceptual patterns arise during the design of ontologies for different domains and different tasks. Ontology design patterns [2] – modeling solutions to solve a recurrent ontology design problem – were introduced to support the reuse of formalized knowledge. We consider the ontology design patterns as a potential source to be exploited for ontology refinement.

The usefulness of ontology design patterns in semi-automatic ontology construction has already been demonstrated in [3], where they have been used to put automatically constructed ontology elements in context by extending the ontology

[1] Relations are also referred to as properties in related literature
[2] http://dbpedia.org/page/Angela_Merkel

with abstract concepts and relations. In this paper, we aim at the refinement of ontology's axiomatization using highly axiomatized knowledge contained in ontology design patterns. The key idea of the proposed approach is to search for components within an ontology which partially instantiate a given ontology design pattern. In this way, potential missing ontology elements can be identified.

If we consider the previously mentioned relations "before" and "after" contained in DBPedia as well as the ontology design pattern "precedence" introduced in [4] and if we assume the ontology part including these two relations to be a partial instantiation of the given ontology design pattern, we see that there are three axioms missing in DBPedia – the axioms expressing the inverseness of the relations "before" and "after" and their transitivity.

The ability to automatically recognize partial instantiations of an ontology design pattern would therefore allow for checking an ontology for potential missing elements and based on the outcome to automatically generate a list of suggestions for a refinement. In this way, using frequently occurring and richly axiomatized ontology design patterns as input could help to add a considerable amount of axioms to a sparsely axiomatized ontology.

In order to automatically recognize an ontology design pattern by the means of an algorithm, a set of indicative features of this pattern is required. We identify the instantiations of ontology design patterns by their structure and the meaning of their elements expressed by axioms and lexical characteristics of each element. Our matching algorithm is based on these types of features. In this paper, we presume an extended kind of ontology design patterns which contain additional lexical information. In the following, we are going to use the expression *ontology pattern* or *pattern* instead of ontology design pattern.

Our method is mainly independent from the employed concrete ontology representation language. However, we presume that the underlying ontology representation language of concerned ontologies supports complex axiomatizations.

The remainder of this paper is organized as follows: The next Section describes research work related to this paper. Our algorithm is introduced in Section 3. Section 4 summarizes and gives an outlook to further research.

2 Related Work

Semi-automatic ontology construction and refinement has been addressed by several approaches relying on different types of data sources. However, only few of them aim at the acquisition of complex axioms going beyond the modeling capabilities of RDFS.

There is a range of methods exploiting the information contained within natural language texts in order to acquire additional axioms (for an overview see [5]). [6] proposes a method for an axiomatization of glossaries such as WordNet based on parsing and converting of natural language descriptions into formal definitions. [7] also aims at the acquisition of complex axioms by the means of deep syntactic analysis of natural language definitions.

There is a range of approaches relying on multiple data sources such as [8] which aims at the acquisition of a particular type of axioms, namely disjointness axioms, by gathering syntactic and semantic evidence from different data sources. RELExO [9] combines learning complex class descriptions from textual definitions with the FCA-based technique of relational exploration in order to clarify the subclass relationship of concepts of an ontology. It generates hypotheses about class extension relationships which cannot be deduced or denied using the axioms already contained in the ontology. Then, it looks for counterexamples in the set of instances contained in the ontology and, if none could be found, it asks the expert to provide a counterexample or to approve the suggested hypothesis. RoLExO [10] relies on the same type of user interaction and hypothesis verification, but generates hypotheses about complex domain-range restrictions. A method proposed in [11] is another example of extracting hypothetical domain axioms based on a given set of entities. These approaches are complementary to ours, since they rely on other sources of information to acquire complex axioms.

Blomqvist [3] proposes a framework for pattern-based semi-automatic ontology construction and refinement. This work focuses on the refinement of 'lightweight' ontologies concerning the logical complexity and expressiveness, which are not intended to obtain a rich axiomatization. For this reason, ontology patterns are not used to enrich the ontology with complex axioms, but to put the automatically learnt ontology elements into context by connecting them with the more general concepts and relations of the pattern.

To the best of our knowledge we are the first to address the general use of ontology design patterns for semi-automatic enrichment of ontologies with complex axioms.

3 Matching Ontologies and Ontology Patterns

The proposed method is based on ontology matching. Matching of ontologies has been widely covered in literature. An overview of the existing approaches can be found in [12]. We use a modified ontology matching technique due to the specific requirements for matching ontology patterns with ontologies. The main particularity of pattern matching is the high average level of abstractness characteristical for the concepts of a pattern. The concepts contained in a pattern are usually abstract enough to match many different concepts in an ontology. Therefore, relations are often the major indicators for a pattern instantiation. Especially lexical information about relations is essential for a better performance of the matching algorithm. Thus, we consider it useful to invest additional effort to a-priori enrich patterns with lexical information. Our algorithm is designed to exploit provided additional lexical information.

Before presenting the algorithm, we state the underlying criteria for a high likelihood of pattern realization by an ontology part. Thereby, we reduce the problem of identifying partial instantiations of a pattern to the problem of identifying complete pattern instantiations. We rely on the following set of criteria:

A part O of an ontology ONTOLOGY is a potential instantiation of the considered ontology pattern P, if its structure can be matched completely with the structure of P in a way that

1. *Each concept C_P contained in P has exactly one corresponding concept C_O in O, which is equivalent to C_P or a subconcept of it;*[3]
2. *Each relation R_P contained in P has exactly one corresponding relation R_O in O, so that domain and range concepts of R_O are the correspondents (according to 1) of the domain and range concepts of R_P and R_O implies R_P;*
3. *Each axiom A_P of P can be deduced from ONTOLOGY when its concepts and relations are replaced by their correspondents according to 1 and 2.*

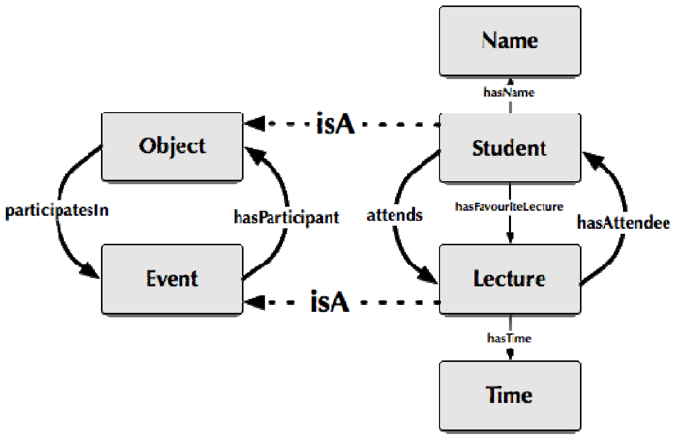


Fig. 1. Matching of an ontology part and a pattern

Figure 1 shows how the content ontology pattern "Participation" is matched with a part of an example ontology according to the stated criteria. The concept "Student" is a specific "Object", the concept "Lecture" is a specific "Event", and thus they correspond as required by condition 1. Relations "hasParticipant" and "hasAttendee" as well as "participatesIn" and "attends" are expressed by synonymous expressions and their domain and range concepts correspond to each other (condition 1). For this reason they imply each other and correspond to each other as required by condition 2. If the pattern also contains an axiom declaring "participatesIn" to be the inverse relation of "hasParticipant", then according to condition 3 it must be possible to deduce it from the set of ontology axioms. In this case, an axiom declaring "attends" to be inverse to "hasAttendee" would suffice.

The criteria stated above provide the basis for our matching algorithm. It uses lexical properties of relations and concepts to verify whether concepts and relations correspond to each other as required in 1 and 2. We will briefly describe the matching of ontology elements based on lexical properties in the following subsection before discussing the algorithm in more detail.

[3] Equivalence, subconcept and implication relationships are correspondences established by the ontology matching method introduced later on.

3.1 Matching Lexical Properties

The goal of the lexical matching is to determine whether a concept is equivalent to or a subconcept of a particular concept and a relation is implied by a particular relation. For this purpose, we use the lexical information contained in ontologies and rely on the availability of particular lexical information in patterns. In the following, we describe these kinds of information.

Lexical information contained in ontologies differs in its detailedness and purpose. We distinguish between a label and a linguistic pattern (LP). A label of an element is a string used as a name for a concept or a relation whereas a LP is used to recognize the instances of a concept or a relation in text. LPs can range from simple regular expressions to more complex structures enriched with different kinds of linguistic information such as concept's part-of-speech type. Even though LPs would be very useful for matching due to their potential richness, the representation of LPs is not standardized in widely used ontology representation languages such as OWL. Therefore, we do not consider LPs in our approach and use only the labels of elements.

For the verification of conditions 1 and 2 using labels, we rely on a list of synonyms and hyponyms for each pattern concept and a list of synonyms and troponyms[4] for each pattern relation. We match each synonym and hyponym or troponym with each label of the potentially corresponding pattern element in the ontology based on string-similarity.

3.2 Matching Algorithm

The matching algorithm in its simplified form can be stated as shown in Fig. 2 and Fig. 3. The algorithm receives an ontology and an ontology pattern as input and generates a list of pattern instantiations as output. It first identifies pairs of lexically matching ontology and pattern elements. Then, to avoid unnecessary computations, it selects the pattern element, which has the fewest lexical matches in the ontology. Since the pattern can only be matched as long as all of its elements have a corresponding element in the ontology, considering only the occurrences of the pattern element with the fewest number of correspondents assures that the least number of ontology parts is analyzed. Each occurrence of the selected element is then analyzed using the recursive procedure growAlignments starting with the given pair of matched elements.

Due to possible hyponymy or troponymy between the elements of the pattern and the ontology, several valid alignments are possible. For a particular initial partial alignment the outcome can differ depending on the order in which elements are matched. For this reason, the algorithm tries all possible ways to construct an alignment and gathers all valid alignments.

[4] Troponyms are expressions for more restrictive relations

```
Algorithm 1 The alignment algorithm
Require: ontology $O$, ontology pattern $P$
  result $\leftarrow \emptyset$ {result: set of valid alignments}
  for all $p \in P$ do
    occurrences $\leftarrow \emptyset$
    for all $o \in O$ do
      if lexicalMatchingSuccessful($p$, $o$) then
        occurrences $\leftarrow$ occurrences $\cup\{(p, o)\}$
      end if
    end for
  end for
  $l \leftarrow$ leastFrequentPatternElement(occurrences)
  $A \leftarrow \emptyset$
  for all $c$ with $(l, c) \in$ occurrences do
    $A \leftarrow A\cup$ growAlignments($\{(l, c)\}$)
    for all $a \in A$ do
      if axiomaticMatchingSuccessful($a$) then
        result $\leftarrow$ result $\cup a$
      end if
    end for
  end for
  return result
```

Fig. 2. Alignment algorithm

```
Algorithm 2 growAlignments is a recursive procedure
Require: $a$: partial alignment
  result $\leftarrow \emptyset$
  markAllAlignedElementsAsGreen($a, O, P$)
  for all $p_g \in$ greenPatternElements do
    $o_g \leftarrow$ ontologyCorrespondentOf($p_g, a, O$)
    for all $p_r \in$ redNeighboursInPattern($p_g, P$) do
      for all $o_r \in$ redNeighboursInOntology($o_g, O$) do
        if $(p_r, o_r) \in$ occurrences then
          result $\leftarrow$ result $\cup$ growAlignments($a \cup \{(p_r, o_r)\}$)
        end if
      end for
    end for
  end for
  return result
```

Fig. 3. Recursive procedure growAlignments

It marks the already matched pattern and ontology elements green and the remaining elements red. For each green pattern element *A* it calculates the remaining red neighbors and matches each of them with the remaining red neighbors of the ontology element corresponding to *A*. If the lexical matching was successful for a pair of elements, they are included into the current alignment which forms the input for another run of the described procedure. The resulting alignments are gathered in a set.

After collecting all valid alignments for the currently analyzed pattern occurrence, axiomatic matching is applied to each alignment to verify that axioms of the ontology pattern can be deduced from the axioms of the ontology, if concepts and relations in the pattern axioms are replaced by the concepts and relations of the ontology. For this task, a state-of-the-art reasoner such as Pellet[5] or HermiT[6] can be used.

4 Ontology Refinement Based on Partial Pattern Instantiations

The algorithm presented above can be used to find partial pattern instantiations by separating pattern elements into obligatory and optional elements and applying the algorithm to the set of obligatory pattern elements. Assuming the availability of a set of richly axiomatized patterns containing additional sets of synonyms and hyponyms for each concept and relation, the ontology engineer can compose a list of patterns for the ontology refinement by choosing the whole set at once or selecting some patterns manually if he or she only needs a particular type of patterns.

For each pattern in this list, the ontology engineer can select the obligatory elements and the level of accuracy, which is the acceptable extent of pattern incompleteness, expressed as the number of pattern elements relative to the total number of pattern elements. The level of accuracy can be set for each pattern or for the whole refinement process. It allows to limit the required user interaction and at the same time to influence the matching performance towards a higher recall or a higher precision. The ontology engineer can also use the default settings. Per default, the level of accuracy is greater zero, which allows considering potential pattern matches containing at least one pattern element. The default obligatory elements are the concepts and relations of each pattern. Axioms are however optional.

After the setup, the algorithm is run for each of the selected patterns. Found alignments are checked for axiomatic incompatibility with the optional pattern elements in order to avoid refinement suggestions which result in an inconsistent ontology. Finally, for each pattern, a list of refinement suggestions is generated and presented to the ontology engineer, who can select some suggestions for the integration into the ontology and start the automatic integration process.

During the integration, partial alignments are used to integrate the unmatched pattern elements into the ontology. Thereby, matched elements themselves are not integrated, but are replaced in axioms by their correspondents before integrating the axioms and unmatched pattern elements into the ontology (Table 1). Concepts and relations missing in the ontology can be optionally renamed by an expert in order to obtain less general names and in this way to better suit the level of abstraction present in the ontology.

[5] http://clarkparsia.com/pellet/
[6] www.hermit-reasoner.com/

Table 1. Integration of pattern elements into an ontology.

Type of unmatched pattern element	Action before inserting the element into the ontology
Concept	Optional renaming by an expert
Relation	Replacement of all matched pattern concepts contained in domain and range axioms by their correspondents, optional renaming by an expert
Axiom	Replacement of all matched concepts and relations by their correspondents

5 Feasibility Study

We conducted an experiment on the ontologies contained in the Watson Ontology repository[7] in order to assess the potential of the proposed method. In the experiment, we used the previously described example consisting of the transitive relations "before" and "after" to examine how well the proposed method can perform for axioms involving transitivity and inverseness of relations.

In the experiment, we used only the relation label itself for the lexical matching. The relations were considered as obligatory pattern elements whereas the axioms about their transitivity and inverseness were considered to be optional.

We used the Watson Search Engine [13] to identify the ontologies containing an *ObjectProperty* definition for at least one of the relations. Thereby, 14 documents were identified and matched against the pattern with results as displayed in Table 2.

Table 2. Experiment results: matching of the after-before-pattern with ontologies indexed by the Watson Ontology Search Engine.

Ontology URL	Result
morpheus.cs.umbc.edu/aks1/ontosem.owl	Inverse only
lists.w3.org/Archives/Public/www-rdf-logic/2003Apr/att-0009/SUMO.daml	"After" is missing
secse.atosorigin.es:10000/ontologies/SUMO.owl	"After" is missing
daml.umbc.edu/ontologies/cobra/0.3/daml-time	Inverse only
ai.sri.com/daml/ontologies/time/Time.daml	Inverse only
cs.umd.edu/~golbeck/daml/slaveOnt.daml	No transitivity and no inverseness
cs.vu.nl/~pmika/owl-s/time-entry-fixed.owl	Complete
isi.edu/~pan/damltime/time-entry.owl	Complete
pervasive.semanticweb.org/ont/2004/06/time	Complete
pervasive.semanticweb.org/ont/dev/time	Complete
isi.edu/~pan/damltime/time.owl	Complete
mogatu.umbc.edu/ont/2004/01/Time.owl	Complete
sweet.jpl.nasa.gov/sweet/time.owl	Complete
daml.umbc.edu/ontologies/cobra/0.4/time-basic	Complete

[7] http://watson.kmi.open.ac.uk/WatsonWUI/

Eight of 14 documents resulted in a complete match of the pattern including all axioms. Three of the ontologies did not include the inverseness axiom, but the transitivity axioms. Two documents did not contain a definition for the relation "after", but a definition for the relation "before" which was defined as transitive. One document did not contain any of the mentioned axioms. We manually examined the refined ontologies and found that the performed completions were semantically justified.

6 Summary and Outlook

In this paper, we presented an algorithm for the identification of ontology pattern instantiations in ontologies along with a method to transfer complex axioms contained in ontology design patterns into a target ontology. The results of our experiment demonstrate the potential of the reuse of formalized knowledge. However, in order to assess the impact of the method more precisely, we plan a large-scale evaluation involving a large set of pattern with different characteristics.

The availability of appropriate and complete ontology patterns is essential for the effectiveness of our approach. Hence, we are currently working on semi-automatic methods to acquire useful patterns as well as the necessary lexical information for each pattern. For the former, we are planning to exploit existing ontologies to identify frequently co-occurring characteristics of ontology elements and in this way to identify particularly useful ontology patterns for ontology refinement. For the latter, we expect existing broad-coverage data sets such as WordNet, BillionTriple-Challenge[8] and DBPedia to be valuable resources. We also intend address the acquisition of composed relation labels such as *followed_by* or *authorOf*, since they are typical in the existing ontologies and difficult to obtain from the usual grossaries. For this purpose, we intend to use the existing methods for the extraction of synonyms and hyponyms based on Harris' Distributional Hypothesis [14] such as [15].

Since the effectiveness of our approach is highly dependent on the quality of the lexical matching, we are currently working on the incorporation of disambiguation techniques as well as matching techniques based on LPs in our lexical matching approach.

Acknowledgments. This work is supported by the EU FP6 NeOn project http://www.neon-project.org.

References

1. Gangemi, A.: Ontology design patterns for semantic web content. In: International Semantic Web Conference, 262–276 (2005)

[8] http://vmlion25.deri.ie/

2. Presutti, V., Gangemi, A.: Content Ontology Design Patterns as Practical Building Blocks for Web Ontologies, In: Proc. of the 27th Int. Conf. on Conceptual Modeling, (2008)
3. Blomqvist. Blomqvist, E.: Semi-automatic Ontology Construction based on Patterns. PhD-Thesis. Linköping University, Department of Computer and Information Science (2009)
4. Presutti, et al.: A Library of Ontology Design Patterns: Reusable Solutions for Collaborative Design of Networked Ontologies. NeOn D2.5.1 (2008)
5. Buitelaar, P., Cimiano, P.: Ontology Learning and Population: Bridging the Gap between Text and Knowledge, volume 167 of Frontiers in Artificial Intelligence and Applications. IOS Press (2008)
6. R. Navigli and P. Velardi.: Ontology enrichment through automatic semantic annotation of on-line glossaries. In Proc. of the 15th Int. Conf. on Knowledge Engineering and Knowledge Management (EKAW), LNCS, vol. 4248, pp. 126–140. Springer, Heidelberg (2006)
7. Völker, J., Hitzler, P., Cimiano, P.: Acquisition of OWL DL axioms from lexical resources. In: Proc. of the 4th European Semantic Web Conference (ESWC'07) (2007)
8. Haase, P., Völker, J.: Ontology learning and reasoning - dealing with uncertainty and inconsistency. In da Costa, P.C.G., Laskey, K.B., Laskey, K.J., Pool, M., eds.: Proc. of the Workshop on Uncertainty Reasoning for the Semantic Web (URSW). 45–55 (2005)
9. Völker, J., Rudolph, S.: Lexico-logical acquisition of OWL DL axioms – An integrated approach to ontology refinement. In: Proc. of the 6th Int. Conf. on Formal Concept Analysis (ICFCA'08) (2008)
10. Völker, J., Rudolph, S.: Fostering web intelligence by semi-automatic owl ontology refinement. In Proceedings of the 7th International Conference on Web Intelligence (WI) (2008)
11. Baader, F., Ganter, B., Sertkaya, B., Sattler, U.: Completing description logic knowledge bases using formal concept analysis. In Veloso, M.M., ed.: IJCAI. 230–235 (2007)
12. Shvaiko, P., Euzenat, J.: A survey of schema-based matching approaches. Journal on Data Semantics, IV. 146–171 (2005)
13. d'Aquin, M., Baldassarre, C., Gridinoc, L., Sabou, M., Angeletou, S., Motta., E.: Watson: Supporting next generation semantic web applications. In Proc. of the WWW/Internet conference (2007)
14. Harris, Z.S.: Word. Distributional Structure 10. 146–162 (1954)
15. Suchanek, F. M., Sozio, M., Weikum, G.: SOFIE: a self-organizing framework for information extraction. In Proc. of the 18th Int. Conf. on World Wide Web (WWW '09) (2009)

Using Lexico-Syntactic Ontology Design Patterns for ontology creation and population

Diana Maynard and Adam Funk and Wim Peters

Department of Computer Science
University of Sheffield
Regent Court, 211 Portobello
S1 4DP, Sheffield, UK

Abstract. In this paper we discuss the use of information extraction techniques involving lexico-syntactic patterns to generate ontological information from unstructured text and either create a new ontology from scratch or augment an existing ontology with new entities. We refine the patterns using a term extraction tool and some semantic restrictions derived from WordNet and VerbNet, in order to prevent the overgeneration that occurs with the use of the Ontology Design Patterns for this purpose. We present two applications developed in GATE and available as plugins for the NeOn Toolkit: one for general use on all kinds of text, and one for specific use in the fisheries domain.

Key words: natural language processing, relation extraction, ontology generation, information extraction, Ontology Design Patterns

1 Introduction

Ontology population is a crucial part of knowledge base construction and maintenance that enables us to relate text to ontologies, providing on the one hand a customised ontology related to the data and domain with which we are concerned, and on the other hand a richer ontology which can be used for a variety of semantic web-related tasks such as knowledge management, information retrieval, question answering, semantic desktop applications, and so on.

Automatic ontology population is generally performed by means of some kind of ontology-based information extraction (OBIE) [1, 2]. This consists of identifying the key terms in the text (such as named entities and technical terms) and then relating them to concepts in the ontology. Typically, the core information extraction is carried out by linguistic pre-processing (tokenisation, POS tagging, etc.), followed by a named entity recognition component, such as a gazetteer and rule-based grammar or machine learning techniques.

In this paper we discuss the use of information extraction techniques involving lexico-syntactic patterns to generate ontological information from unstructured text and either create a new ontology from scratch or augment an existing ontology with new entities. This application represents a typical situation where NLP (natural language processing) techniques can assist in the development of

Semantic Web technology. While the use of such patterns is not new in itself (for example the Hearst patterns [3]), most previous work in this area has focused on using and extending or refining the set of Hearst patterns with additional information, or has focused on patterns within a very specific domain. In this work, we investigate the addition of lexico-syntactic patterns corresponding to ontology design patterns (ODPs) [4]which, in contrast to the Hearst patterns, are very general and thus cover a wide range of sentences but are also very ambiguous. While Hearst patterns produce high precision but low recall, the ontology design patterns produce high recall but low precision. We also include some additional lexico-syntactic patterns and investigate the addition of semantic restrictions to reduce the overgeneration problem. A detailed description of all the patterns and of the semantic restrictions is given in Section 2.

The system is implemented in GATE, an architecture for natural language processing which contains a number of pre-existing language processing components and applications, and enables the user to develop their own applications and integrate new plugins [5]. Two applications are available as plugins for the NeOn toolkit[1]: one for general use on all kinds of text, and one for specific use in the fisheries domain. These are described in more detail in Section 3.

2 Lexico-syntactic patterns

Traditional rule-based NE recognition applications usually rely on a fairly small set of patterns which aim to identify the relevant entities in text. These make extensive use of gazetteer lists which provide all or part of the entity in question, in combination with linguistic patterns (see for example [6] for a discussion of the importance of gazetteers in pattern-based NE recognition). A typical rule to identify a person's name consists of matching the first name of the person with an entry in the gazetteer (e.g. "John" is listed as a possible first name), followed by an unknown proper noun (e.g. "Smith", which is recognised as a proper name by the POS tagger). Most patterns include some combination of proper noun or word with an initial capital letter (for English) and either some gazetteer entry or linguistic feature.

However, identifying ontological concepts and/or relations requires a slightly different strategy. For open relation extraction [7], we have no such lists to use as a starting point. Even where we do have a seed ontology or known lists of terms and can make use of this information, it is often still insufficient because the concept may not be in the ontology yet, may be in the ontology but ambiguous, or may exist there in a different form (e.g. as a synonym or as a linguistic variant).

An alternative approach to traditional recognition techniques is to make use of linguistic patterns and contextual clues. Lexico-syntactic pattern-based ontology population has proven to be reasonably successful for a variety of tasks [8]. The idea of acquiring semantic information from texts dates back to the

[1] `http://www.neon-toolkit.org/`

early 1960s with Harris' *distributional hypothesis* [9] and Hirschman and Sager's work in the 1970s [10], which focused on determining sets of sublanguage-specific word classes using syntactic patterns from domain-specific corpora. A detailed description and comparison of lexical and syntactic pattern matching can be found in [11], In particular, research in this area has been used in specific domains such as medicine, where a relatively small number of syntactic structures is often found, for example in patient records. Here the structures are also quite simple, with short and relatively unambiguous sentences typically found: this makes syntactic pattern matching much easier.

We have identified three sets of patterns which can help us identify concepts, instances and properties to extend the ontology: the well-known Hearst patterns (Section 2.1), the Lexico-Syntactic Patterns developed in NeOn corresponding to Ontology Design Patterns (Section 2.2), and some new contextual patterns defined by us which take into account contextual information (Section 2.3). As a first step, we identified patterns which generated basic ontology elements such as instances, classes, subclasses and properties.

2.1 Hearst patterns

The Hearst patterns are a set of lexico-syntactic patterns that indicate hyponymic relations [3], and have been widely used by other researchers, e.g. [12]. Typically they achieve a very high level of precision, but quite low recall: in other words, they are very accurate but only cover a small subset of the possible patterns for finding hyponyms and hypernyms. The patterns can be described by the following rules, where NP stands for a Noun Phrase and the regular expression symbols have their usual meanings[2]:

- `such NP as (NP,)* (or|and) NP`
  Example: ... works by such *authors* as *Herrick*, *Goldsmith*, and *Shakespeare*.
- `NP (,NP)* (,)? (or|and) (other|another) NP`
  Example: *Bruises*, *wounds*, or other *injuries*...
- `NP (,)? (including|especially) (NP,)* (or|and) NP`
  Example: All *common-law countries*, including *Canada* and *England*...

Hearst actually defined five different patterns, but we have condensed some of them into a single rule. Also, where Hearst defines the relations as hyponym-hypernym, we need to be more specific when translating this to an ontology, as they could represent either instance-class or subclass-superclass relations. To make this distinction, we tested various methods. In principle, POS-tagging should be sufficient, since proper nouns generally indicate instances, but our tagger mistags capitalised common nouns (at the beginning of sentences) as proper nouns frequently enough that we cannot rely on it for this purpose. We also looked at the presence or absence of a determiner preceding the noun (since proper nouns in English rarely have determiners) and whether the noun is singular or plural, but this still left the problem of the sentence-initial nouns.

[2] `()` for grouping; `|` for disjunction; `*`, `+`, and `?` for iteration.

Finally, we decided to pre-process the text with the named entity recognition application ANNIE, and only consider certain types of named entities (*Person*, *Location*, *Organization*, and potentially some unknown entity types) as candidates for instances; all other NPs are considered to be classes. This gave much better results, occasionally missing an instance but rarely overgenerating.

2.2 ODP Lexico-Syntactic Patterns

The second type of patterns investigated was the set of Lexico-Syntactic Patterns (LSPs) corresponding to Ontology Design Patterns. We implemented a number of these patterns in our application. Some patterns could not be implemented because the GATE ontology API and the NEBONE plugin (which enables the ontology editing) do not contain the functionality for all restrictions.

In the following rules, `<sub>` and `<super>` are like variable names for the subclasses and superclasses to be generated; `CN` means *class of*, *group of*, etc.; `CATV` is a classification verb[3]; `PUNCT` is punctuation; `NPlist` is a conjoined list of NPs ("X, Y and Z").

1. Subclass rules
   – `NP<sub> be NP<super>`
   – `NPlist<sub> be CN NP<super>`
   – `NPlist<sub> (group (in|into|as) | (fall into) | (belong to)) [CN] NP<super>`
   – `NP<super> CATV CV? CN? PUNCT? NPlist<sub>`

   Example: *Frogs* and *toads* are kinds of *amphibian.*
   Thyroid medicines belong to the general group of *hormone medicines.*
2. Equivalence rules
   – `NP<class> be (the same as|equivalent to|equal to|like) NP<class>`
   – `NP<class> (call | denominate | (designate by|as) | name) NP<class>` (where the verb is passive)

   Example: *Poison dart frogs* are also called *poison arrow frogs.*
3. Properties
   – `NP<class> have NP<property>`
   – `NP<instance> have NP <property>`

   Example: *Birds* have *feathers.*
   Sharks have 32 teeth.

It is important to note that these particular LSPs were designed to be used as support for ontology modelling, rather than directly for automatic discovery of ontological relations. The effect of this is that while these patterns are quite productive (for example `X is a Y`), most of them are potentially ambiguous and susceptible to overgeneration when applied to the automatic discovery process. In particular, general patterns such as "NP have NP" and "NP be NP"

[3] E.g., *classify in/into*, *comprise*, *contain*, *compose (of)*, *group in/into*, *divide in/into*, *fall in/into*, *belong (to).*

are very problematic. For example, the former pattern also matches sentences like “Writers have penguins based at the North Pole” and extracts the relation “writers have penguins” which is clearly wrong. Similarly, the pattern “NP be NP” would match sentences like “Sheep are a separate species” and extract the concept “sheep” as a subclass of “separate species” which makes no sense. There is also much ambiguity with this pattern: for example, in the sentence “Heliculture is the farming of snails”, “heliculture” should be recognised as a synonym of “farming of snails” and not as a subclass. Clearly, such patterns are far too general to be used off the shelf. In Section 2.4 we discuss some restrictions we have implemented which aim to counteract these and other problems.

2.3 Contextual patterns

We also defined a set of rules designed to make use of contextual information in the text about known entities already existing in the ontology (unlike the lexico-syntactic patterns which assume no previous ontological information is present). These rules are used in conjunction with the OntoRootGazetteer plugin in GATE, which enables any morphological variant of any class, instance or label in the ontology to be matched with (any morphological variant of) any word or words in the text. Which elements from the ontology are to be considered (e.g., whether to include properties, and if so which ones) is determined in advance by the user when setting up the application. Note that because the generation process is incremental, involving a pipeline of sequential processsing resources, and because we use NEBOnE which generates the ontology on-the-fly, we do not necessarily need a seed ontology from which to start, because we can make use of the ontology entities already generated by the previous two sets of patterns. More information about the application and about NEBOnE is given in Sections 3 and 3.3 respectively.

Below we describe the contextual patterns we have identified:

1. **Add a new subclass**: `(Adj|N) NP<class>` → `NP<subclass>`.
   This matches a class name aready in the ontology preceded by an adjective or noun, such as adjective preceding a known type of fish, which we assume is a more specific type. For example, when we encounter the phrase . . . Japanese flounder. . . in a text and *flounder* is already in the ontology, we add *Japanese flounder* as a subclass of *flounder.*
2. **Add a new class** (a more generic version of the Hearst patterns). Here we postulate that an unknown entity amidst a list of known entities is likely to be also an entity of the same type. For example, if we have a list of classes of fish, and there is an unknown noun phrase in amongst the list, we can presume that this is also a class of fish. To decide where to add this new class in the ontology, we can look for the Most Specific Common Abstraction (MSCA) of all the other items in the list (i.e. the lowest common superclass of all the classes in the list) and add the new entity as a subclass of this class.
   Example: *Hornsharks*, *leopard sharks* and *catsharks* can survive in aquarium conditions for up to a year or more.

where *hornshark* and *leopard shark* are classes in the ontology and *catshark* is unknown, so we can recognise *catshark* as a subclass with the same parent as that of *hornshark* and *leopard shark*, in this case *shark*.
3. **Add an alternative name as a synonym**: a name followed by an alternative name in brackets is a very common pattern in some kinds of text. For example in texts about flora and fauna we often get the common name followed by the Latin name in brackets, as in the following sentence:
Example: *Mummichogs* (*Fundulus heteroclitus*) were the most common single prey item.
If we know that one of the two NPs is a class or instance in the ontology, we can predict fairly accurately that the other NP is a synonym.

2.4 Adding semantic restrictions

Due to the overgeneralisation of some of the patterns described above, in particular the ODP LSPs, we have incorporated some restrictions on them. First, we restrict possible subclasses and classes to terms rather than to all NPs. For this, we use TermRaider, a term selection algorithm (currently unpublished) we have developed based on linguistic filtering and tf-idf scoring. This increases the precision dramatically, but lowers the recall a little; however, adjusting TermRaider's parameters to be a little more flexible with patterns should improve the recall.

The second restriction we imposed was to include lexical resources containing semantic classes from WordNet [13] and VerbNet [14], which enable the incorporation of deeper semantic information. This allows us (i) to look for verbal patterns connecting terms in a sentence, using the ANNIC plugin in GATE [15], and (ii) to restrict the kinds of relation extracted. For example, we can restrict the kinds of entities that have body parts associated with them to animals and humans. We aim not only to reduce the number of errors, but also to eliminate the kind of general relations which while not incorrect, are not very useful. For example, knowing that a turtle is a local creature is not of much interest unless more contextual information is provided (i.e. in which region it is local).

Restrictions on Subclass Patterns One example of a restriction we placed was on the subclass rule `(Adj|N) NP<class>` → `NP<subclass>` from the set of contextual patterns, which we modified so that either the superclass must already exist in the ontology as a recognised class, or such that certain semantic restrictions apply. One such restriction states that both the proposed subclass and superclass must have the semantic category "animal". For example, this enables us to recognise "carrot weevil" as a subclass of "weevil". This rule in particular has very high accuracy (98%) and only seems to cause errors as a result of incorrect semantic categories from WordNet.

Restrictions on Properties One of the most error-prone rules was the Property rule `X has Y` from the Lexico-Syntactic Patterns set, which was clearly far too general. We restricted this to again use semantic categories of WordNet. For patterns involving animals we can state that X must be an animal and Y must be a body part. This gave much better results (approximately 75% accuracy, although low recall). Another restriction is the type of thing that can be considered a property. We experimented with restricting the range of the property to the following semantic categories from WordNet: plant, shape, food, substance, object, body, animal, possession, phenomenon, artifact, and found much improved results.

3 SPRAT and SARDINE

We have developed two applications in GATE which make use of the lexico-syntactic pattern matching techniques to create and/or populate ontologies. Both applications are available as part of the GATE webservices (SAFE) plugin for the NeOn toolkit.

First, we have developed a generic application, SPRAT (Semantic Pattern Recognition and Annotation Tool) which can be used on any kind of text. This recognises new concepts, instances and properties, as described above, and adds these to a new or existing ontology. We have tested the application on wikipedia texts about animals (See Section 4) with good results so far, and plan to test on other domains and text types.

Second, we have developed a specific application, SARDINE (Species Annotation and Recognition and Indexing of Named Entities) which is aimed at the fisheries domain. The idea behind SARDINE is to identify mentions of fish species from text. The main difference between SARDINE and SPRAT is that, in addition to being developed for a specific domain, SARDINE also relies on a pre-existing domain-specific ontology which acts as a seed. We use the species ontology developed by the FAO[4] for this purpose. The application recognises:

- existing fish names listed in the seed ontology
- potential new fish names not listed in the seed ontology
- potential relations between fish names

For the new fish, it attempts to classify them in the ontology, based on linguistic information such as synonyms and hyponyms of existing fish. The application can either generate the new items directly into the seed ontology, or create a new ontology in the same way as SPRAT does. The latter is generally preferable because the original seed ontology is quite large and cumbersome, so it is easier to create a new smaller ontology which can then be easily verified by a human expert and then merged with the original seed ontology.

3.1 Processing Resources

Both applications are composed of a number of GATE components: some linguistic pre-processing followed by a set of gazetteer lists and the JAPE grammars described above. The components are as follows:

[4] Food and Agriculture Organization of the United Nations – `http://www.fao.org/`

– Tokeniser: divides the text into tokens
– Sentence Splitter: divides the text into sentences
– POS-Tagger: adds part-of-speech information to tokens
– Morphological Analyser: adds morphological information (root, lemma etc.) to tokens
– NP chunker: divides the text into noun phrase chunks
– Gazetteers: looks up various items in lists
– OntoRootGazetteer (optional): looks up items from the ontology and matches them with the text, based on root forms
– JAPE transducers: annotates text and adds new items to the ontology

The application can either create an ontology from scratch, or modify an existing ontology. SARDINE also requires the presence of a seed ontology, which could be the ontology to be modified, or a different one. The ontology used is the same one for the whole corpus: this means that if a number of documents are to be processed, the same ontology will be modified. If this is not the desired behaviour, then there are two options:

1. A separate corpus is created for each document or group of documents corresponding to a single output ontology. The application must be run separately for each corpus.
2. A processing resource can be added to the application that clears the ontology before re-running on the next document. This requires that the ontology is saved at the end of the application, after processing each document.

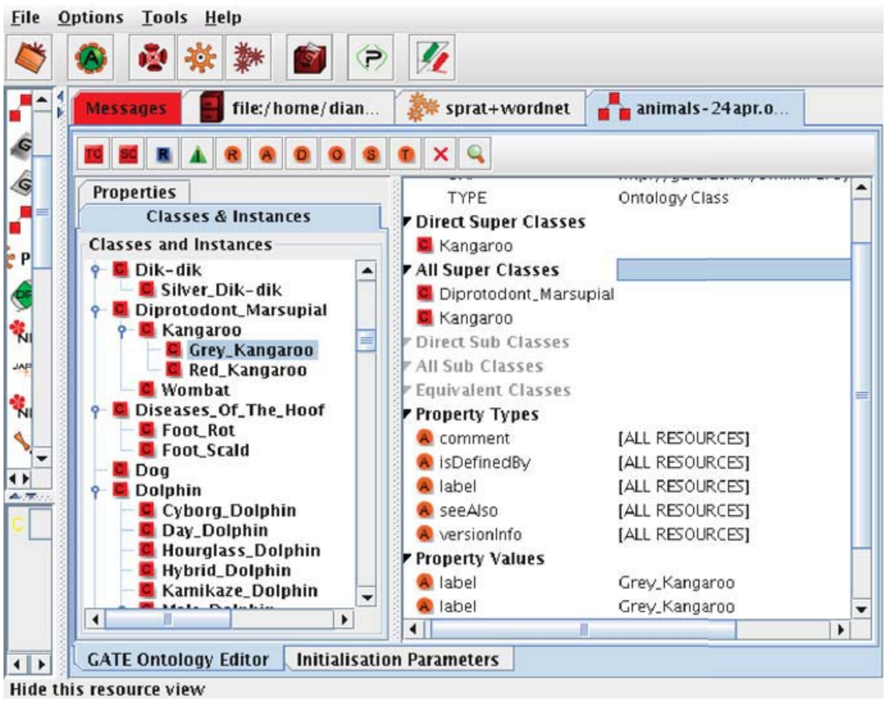
3.2 Implementation of patterns

The patterns themselves are implemented as JAPE rules [16]. On the left hand side (LHS) of the rule is the pattern to be annotated. This consists of a number of pre-existing annotations which have been created as a result of pre-processing components (such as POS tagging, gazetteer lookup and so on) and (potentially) earlier JAPE rules. The example below shows a pattern for matching a subclass relation, such as “Frogs are a kind of amphibian” where “frog” is annotated as a subclass of “amphibian”.

```
Rule:Subclass1
(
 ({NP}):sub
 ({Lookup.minorType == be}
 {Token.category == DT}{Lookup.majorType == kind})
 ({NP}):super
) --> ...
```

This pattern matches a noun phrase (identified by our NP Chunker), followed by some morphological variant of the verb “to be” (identified via the gazetteer lookup), a determiner (identified via the POS tagger), some word(s) indicating a “kind of” relation (identified via the gazetteer lookup) followed by another noun phrase (identified by the NP Chunker). The two noun phrases (corresponding ultimately to the subclass and superclass) are given labels ("sub" and "super") which will used in the second part of the rule.

The right hand side (RHS) of the rule invokes NEBOnE and creates the new items in the ontology, as well as adding annotations to the document itself. NEBOnE is responsible also for ensuring that the resulting changes to the ontology are wellformed: this is described in more detail in Section 3.3. The RHS of the rule first gets the relevant information from the annotations (using the labels assigned on the LHS of the rule), then adds a new class below the root class for the superconcept (labelled "amphibian" in our example), a new subclass of this (labelled "frog" in our example), and finally adds annotations to the entities in the document. Figure 1 shows a screenshot from GATE of an ontology created.



Dik-dik
    Silver_Dik-dik
Diprotodont_Marsupial
    Kangaroo
        Grey_Kangaroo
        Red_Kangaroo
    Wombat
Diseases_Of_The_Hoof
    Foot_Rot
    Foot_Scald
Dog
Dolphin
    Cyborg_Dolphin
    Day_Dolphin
    Hourglass_Dolphin
    Hybrid_Dolphin
    Kamikaze_Dolphin

TYPE	Ontology Class
Direct Super Classes	
Kangaroo	
All Super Classes	
Diprotodont_Marsupial	
Kangaroo	
Direct Sub Classes	
All Sub Classes	
Equivalent Classes	
Property Types	
comment	[ALL RESOURCES]
isDefinedBy	[ALL RESOURCES]
label	[ALL RESOURCES]
seeAlso	[ALL RESOURCES]
versionInfo	[ALL RESOURCES]
Property Values	
label	Grey_Kangaroo
label	Grey_Kangaroo

Fig. 1. Generated ontology in GATE

3.3 NEBOnE

Both applications use the specially developed NEBOnE plugin for GATE in order to generate the changes to the ontology. NEBOnE (Named Entity Based ONtology Editing) is an implementation for processing natural language text and manipulating an ontology, and is derived from the CLOnE plugin [17] for GATE. The major difference between CLONE and NEBOnE is that while CLONE relies on a restricted input text (generated by the user in a controlled language), NEBOnE can be used with unrestricted free text, so it is a lot more flexible. When the NEBOnE plugin is installed, actions concerning the ontology are implemented on the RHS (right-hand side) of JAPE rules, such as adding or deleting new classes, instances, subclasses, properties and so on.

Once the text has been pre-processed, a JAPE transducer processes each sentence in the input text and manipulates the ontology appropriately. This Processing Resource refers to the contents of the ontology in order to analyse the input sentences and check for errors; some syntactically identical sentences may have different results if they refer to existing classes, existing instances, or non-existent names, for example.

4 Evaluation

We evaluated the accuracy of the lexical patterns using a corpus of 25 randomly selected wikipedia articles about animals, such as the entries for *rabbit*, *sheep* etc. We ran SPRAT and examined the results in some detail[5]. In total, SPRAT generated 1058 classes, of which 83.6% were correct; 659 subclasses, of which 76.6% were correct, 23 instances, of which 52.2% were correct, and 55 properties, of which 74.5% were correct. Note that, unlike in traditional named entity recognition evaluation, we use a strict method of scoring where a partially correct response, i.e. one where the span of the extracted entity is too short or too long, is considered as incorrect. This is because for ontology population, having an incorrect span is generally a more serious error than in named entity recognition.

We should point out that in these type of texts (articles about animals) the number of instances is quite small. The wrongly extracted instances were largely the result of erroneous named entity recognition. For example, *Barbados Blackbelly* was wrongly recognised by the system as a named entity, and was therefore extracted as an instance rather than as a subclass of *Sheep Breed*.

While we find the initial results from SPRAT very encouraging, we can see that the patterns implemented are far from foolproof, since unlike with a controlled language such as CLOnE, we cannot rely on a one-to-one correspondence between a simple syntactic structure and its semantics. First we have the problem of overgeneration. Already, we have discarded some potential patterns (such as some of the ODP LSPs) that we consider to generate too many errors. Further refinement is still necessary here, either to remove other patterns or to reimplement them in a different way.

One of the main causes of overgeneration is caused by the span of the noun phrase describing the concept to be added to the ontology. We have experimented with different possibilities. A larger span provides finer distinctions and thus

[5] We have not currently evaluated SARDINE formally, but informal tests show similar results

better classes, but overgenerates considerably, while a smaller span produces more general classes but better accuracy (does not overgenerate so much). By restricting the noun phrases to terms recognised by TermRaider, we solve this problem somewhat, but this means that the results are only as good as the terms recognised. It is also apparent that sometimes the restriction to terms risks losing some important information. For example, in the sentence:

> Mygalomorph and Mesothelae spiders have two pairs of book lungs filled with haemolymph

we can correctly recognise the relation "*Mesothelae spiders* have *book lungs*", but a better relation might be "*Mesothelae spiders* have *two pairs of book lungs*". We might also want to capture the fact that the book lungs are filled with haemolymph.

Second, as we discussed earlier, and as mentioned in [4],lexico-syntactic patterns tend to be quite ambiguous as to which relations they indicate. For example, `NP have NP` could indicate an object property or a datatype property relationship. Also, English word order can lead to inverse relations. For example, in the sentence "A traditional Cornish pilchard dish is the stargazy pie", *stargazy pie* is a kind of Cornish pilchard dish, but the sentence can equally be written "The stargazy pie is a traditional Cornish pilchard dish". Here, the use of the definite and indefinite determiner helps to identify the correct relationship, but this is not always the case. Often, further context is also crucial. For example, in the sentence "Both African males and females have external tusks", it is not very useful to extract the concept *females* with the property *have external tusks* unless you know that *females* actually refers to female African elephants. To extract this information would require also coreference matching, which is planned for the future.

Finally, complex and negative sentences can cause errors. From the phrase "DAT is a legitimate therapy", we could easily deduce that *DAT* could be classified as an instance of *therapy*. However, further inspection of the wider context reveals that the opposite is true, as the sentence actually reads "...there is no compelling scientific evidence that DAT is a legitimate therapy." This is a common problem with shallow NLP systems.

Integration of a full parser has also been investigated, but discarded on the grounds of speed (full parsing is extremely computationally expensive in this situation). In particular, we found that the sentences in Wikipedia articles, which we have used for training and testing, are quite hard to parse well, because they frequently exhibit a long and complex sentence structure which is highly ambiguous to a parser. This causes not only speed but also accuracy problems.

5 Related work

As already mentioned, the use of lexico-syntactic patterns in itself is far from new and has already proved to be successful for a variety of tasks [8]. Various attempts have been made to extend the Hearst patterns in a semi-automatic way, for example using the web as evidence [12]. Other methods focus mainly on a specific kind of pattern, such as *part-of* relations [18], or use clustering approaches [19]. The disadvantage of the latter is that they require large corpora to work well and generally fail to produce good clusters from fewer than 100 million words.

The closest approach to ours is probably Text2Onto [20], which performs relation extraction on the basis of patterns. It combines machine learning approaches with basic linguistic processing such as tokenisation, lemmatisation and shallow parsing. Our approach differs in that it has a greater number of lexico-syntactic patterns, including the ODP ones, and it currently uses only a rule-based approach rather than machine learning, with no statistical clustering or parsing. This leads to much increased precision over Text2Onto, though fewer relations are produced. It also enables a more flexible approach and fine-tuning of the system.

We also took inspiration from some currently unpublished research carried out at DFKI in the Musing project[6], which aims to derive T-Box Relations from unstructured texts in German. In this work, attention is focused primarily on deriving relations between parts of German compound nouns, but we can make use of similar restrictions.

Within the range of activities required for ontology learning, our approach covers a number of intermediate stages in the process of ontology acquisition, namely term recognition and relation extraction. In the initial acquisition stage, it will recognise terms from the corpus only if they participate in any of the patterns. This guarantees termhood only up to a certain extent. For relation extraction, we do not make use of a parser. There are many applications that make use of syntactic dependencies, e.g. [21, 22]. Our approach differs from this in that our patterns are defined at low levels of syntactic constituency, such as noun phrases, and by means of finite state transducers. Identifying and engineering on the basis of the linguistic building blocks that are relevant for each ontology editing task eliminates the need for a parser. This bottom-up approach is much faster and less error-prone than a parser, and is more in line with the ontology bootstrapping approach advocated in [23].

6 Conclusions and Further Work

In summary, the idea behind this work is to investigate the extent to which such patterns can be used either on their own or in conjunction with the user to generate or populate a more detailed ontology from text. Both SPRAT and SARDINE applications assist the user in the generation and/or population of ontologies from text. They are available to download for use as GATE Web-service plugins for the NeOn toolkit [7]. The lexico-syntactic patterns we have implemented provide a good basis, but there is some work still to go in improv-

[6] `http://www.musing.eu/`

[7] `http://www.neon-toolkit.org`

ing the rules, and we have put forward a number of suggestions for ways in which this might be done.

One further possibility for improvement is to incorporate combinations of Hearst patterns and statistically derived collocational information, because its combination with lexico-syntactic patterns has proven to improve precision and recall [24]. Integration of a full parser has also been investigated, but discarded on the grounds of speed (full parsing is extremely computationally expensive in this situation). In particular, we found that the sentences in Wikipedia articles, which we have used for training and testing, are quite hard to parse well, because they frequently exhibit a long and complex sentence structure which is highly ambiguous to a parser. This causes not only speed but also accuracy problems.

Acknowledgements. This research was partially supported by the EU Sixth Framework Program project NeOn (IST-2005-027595).

References

1. Maynard, D., Cunningham, H., Kourakis, A., Kokossis, A.: Ontology-Based Information Extraction in hTechSight. In: First European Semantic Web Symposium (ESWS 2004), Heraklion, Crete (2004)
2. Saggion, H., Funk, A., Maynard, D., Bontcheva, K.: Ontology-based information extraction for business applications. In: Proceedings of the 6th International Semantic Web Conference (ISWC 2007), Busan, Korea (November 2007)
3. Hearst, M.A.: Automatic acquisition of hyponyms from large text corpora. In: Conference on Computational Linguistics (COLING'92), Nantes, France, Association for Computational Linguistics (1992)
4. de Cea, G.A., Gómez-Pérez, A., Ponsoda, E.M., Suárez-Figueroa, M.C.: Natural language-based approach for helping in the reuse of ontology design patterns. In: Proceedings of the 16th International Conference on Knowledge Engineering and Knowledge Management Knowledge Patterns (EKAW 2008), Acitrezza, Italy (September 2008)
5. Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V.: GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications. In: Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02). (2002)
6. Mikheev, A., Moens, M., Grover, C.: Named Entity recognition without gazetteers. In: Proceedings of the Ninth Conference of the European Chapter of the Association for Computational Linguistics (EACL'99). (1999) 1–8
7. Banko, M., Etzioni, O.: The tradeoffs between open and traditional relation extraction. In: Proceedings of ACL-08. (2008)
8. Etzioni, O., Cafarella, M., Downey, D., Kok, S., Popescu, A., Shaked, T., Soderland, S., Weld, D.S., Yates, A.: Web-scale Information Extraction in KnowItAll. In: Proceedings of WWW-2004. (2004) `http://www.cs.washington.edu/research/knowitall/papers/www-paper.pdf`.
9. Harris, Z.: Mathematical Structures of Language. Wiley (Interscience), New York (1968)
10. Hirschman, L., Grishman, R., Sager, N.: Grammatically based automatic word class formation. Information Processing and Retrieval **11** (1975) 39–57

11. Maynard, D.G.: Term Recognition Using Combined Knowledge Sources. PhD thesis, Manchester Metropolitan University, UK (2000)
12. Pantel, P., Pennacchioni, M.: Espresso: Leveraging generic patterns for automatically harvesting semantic relations. In: Proceedings of Conference on Computational Linguistics / Association for Computational Linguistics (COLING/ACL-06), Sydney, Australia (2006) 113–120
13. Fellbaum, C., ed.: WordNet - An Electronic Lexical Database. MIT Press (1998)
14. Schuler, K.K.: VerbNet: A broad-coverage, comprehensive verb lexicon. PhD thesis, University of Pennsylvania (2005)
15. Aswani, N., Tablan, V., Bontcheva, K., Cunningham, H.: Indexing and Querying Linguistic Metadata and Document Content. In: Proceedings of Fifth International Conference on Recent Advances in Natural Language Processing (RANLP2005), Borovets, Bulgaria (2005)
16. Cunningham, H., Maynard, D., Tablan, V.: JAPE: a Java Annotation Patterns Engine (Second Edition). Research Memorandum CS–00–10, Department of Computer Science, University of Sheffield (November 2000)
17. Funk, A., Tablan, V., Bontcheva, K., Cunningham, H., Davis, B., Handschuh, S.: CLOnE: Controlled Language for Ontology Editing. In: Proceedings of the 6th International Semantic Web Conference (ISWC 2007), Busan, Korea (November 2007)
18. Berland, M., Charniak, E.: Finding parts in very large corpora. In: Proceedings of ACL-99, College Park, MD (1999) 57–64
19. Pantel, P., Ravichandran, D.: Automatically labeling semantic classes. In: Proceedings of HLT/NAACL-04), Boston, MA (2004) 321–328
20. Cimiano, P., Voelker, J.: Text2Onto - A Framework for Ontology Learning and Data-driven Change Discovery. In: Proceedings of the 10th International Conference on Applications of Natural Language to Information Systems (NLDB), Alicante, Spain (2005)
21. Cimiano, P., Hartung, M., Ratsch, E.: Learning the appropriate generalization level for relations extracted from the Genia corpus. In: Proc. of the 5th Language Resources and Evaluation Conference (LREC). (2006)
22. Gamallo, P., Gonzalez, M., Agustini, A., Lopes, G., de Lima, V.: Mapping syntactic dependencies onto semantic relations. In: Proc. of the ECAI Workshop on Machine Learning and Natural Language Processing for Ontology Engineering. (2006)
23. Maedche, A.: Ontology Learning for the Semantic Web. Kluwer Academic Publishers, Amsterdam (2002)
24. Cederberg, S., Widdows, D.: Using LSA and noun coordination information to improve the precision and recall of automatic hyponymy extraction. In: Proceedings of the 7th conference on Natural language learning at HLT-NAACL, Morristown, NJ (2003) 111–118

Representing the Component Library into Ontology Design Patterns

Aldo Gangemi[1] and Vinay K. Chaudhri[2]

[1] STLab, ISTC-CNR, Roma, Italy
`aldo.gangemi@cnr.it`
[2] SRI International, Menlo Park, US
`vinay.chaudhri@sri.com`

Abstract. For Ontology Design Patterns (OP) to be widely adopted by conceptual modelers, we need a critical amount of them, and that amount should be larger for patterns that describe good practices for modeling content (ie, Content Patterns or CP), e.g. about time, space, events, biological entitites, medical cases, legal norms, etc. It is possible and desirable to reuse existing repositories that contain modeling solutions, and to represent them as OPs. This paper analyzes some components from the Component Library (CLIB), proposing some solutions to represent them into OWL2 CPs. Some constructs from CLIB components need the expressivity of rule languages in order to fully represent them, but these extra-features can be separated from the basic ontological content. Additionally, CLIB components are shown to be enrichable with the pattern annotation schema defined for OPs, which also allows a quick upload and publication on `ontologydesignpatterns.org`.

1 Introduction

Ontology Design Patterns (OP) [20] are reusable solutions for modeling ontologies, based on good practices. In order to be widely adopted by ontology modelers, we need a large amount of them, especially for patterns that describe good practices for modeling content (Content Patterns, CP), either general (time, space, events, etc.) or specific (biological entities, medical cases, legal norms, etc.).

The `ontologydesignpatterns.org` initiative [19] aims to collect the OPs collaboratively, and several of them are being uploaded on its semantic wiki-based site. On the other hand, it is also possible to reuse existing repository of resources, which contain modeling solutions that can be represented as CPs. One of them is the Component Library (CLIB) [8, 10],[3] which contains hundreds of solutions, and explicitly builds on the idea of a *system of concepts* [8], as well as to that of *knowledge patterns* [10]. The original idea of the authors of CLIB [8] is indeed very close to that of CPs:

[3] `http://www.cs.utexas.edu/ mfkb/RKF/tree/`

... our goal is to identify repeated patterns of axioms in a large theory, and then abstract and reify those patterns as components in their own right (analogous to the notion of "design patterns" in object-oriented programming ...

... in contrast to [a] DL algorithm which exhaustively constructs concept representations without regard to task, our algorithm is goal-driven, constructing only those parts of the concept representation required to answer questions. Our trade-off is to sacrifice completeness for a language sufficiently expressive for our purposes. An interesting consequence of our approach is that the concept description which is built is question-specific, containing just that information required to answer the question(s) which were posed ...

In practice, the main requirements of CPs: small, task-based models that fit one or more competency questions by following good practices [16, 20], are shared by CLIB components. The main differences with current OWL CPs include the encoding of CLIB is done in the KM language. The statements in KM have straightforward and well-defined semantics in First-order logic [9]. KM components also represent dynamic aspects of actions the representation for which has not been studied in the context of OWL.

This paper analyzes some components from CLIB, showing how to represent them into OWL CPs. OWL2 [21] is employed in order to take advantage of the maximal expressivity currently available for the Semantic Web. Some constructs from CLIB components need the expressivity of rule languages in order to be fully represented. These extra-features can be separated from the basic ontological content, leaving intact the value of CLIB as a resource for CPs.

Additionally, CLIB components are shown to be enrichable with the pattern annotation schema defined for OPs, which also allows a quick upload and publication on the `ontologydesignpatterns.org` wiki.

The paper is organized as follows: in Section 2 we introduce CLIB and its main features; in Section 3 some uses of the CLIB are described; in Section 4 we represent some CLIB components as CPs on the Semantic Web; in Section 5 we present the publishing plans for CLIB components as CPs.

2 Component Library

The Component Library or CLIB was created with the goal of enabling users with little experience in knowledge engineering to represent knowledge from their domain of expertise by instantiating and composing generic components from a small, hierarchical library. Components are coherent collections of axioms that can be given an intuitive label, usually a common English word. The components should be general enough that their axiomatization is relatively uncontroversial. Composition consists of specifying relationships between instantiated components so that additional implications can be computed. The current CLIB contains a few hundred components, and less than one hundred relations.

Each component of CLIB is formally expressed in KM [9], which in turn is defined in first-order logic. KM includes a situation mechanism for representation and reasoning with actions and the changes they cause.

The main division in CLIB is between entities (things that are) and events (things that happen). Events are states and actions. States represent relatively static situations brought about or changed by actions.

2.1 Actions and States

The actions are grouped into fifteen top-level clusters, each having several more specific subclasses. These clusters are: Add, Remove, Communicate, Create, Break, Repair, Move, Transfer, Make-Contact, Break-Contact, Make-Accessible, Make-Inaccessible, Perceive, Shape, and Orient.

The list was developed by consulting linguistic resources such as WordNet [15], the defining vocabulary of the Longman's Dictionary of Contemporary English and the Roget's theasaurus. We consider here a few examples of the formal axioms that define the actions in the component library. (In the following, the words shown in all caps are the concepts drawn from the CLIB, and the words shown in italics are the relations drawn from the CLIB.)

Conditional rules: If the raw material of a PRODUCE is a SUBSTANCE, then the *product* is composed of that SUBSTANCE. If the *raw materials* are OBJECTs, then the *product* has those OBJECTs as *parts*.

Definitions: An instance of MOVE whose destination is *inside* a CONTAINER is automatically reclassified as an instance of ENTER.

Simulation: If the *destination* of a MOVE is a SPATIAL-ENTITY then the *location* of the OBJECT of the MOVE after the MOVE is that SPATIAL-ENTITY.

States are coherent collections of axioms that represent situations brought about or changed by actions. Many of the CLIB actions are defined in terms of the change in state they cause. This relationship between actions and states is made explicit in the library: there are actions that put objects into states, actions that take objects out of states and actions whose behavior is affected by objects being in states. For example, the BREAK action puts an object into a BE-BROKEN state. The REPAIR action takes an object in a BE-BROKEN state out of that State.

2.2 Entity and Roles

The entity hierarchy in CLIB is less developed than the hierarchy of actions. An important sub-division of entities contains role concepts. A role can be thought of as a temporally unstable entity. It is what an entity is in the context of some event. For example, PERSON is an entity while EMPLOYEE is a role. A PERSON remains a PERSON independent of the events in which she participates. Conversely, someone is an EMPLOYEE only by virtue of participation in an EMPLOYMENT event. A more detailed discussion on the representation of roles and the work related to them is available elsewhere [14].

2.3 Relations

The CLIB contains a small set of relations to connect Entities and Events. The design of the relations between events and entities was inspired by the case roles in linguistics [1], the design of relations between entities was based on the semantics of English noun phrases, and the choice of relationships between events followed from studies in discourse analysis, and process planning.

Examples of event-entity relations are: *agent, object, instrument, etc.* Examples of entity-to-entity relations are *content, has-part, location, material, etc.* Examples of event-to-event relationships are *causes, defeats, enables, prevents, etc.*

While the current CLIB contains domain and range constraints for all these relations, it does not yet contain their complete axiomatization. For example, the CLIB does not yet contain axioms for what it means for an action A to *prevent* another action B.

2.4 Properties

The CLIB has a small number of properties. Properties link entities to values. For example, the *size* of an entity is a property that takes a value. The value can be a cardinal (25 kilograms), a scalar (big relative to housecats) or a categorical (brown). The current CLIB has about 25 general categories. This final list of properties includes such properties as *age, area, capacity, color, length, shape, size, smell* and *wetness.*

3 Uses of the Component Library

The Component Library has been used in several projects, but most notably, in Vulcan's Project Halo (See `http://www.projecthalo.com`) , and DARPA's Project CALO (See `http://caloproject.sri.com`). More detailed description of these uses are available elsewhere [3, 4], but for the present paper, we only focus on its use in a system called AURA that has been developed under Vulcan's Project Halo [4].

The short-term goal of AURA is to enable domain experts to construct declarative knowledge bases (KBs) from a science textbook in the domains of Physics, Chemistry, and Biology in a way that it can answer questions similar to those in a college level exam. The overall concept of operation for AURA is as follows: a knowledge formulation engineer (KFE) with at least a graduate degree in the discipline of interest undergoes 20 hours of training to enter knowledge into AURA; a different person, called a question formulation engineer (QFE), with a high school level education undergoes 4 hours of training and asks questions of the system. Knowledge entry is inherently a skill-intensive task, and therefore, requires more advanced training in the subject as well as in using the system. A QFE is a potential user of the system, and the training requirement was kept lower because we wanted the barrier to using the system to be as low as possible.

For this section, we primarily focus on the knowledge formulation component of AURA because that highlights the use of CLIB more clearly.

The KFEs build their KBs by starting from CLIB. AURA implements a way to convert the axioms in CLIB into a graphical form, to allow a KFE to search for required components, and to graphically assemble them into a domain-specific representation. As a concrete example, we show how a KFE would represent the concept of Virus Infection in Figure 1.

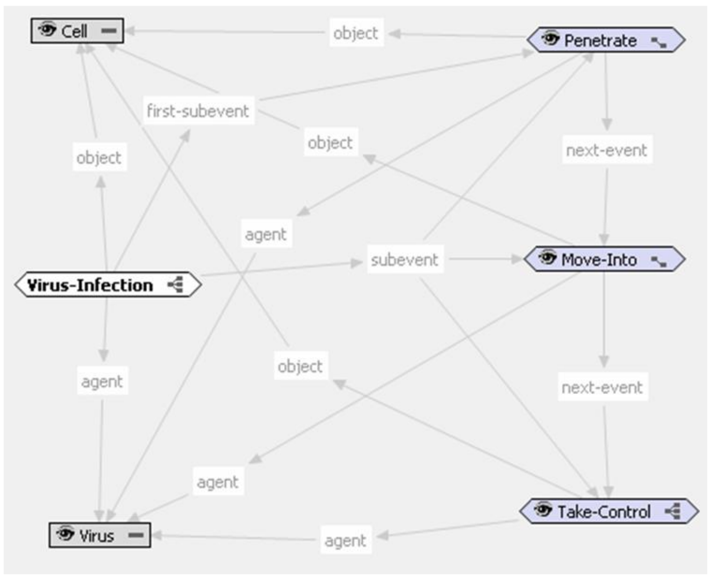


Fig. 1. The Representation of Virus Infection using the CLIB

In this Figure, a KFE has connected three generic CLIB actions: PENETRATE, MOVE-INTO and TAKE-CONTROL, together to first define a sequence amongst those events using the relation *next-event*, and *first-event*, and then specialized those events using relations to VIRUS and CELL.

AURA has undergone substantial testing in its ability to allow KFEs to formulate knowledge in Physics, Chemistry, and Biology showing the effectiveness of this approach for knowledge representation and acquisition [7].

4 Representing CLIB for the Semantic Web

Given the prior success in exploiting CLIB for knowledge representation and acquisition, the time is now ripe to broaden its usage especially in the semantic web community. Doing so will require at least the following steps. First, we need to represent the content of CLIB in a format that is widely used in the semantic web community. OWL is an obvious starting point, but we expect that for fully representing the content of CLIB a language more expressive than OWL will be needed. Second, we need to subject CLIB to community review so that its representations are generally agreed upon and accepted. Such an exercise is consistent with the original goal of CLIB to ensure that the component definitions are non-controversial and represent the consenus view on the definition of the concepts they represent. Such a goal is also consistent with the goal of the ontology patterns portal `ontologydesignpatterns.org`. Finally, we need to start constructing use cases that are of relevance to semantic web that demonstrate how the CLIB representations can be useful for modeling problems other than what it has been used for.

For the rest of the section, we focus on the problem of representing the content of CLIB using semantic web languages. We will first take an example concept from CLIB, and first explain its formal semantics as they are represented in its native representation language KM. Then we show the representation of the same knowledge using OWL, and a rule language SILK.

4.1 CLIB constructs and formal semantics

In order to exemplify what CLIB constructs can be translated into OWL, by preserving as much semantics as possible, we show here the KM axioms for the `Attach` component in CLIB. The `Attach` is an action that "*causes two things to be attached to each other*". We have included only inferentially significant axioms, and omitted the ones that are aimed at natural language generation, or for controlling how they are displayed to the user.

```
(Attach has
  (superclasses (Action))
  (required-slot (object base))
  (primary-slot (agent))
)

(every Attach has
  (object ((exactly 1 Tangible-Entity) (a Tangible-Entity)))
  (base ((exactly 1 Tangible-Entity) (a Tangible-Entity)))

  ;; SOFT PCS:
  (soft-pcs-list (
      (:triple (the base of Self)
                 object-of (mustnt-be-a Be-Inaccessible))))
  (resulting-state ((a Be-Attached-To)))
  (add-list ((:set
        (:triple (the resulting-state of Self)
            object (the object of Self)
            [Attach-add-1])
        (:triple (the resulting-state of Self)
            object (the base of Self)
            [Attach-add-2])))))

(every Attach has
  (preparatory-event ((:default
      (a Make-Contact with
            (object ((the object of Self)))
            (base   ((the base of Self))))
      (a Detach with
            (object ((the object of Self)))
            (base   ((the base of Self))))
))))
```

Informally, the KM code says that `Attach`:

– is subsumed by the more general component `Action`
– is always related to exactly one `object` and one `base`, both of type `TangibleEntity`
– can be related to an `agent`
– has a `resulting-state` of type `Be-Attached-To`, which has *coreferential* links to the `object` and `base` of `Attach`, and (operationally) generates these links when instantiated
– has two *defeasible* `preparatoryEvents` (`(preparatory-event ((:default`): `MakeContact` and `Detach`, which have coreferential links to the `object` and `base` of `Attach`
– has a "soft constraint" (`soft-pcs-list`): the base of `Attach` (coreferential link) cannot be in a state of type: `BeInaccessible`

Although not explicit in the above code, `Action` inherits other properties from its subsuming components (`Action` and `Event`):

– is always related to at least one `instrument` of type `Entity`
– is always related to at least one `subevent` of type `Event` (including itself: a *non-proper* subevent relation)
– can be related to a `nextEvent` of type `Event`
– can be related to a `timeDuring` of type `TimeInterval`

Finally, it has properties that derive from hardcoded functionalities of KM. For example, the following code collects the subevents of any `Event` (then including `Attach` events) into a single list, which can then be simulated by KM (there are several such operational statements that apply to temporal, spatial, agentive, etc. aspects of events.[4]:

[4] The (operational) semantics of the `actions` slot is built into KM

```
(actions ((:set (forall (the subevent of Self)
                        (the actions of It))
                Self)))
```

The expressivity of KM largely exceeds OWL model-theoretical semantics. For example, as explained in 4.2, in OWL (either 1 or 2), coreferential links cannot be declared in general, but only in special cases (transitive properties, property chains). Also, defeasible axioms and soft constraints are not expressible in regular OWL.[5]

Table 1 shows some *reengineering patterns*, presented as correspondences between constructs that are used in CLIB, and their equivalents in OWL1 or OWL2.[6] More correspondences are exemplified in Section 4.2.

KM vs SW Semantics		
KM Construct	SWL	SWL Construct
`superclasses`	*OWL1, OWL2*	`SubClassOf`
`required-slot`	*OWL1, OWL2*	`SubClassOf ObjectSomeValuesFrom`
`primary-slot`	*OWL1, OWL2*	`SubClassOf ObjectMinCardinality 0`
`exactly` n	*OWL1, OWL2*	`ObjectExactCardinality` n
`exactly` n `A`	*OWL2*	`ObjectExactCardinality` n `A`
`superslots`	*OWL1, OWL2*	`SubObjectPropertyOf`
`domain`	*OWL1, OWL2*	`ObjectPropertyDomain`
`instance-of`	*OWL2*	`ClassAssertion`

Table 1. A list of some KM constructs used in CLIB, and their approximation as constructs from Semantic Web languages.

4.2 Representing CLIB using OWL

A translator to convert portions of CLIB into OWL is already available [6]. This translator has been used extensively as part of both the Halo and CALO projects. In the CALO project, the OWL ontology generated by this translator was used to integrate several diverse programming languages and reasoning modules [3]. In the AURA system, the OWL export of CLIB and the content authored by the KFEs was used to define a mapping between the CLIB and an ontology generated through Semantic Media Wiki [5]. But, as expected, this translation is incomplete. As a concrete example, we show below an OWL representation of the `Attach` component that we had shown earlier.

```
SubClassOf(Attach Action)
```

[5] There exist proposals and prototype implementatioms for extending OWL towards non-standard description logics.

[6] We present all OWL formulas in OWL2 Functional Syntax [18].

```
SubClassOf(Attach ObjectSomeValuesFrom(base Tangible-Entity))
SubClassOf(Attach ObjectSomeValuesFrom(object Tangible-Entity))
```

In this OWL representation, there is no representation for the slots that capture qualified cardinality restrictions, meta-level assertions, and the dynamic aspects of `Action`: for example, its `add-list` and `del-list` as from the `Attach` example above. OWL1 does not provide any support for those constructs. OWL2 has enough expressivity: for example, the following constructs can then be added, by means of more reengineering patterns that help finding correspondences to the semantics assumed in KM:

- qualified cardinality restrictions can be expressed natively in OWL2 (see table 1), e.g. `SubClassOf(Attach ObjectExactCardinality (1 base TangibleEntity));`
- meta-level assertions like those used in CLIB for classifying slots (e.g. `(causes (instance-of (CausalRelation)))`, can be represented by means of OWL2 "punning", which provides different interpretations for the constants of an ontology. For example, `ClassAssertion` axioms can be asserted of classes, individuals, or properties without violating the formal semantics of OWL2, e.g. `ClassAssertion(causes CausalRelation);`
- formal axioms for actions can be approximated by using OWL2 property chains. Conditional rules like the one presented in Section 2.1 can be schematized as follows:

  ```
  (if [A R1 B] then (forall [A R2 C] (:triple [It R3 B] [A-add-1])))
  ```

  and can be reengineered by firstly declaring some OWL object properties with appropriate domains and ranges for the antecedent part of the conditional rule:

  ```
  ObjectPropertyDomain(R1 A)
  ObjectPropertyDomain(R1 B)
  ObjectPropertyDomain(R2 A)
  ObjectPropertyDomain(R2 C)
  ObjectPropertyDomain(R3 C)
  ObjectPropertyDomain(R3 B)
  ```

  and then declaring an object property chain axiom:

  ```
  SubObjectPropertyOf(SubObjectPropertyChain(R2- R1) R3)
  ```

  Similarly, definitional rules can be represented by means of property chains, used in restrictions within equivalence axioms. The only rule type that seems completely outside OWL2 is simulation. In that case, it's the dynamics of the process that needs to be simulated in the language, not just represented, and this requires not only an `add-list`, but also a `del-list`, which could only be added programmatically to OWL;

– axioms including coreference are also very difficult to represent in OWL2 (coreference is prevented for complexity reasons). They can be partly represented by property chains, but the actual semantics has a substantial gap. For example, the `add-list` construct for the `Attach` component requires that the resulting state of `Attach` has the same object as the object and base of `Attach`, i.e. the tangible entities that result to be attached after the process. In OWL2 we can assert e.g.:

  ```
  SubClassOf(Attach ObjectExactCardinality(1 base
     ObjectIntersectionOf(ObjectExactCardinality(1 baseOf
        BeAttachedTo) TangibleEntity)))
  ```

  but the semantics does not catch the coreference, so that the BeAttachedTo states of `Attach` and of its base and object can be different. Therefore, given the complexity of this OWL2 construct, and its inability to catch the important part of the original axiom, it seems pretty inadequate to be incorporated into an OWL CP proposal;
– soft constraints and defeasible axioms are not covered by OWL2, but extensions of OWL exist that deal with probabilistic, possibilistic, and other varieties of soft reasoning. We have not yet decided if soft axioms should be provided in knowledge patterns as a general guideline, but we are exploring more evidence of its advantages, and community feedback. Defeasible axioms within universal axioms can be approximated by cutting the defeasible constraint on type, for example:
  `SubClassOf(Attach ObjectSomeValuesFrom(preparatoryEvent Event))`
  (since Event is the range of the slot `preparatoryEvent`).

As a wrap-up, after running all reengineering procedures described above, the `Attach` component gets represented in OWL2 as follows:

```
SubClassOf(Attach Action)
SubClassOf(Attach ObjectExactCardinality (1 base TangibleEntity))
SubClassOf(Attach ObjectExactCardinality (1 object TangibleEntity))
SubClassOf(Attach ObjectMinCardinality (0 agent Entity))
SubClassOf(Attach ObjectSomeValuesFrom(resultingState BeAttachedTo))
SubClassOf(Attach ObjectSomeValuesFrom(preparatoryEvent Event))
```

We do not include the inherited axioms from parent components (`Action` and `Event`).

We hope to investigate the use of OWL2 profiles[7] and OWL-based rule languages [17] to see how more of the information in CLIB could be represented.

4.3 Representing CLIB using Rule Languages

The taxonomic subset of CLIB can be captured to a great extent using the OWL family of languages. Property chains seem to help somehow to harvest

[7] `http://www.w3.org/TR/owl2-profiles/`

more. However, for a complete representation, a rule language is necessary. For example, the Virus Infection example shown earlier is represented in KM as follows:

```
(Virus-Infection has
   (subclass-of (Move)))

(every Virus-Infection has
    (agent ((a Virus)))
    (object ((a Cell)))
    (first-subevent ((the Penetrate sub-event of Self)))
    (subevent ((a Penetrate with
                 (agent ((the agent of Self)))
                 (object ((the object of Self))))
               (a Move-Into with
                  (agent ((the agent of Self)))
                  (object ((the object of Self))))
               (Take-Control with
                  (agent ((the agent of Self)))
                  (object ((the object of Self)))))))
```

Fully representing this axiom will require a rule language. We already have an effort underway to represent such rules using a new semantic web language called SILK (See `http://silk.semwebcentral.org`). SILK is a successor of SWSL and is more expressive than OWL. The above rule can be represented in SILK as follows:

```
?x[agent -> _#1(?x):Virus],
?x[object -> _#2(?x):Cell],
?x[subevent -> _#3(?x):Penetrate [next-event -> _#4,
                                  agent -> _#1,
                                  object -> _#2]]
              _#4(?x):Move-Into [next-event -> _#5,
                                  agent -> _#1,
                                  object -> _#2]]
              _#6(?x):Take-Control [agent -> _#1,
                                     object -> _#2]]
?x[first-subevent -> _#3(?x)]
:- ?x: Virus-Infection.
```

We believe that for fully representing the ontology design patterns in CLIB, an expressive representation language such as SILK is indispensable.

For the editing, manipulation and storage of the knowledge created by the KFEs, AURA uses a representation that is based on a network of individuals - also known as prototypes [9]. A prototype captures a rule whose antecedent is existentially quantified, and its consequent represents a network of existentially quantified individuals. We show the representation of the above rule using prototypes below:

```
(_Virus-Infection2117 has
  (prototype-scope (Virus-Infection))
  (prototype-participants (_Cell2168
                           _Virus2167
                           _Penetrate2166
                           _Move-Into2165
                           _Take-Control2164
                           _Virus-Infection2117))
  (object (_Cell2168))
  (agent (_Virus2167))
  (subevent (_Penetrate2166
             _Move-Into2165
             _Take-Control2164))
  (first-subevent (_Penetrate2166)))

(_Cell2168 has
  (instance-of (Cell)))

(_Virus2167 has
  (instance-of (Virus)))

(_Penetrate2166 has
  (object (_Cell2168))
  (agent (_Virus2167))
  (instance-of (Penetrate))
  (next-event (_Move-Into2165)))

(_Move-Into2165 has
  (object (_Cell2168))
  (agent (_Virus2167))
  (instance-of (Move-Into))
  (next-event (_Take-Control2164)))

(_Take-Control2164 has
  (object (_Cell2168))
  (agent (_Virus2167)))

(Virus-Infection has (superclasses (Move)))
```

It may be possible to represent prototype version of the above rule directly into OWL, but it would require further research to determine if the inferences that are expected of a prototype can also be supported in OWL directly. We are including the prototype representation in this paper to illustrate a possible approach of dealing with the expressiveness limitations of OWL.

It will also be interesting to explore the potential of integrating RIF[8] [2], also including its production rule dialect [13] in order to extend the ontology patterns representation with full-fledged expressivity, while remaining within the W3C standards.

5 Publishing CLIB-CP

We are planning to represent the generic fom CLIB and their most useful axioms into OWL2. We are also translating the metadata that annotate the components. They consist mainly of natural language processing-related code for generating friendly renderings of the components, and of some short comments that summarize the intentional content of each component, for example: "A conceal causes something to be concealed by something else.". However, in the `ontologydesignpatterns.org` initiative, we are interested in a rich annotation of patterns, which provides information about intent, consequences, related patterns, scenarios, competency questions, etc. In the automatic reengineering, we are translating the short comments as strings for the `Intent` value in the pattern annotation schema, and then we are adding the URIs of patterns mentioned within others as related ones. All the other metadata will be possibly introduced by engaging the open communities interested in reusing elements from our portal, or in asking for clarifications.

Several systems to discuss and evaluate patterns have been developed for the ODP portal [12]: they allow to automatically upload an OWL version of a pattern, to ask for reviews and public comments, to provide feedback and reviews, to discuss, to get consensus, and to recommend the patterns as best practices eventually. For the CLIB-based CPs, a script is being implemented to translate and import all at once, but adjusting some parameters.

Once we have a good fraction of CLIB published as ontology patterns on our portal, we will use the review mechanism of the portal to evalute the ontology patterns. For example, each component that is represented as an ontology pattern will be reviewed from the point of view of completeness, accuracy, and to what extent it represents the consensus meaning of that component. The review process will be moderated by a scientific committee overseen by an ODP editorial board.

6 Conclusions

The representation work on CLIB is related to the larger initiative of populating an online, collaboratively-maintained library of ontology design patterns. Since reuse of content patterns requires a critical amount of them, the effort concentrates on one hand on building the community that submits, discusses, and validates pattern proposals, and on the other hand on representing existing resources that have substantial affinity with content patterns. Representation

[8] `http://www.w3.org/TR/rif-rdf-owl/`

practices have been devised until now on FrameNet [11] and the CLIB, which is the contribution of this paper. CLIB is probably the most developed repository of *knowledge* patterns, and has a scope that goes beyond that of content ontology patterns in trying to represent also event dynamics, simulation, etc. However, the part translatable to OWL and rule languages like RIF or SILK proves to be very inline with the ODP initiative, and will constitute an important inventory and a rock-solid resource for the community.

Acknowledgements

This work has been partly supported by the EU projects NeOn, funded within the 6th IST Framework Programme, and IKS, funded within the 7th IST Framework Programme, and by Vulcan Inc. as part of their Project Halo.

References

1. K. Barker, T. Copeck, S.Delisle, and S. Szpakowicz. Systematic Construction of a Versatile Case System. *Journal of Natural Language Engineering*, 3(4):279–315, 1997.
2. H. Boley, G. Hallmark, M. Kifer, A. Paschke, A. Polleres, and D. Reynolds. RIF Core. W3C Working Draft 18 December 2008. http://www.w3.org/TR/2008/WD-rif-core-20081218/, 2008.
3. V. K. Chaudhri, A. Cheyer, R. Guili, B. Jarrold, K. Myers, and J. Niekarasz. A case study in engineering a knowledge base for an intelligent personal assistant. In *International Workshop on Semantic Desktops held during ISWC-2006*, 2006.
4. V. K. Chaudhri, P. E. Clark, S. Mishra, J. Pacheco, A. Spaulding, and J. Tien. AURA: Capturing Knowledge and Answering Questions on Science Textbooks. Technical Report, SRI International, 2009.
5. V. K. Chaudhri, M. Greaves, D. Hansch, A. Jameson, and F. Pfisterer. Using a semantic wiki as a knowledge source for question answering. In *AAAI Spring Symposium on Symbiosis of Semantic Web and Knowledge Engineering*, Stanford, CA, 2008.
6. V. K. Chaudhri, B. Jarrold, and J. Pacheco. Exporting knowledge bases into OWL. In *OWL: Experiences and Directions*, Athens, Georgia, 2006.
7. V. K. Chaudhri, B. John, S. Mishra, J. Pacheco, B. Porter, and A. Spaulding. Enabling experts to build knowledge bases from science textbooks. In *Proceedings of the International Conference on Knowledge Capture Systems*, 2007.
8. P. Clark and B. Porter. Building concept representations from reusable components. In *Proceedings of AAAI'97*, pages 369–376. AAAI press, 1997.
9. P. Clark and B. Porter. Km – the knowledge machine: Users manual. Technical report, University of Texas at Austin, 1999.
10. P. Clark, J. Thompson, and B. Porter. Knowledge Patterns. In A. G. Cohn, F. Giunchiglia, and B. Selman, editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 591–600, San Francisco, 2000. Morgan Kaufmann.
11. B. Coppola, A. Gangemi, A. Gliozzo, D. Picca, and V. Presutti. Frame Detection over the Semantic Web. In *Proceedings of the Fifth European Semantic Web Conference*. Springer, 2009.

12. E. Daga, V. Presutti, and A. Salvati. http: //ontologydesignpatterns.org and evaluation wikiflow. In A. Gangemi, J. Keizer, V. Presutti, and H. Stoermer, editors, *SWAP*, volume 426 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.
13. C. de Sainte Marie, A. Paschke, and G. Hallmark. RIF Production Rule Dialect. W3C Working Draft 18 December 2008. http://www.w3.org/TR/2008/WD-rif-prd-20081218/, 2008.
14. J. Fan, K. Barker, B. Porter, and P. Clark. Representing roles and purpose. In *Proceedings of the International Conference on Knowledge Capture Systems*, 2001.
15. C. Fellbaum, editor. *WordNet. An Electronic Lexical Database*. MIT Press, 1998.
16. A. Gangemi. Ontology design patterns for semantic web content. In *Proceedings of ISWC 2005, Galway, Ireland*, Berlin, 2005. Springer.
17. I. Horrocks and P. F. Patel-Schneider. A proposal for an owl rules language. In *Proceedings of the 13th International Conference on World Wide Web*, pages 723–731, New York, NY, 2004. ACM Press.
18. B. Motik, P. F. Patel-Schneider, and B. Parsia. OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. W3C Working Draft 08 October 2008. http://www.w3.org/TR/2008/WD-owl2-syntax-20081008/, 2008.
19. Ontology Design Patterns Web Portal. http://www.ontologydesignpatterns.org.
20. V. Presutti and A. Gangemi. Content ontology design patterns as practical building blocks for web ontologies. In *Proceedings of the 27th International Conference on Conceptual Modeling - ER2008*, pages 128–141, 2008.
21. W3C OWL Working Group. OWL 2 Web Ontology Language. W3C Working Draft 11 June 2009, http://www.w3.org/TR/2009/WD-owl2-overview-20090611/, 2009.

Pattern-based OWL Ontology Debugging Guidelines

Oscar Corcho[1], Catherine Roussey[2][3], Luis Manuel Vilches Blázquez[1], and Iván Pérez[4]

[1] Ontology Engineering Group, Departamento de Inteligencia Artificial, Universidad Politécnica de Madrid, Spain
[2] Université de Lyon, CNRS, Université Lyon 1, LIRIS UMR5205, Villeurbanne, France
[3] Cemagref, 24 Av. des Landais, BP 50085, 63172 Aubiére, France
[4] IMDEA Software, Madrid, Spain

Abstract. Debugging inconsistent OWL ontologies is a tedious and time-consuming task where a combination of ontology engineers and domain experts is often required to understand whether the changes to be performed are actually dealing with formalisation errors or changing the intended meaning of the original knowledge model. Debugging services from existing ontology engineering tools and debugging strategies available in the literature aid in this task. However, in complex cases they are still far from providing adequate support to ontology developers, due to their lack of efficiency or precision when explaining the main causes for unsatisfiable classes, together with little support for proposing solutions for them. We claim that it is possible to provide additional support to ontology developers, based on the identification of common antipatterns and a debugging strategy, which can be combined with the use of existing tools in order to make this task more effective.

1 Introduction

Ontology engineering methodologies describe the sets of activities to be carried out for ontology building, together with methods and techniques that can be applied to them. Among these activities, those of ontology formalisation and implementation appear in most methodologies, since the final objective is to obtain one or several ontologies that describe the domain according to the ontology requirement specifications provided in the early stages of development.

Formalisation and implementation activities have different degrees of difficulty, considering the knowledge representation formalism and ontology language selected, and the ontology requirements, among others. For example, implementing an RDF(S) ontology is less difficult than implementing an OWL ontology; and developing a small ontology where only primitive concepts are needed is much simpler than developing a network of ontologies where defined concepts are extensively used and complex inferences have to be drawn upon.

Similarly, there are different degrees of difficulty in ontology debugging. However, these are not deeply characterised in existing ontology engineering methodologies and methods (not even in effort estimation approaches like ONTOCOM [12]). Only some works (e.g., [18]) evaluate existing ontology debugging tools, showing the major issues in this task: run-time performance and robustness of the debugging tool result.

If we focus on DL formalisation and OWL implementation, several debugging tools exist (OWLDebugger [5] [4], SWOOP [8], [7], RepairTab [10]), which have proven their effectiveness in different domains, isolating the minimal set of axioms containing a conflict that leads to the unsatisfiability of a class (MUPS). Tools like RepairTab [10] also propose alternatives to resolve the identified conflicts, showing those entailments that would be lost if the proposed solution was applied. Nevertheless solutions are always limited to two choices: removing part of the existing axioms or replacing a class by one of its superclasses. And ontologies used in the experiments (e.g., the mad cow one) have been written by DL experts with the purpose of showing those conflicts.

Our focus is on real ontologies that have been developed by domain experts, who are not necessarily too familiar with DL, and hence can misuse DL constructors and misunderstand the semantics of some OWL expressions, leading to unwanted unsatisfiable classes. To illustrate this, we will use throughout the paper examples taken from a medium-sized OWL ontology (165 classes) developed by a domain expert in the area of hydrology [19]. The first version of this ontology had a total of 114 unsatisfiable classes. The information provided by the debugging systems used ([5], [8]) on (root) unsatisfiable classes was not easily understandable by domain experts to find the reasons for their unsatisfiability. And in several occasions during the debugging process the generation of justifications for unsatisfiability took several hours, what made these tools hard to use, confirming the results described in [18]. As a result, we found out that in several occasions domain experts were just changing axioms from the original ontology in a somehow random manner, even changing the intended meaning of the definitions instead of correcting errors in their formalisations.

Using this and several other real ontologies we have made an effort to identify common unsatisfiability-leading patterns used by domain experts when implementing OWL ontologies, together with common alternatives for providing solutions to them, so that they can be used by domain experts to debug their ontologies. Then we provide some hints about how to organise the iterative ontology debugging process using a combination of debugging tools and patterns.

2 Patterns and AntiPatterns

In software engineering, a design pattern can be defined as a general, proven and beneficial solution to a common re-occurring problem in software design [2]. Built upon similar experiences, design patterns represent best practices about how to build software. On the contrary, antipatterns are defined as patterns that appear obvious but are ineffective or far from optimal in practice, representing worst practice about how to structure and build software [9].

In knowledge (and more specifically in ontology) engineering the concept of knowledge modelling (ontology design) pattern is used to refer to modelling solutions that allow solving recurrent knowledge modelling or ontology design problems [1][14][13][15]. A similar definition is given for knowledge modelling (or ontology design) antipatterns.

Different types of ontology design patterns are defined [14]:

- Logical Ontology Design Patterns (LP). They are independent from a specific domain of interest, but dependent on the expressivity of the logical formalism used for representation. For example, the n-ary relation pattern enables to model n-ary relations in OWL DL ontologies.
- Architectural Ontology Design Patterns (AP). They provide recommendations about the structure of an ontology. They are defined in terms of LPs or compositions of them. Examples are: taxonomy or lightweight ontology.
- Content Ontology Design Patterns (CP). They propose domain-dependent conceptual models to solve content design problems for the domain classes and properties that populate an ontology. They usually exemplify LPs, and they represent most of the work done on ontology design patterns.

In contrast to ontology design patterns, the work on antipatterns is less detailed ([11],[16],[17],[3]). [3] define a set of class metaproperties and associated patterns in order to check subsumption links and correct them. [17] proposed four patterns based on class names in order to detect possible errors in the taxonomic structure. Four logical antipatterns are presented in [11], all of them focused on property domains and ranges. [16] describes common difficulties for newcomers to DL in understanding the logical meaning of expressions. However, none of these contributions groups antipatterns in a common classification, nor provide a comprehensive set of hints to debug them.

2.1 A Classification of Ontology Design AntiPatterns

We have identified a set of patterns that are commonly used by domain experts in their DL formalisations and OWL implementations, and that normally result in unsatisfiable classes or modelling errors. As aforementioned all these antipatterns come from a misuse and misunderstanding of DL expressions by ontology developers. Thus they are all Logical AntiPatterns (LAP): they are independent from a specific domain of interest, but dependent on the expressivity of the logical formalism used for the representation. We have categorized them into three groups:

- Detectable Logical AntiPatterns (DLAP). They represent errors that DL reasoners and debugging tools normally detect.
- Cognitive Logical AntiPatterns (CLAP). They represent possible modelling errors that may be due to a misunderstanding of the logical consequences of the used expression.

- Guidelines (G). They represent complex expressions used in an ontology component definition that are correct from the logical and cognitive points of view, but for which the ontology developer could have used other simpler alternatives or more accurate ones for encoding the same knowledge.

In the rest of this section we describe the antipatterns identified in each group, providing their name and acronym, their template logical expressions and a brief explanation of why this antipattern can appear and how it should be checked by the ontology developer. It is important to note that DLAPs generate unsatisfiable classes that are normally identified by existing ontology debugging tools, although the information that is provided back to the user is not described according to such a pattern, what makes it difficult for ontology developers to find a good solution according to their domain formalisation. With respect to CLAP and G, they are not detected by these tools as such, although in some cases their combination may lead to unsatisfiable classes that are detected (although not appropriately explained) by tools. As we mention in our future work section, we think that tool support for them could be a major step forward in this task.

Finally, all these antipatterns should be seen as elementary units that cause ontology incoherence. That is, they can be combined into more complex ones. However, providing a solution for the individual ones is already a good advance to the current state of the art, and our future work will be also devoted to finding the most common combinations and providing recommendations for them.

2.2 Detectable Logical AntiPatterns (DLAP)

As aforementioned, these antipatterns represent errors that DL reasoners can detect. They can be classified into four main groups: those related to the misunderstanding of the logical conjunction, those related to the incorrect use of universal restrictions, those related to the incorrect use of the combination of universal/existential restrictions and those related to the incorrect representation of disjoint and equivalent knowledge. We now describe them in detail, with examples taken from earlier versions of HydrOntology [19] [5], and with proposed solutions for them, which should be always validated with the domain expert to make sure that the intended meaning of the represented knowledge model does not change.

AntiPattern AndIsOr (AIO) $C_1 \sqsubseteq \exists R.(C_2 \sqcap C_3); Disj(C_2, C_3);$ [6]

[5] All original examples presented in this paper are in Spanish, and we also provide their approximate translation in English for ease of understanding (given the specificity of the domain, not all terms can be translated directly into English). This ontology and several versions obtained throughout the debugging process are available at http://www.dia.fi.upm.es/ ocorcho/OWLDebugging/

[6] This does not mean that the ontology developer has explicitly expressed that C_2 and C_3 are disjoint, but that these two concepts are determined as disjoint from each other by a reasoner. We use this notation as a shorthand for $C_2 \sqcap C_3 \sqsubseteq \bot$.

This is a common modelling error that appears due to the fact that in common linguistic usage, "and" and "or" do not correspond consistently to logical conjunction and disjunction respectively [16]. For example, "I like cake with almond and with chocolate" is ambiguous. Does the cake contain?

- Some chocolate plus some almond? $Cake \sqsubseteq \exists contain.Chocolate \sqcap \exists contain.Almond$;
- Chocolate-flavoured almond? $Cake \sqsubseteq \exists contain.(Chocolate \sqcap Almond)$;
- Some chocolate or some almond? $Cake \sqsubseteq \exists contain.(Chocolate \sqcup Almond)$;

In the original version of HydrOntology this antipattern appeared twice. We present one instance of this antipattern with its approximate translation into English [7] .
$Caño \sqsubseteq \exists comunica.(Albufera \sqcap Mar \sqcap Marisma)$;
$Pipe \sqsubseteq \exists communicate.(Lagoon \sqcap Sea \sqcap Salt\_Marsh)$;

In order to solve this antipattern we propose replacing the logical conjunction by the logical disjunction, or by the conjunction of two existential restrictions.
~~$C_1 \sqsubseteq \exists R.(C_2 \sqcap C_3$~~$)$; $Disj(C_2, C_3); \Rightarrow C_1 \sqsubseteq \exists R.(C_2 \sqcup C_3)$; or
$C_1 \sqsubseteq \exists R.C_2 \; and \; \exists R.C_3$;

AntiPattern OnlynessIsLoneliness (OIL) $C_1 \sqsubseteq \forall R.(C_2); C_1 \sqsubseteq \forall R.(C_3)$; $Disj(C_2, C_3)$; [8]

The ontology developer created a universal restriction to say that C_1 instances can only be linked with property R to C_2 instances. Next, a new universal restriction is added saying that C_1 instances can only be linked with R to C_3 instances, with C_2 and C_3 disjoint. In general, this is because the ontology developer forgot the previous axiom in the same class or in any of the parent classes.

The following is one of the two definitions of HydrOntology class where this antipattern can be found :
$Aguas\_de\_Transición \sqsubseteq \forall está\_próxima.Aguas\_Marinas \sqcap \forall está\_próxima.Desembocadura \sqcap = 1 está\_próxima.\top$;
$Transitional\_Water \sqsubseteq \forall is\_nearby.Sea\_Water \sqcap \forall is\_nearby.River\_Mouth \sqcap = 1 is\_nearby.\top$;

If it makes sense, we propose to the domain expert to transform the two universal restrictions into only one that refers to the logical disjunction of C_2 and C_3.
~~$C_1 \sqsubseteq \forall R.C_2; C_1 \sqsubseteq \forall R.C_3$~~; $Disj(C_2, C_3); \Rightarrow C_1 \sqsubseteq \forall R.(C_2 \sqcup C_3)$;

AntiPatterns UniversalExistence (UE) $C_1 \sqsubseteq \exists R.C_2; C_1 \sqsubseteq \forall R.C_3$; $Disj(C_2, C_3)$;

[7] For better readability, we do not specify in these examples that the used classes are disjoint.

[8] To be detectable, R property must have at least a value, normally specified as a (minimum) cardinality restriction for that class, or with existential restrictions.

The ontology developer adds an existential/universal restriction to a class without remembering that there was already an inconsistency-leading universal/existential restriction in the same class or in a parent class, respectively.

The following is one of 3 examples of this antipattern in HydrOntology:
$Gola \sqsubseteq Canal\_Aguas\_Marinas; Gola \sqsubseteq \exists comunica.Ria;$
$Canal\_Aguas\_Marinas \sqsubseteq \forall comunica.Aguas\_Marinas;$
$Inlets \sqsubseteq Sea\_Waters\_Canal; Inlets \sqsubseteq \exists communicate.Rivers;$
$Sea\_Waters\_Canals \sqsubseteq \forall communicate.Sea\_Waters;$

These antipatterns are difficult to debug because ontology developers sometimes do not distinguish clearly between existential and universal restrictions. Our proposal is aimed at resolving the unsatisfiability of a class, but as usual it should be clearly analysed by the ontology developer.

$C_1 \sqsubseteq \exists R.C_2;$ ~~$C_1 \sqsubseteq \forall R.C_3$~~$; Disj(C_2, C_3) \Rightarrow C_1 \sqsubseteq \forall R.(C_2 \sqcup C_3);$

AntiPattern UniversalExistenceWithInverseProperty (UEWIP)
$C_2 \sqsubseteq \exists R^{-1}.C_1; C_1 \sqsubseteq \forall R.C_3; Disj(C_2, C_3);$

The ontology developer added restrictions about C_2 and C_1 using a property R but he didn't remember that he had already used its inverse property R^{-1}. The following is an example of this antipattern in HydrOntology:
$Aguas\_Marinas \sqsubseteq \exists es\_alimentada.Aguas\_Quietas\_Naturales;$
$Aguas\_Quietas\_Naturales \sqsubseteq \forall alimentada.Aguas\_Corrientes\_Naturales;$
$Sea\_Water \sqsubseteq \exists is\_fed\_by.Natural_Standing_Water;$
$Natural\_Standing\_Water \sqsubseteq \forall feed.Natural\_Watercourse;$

We propose to add the reverse axiom of the C_2 definition $C_1 \sqsubseteq \exists R.C_2$ and if it makes sense to add a class disjunction in the universal restriction.

$C_2 \sqsubseteq \exists R^{-1}.C_1;$ ~~$C_1 \sqsubseteq \forall R.C_3$~~$; Disj(C_2, C_3); \Rightarrow C_1 \sqsubseteq \forall R.(C_2 \sqcup C_3);$

AntiPattern EquivalenceIsDifference (EID) $C_1 \equiv C_2; Disj(C_1, C_2);$

This pattern, which is only common for ontology developers with no previous training in OWL modelling, comes from the fact that the ontology developer wants to say that C_1 is a subclass of C_2, or viceversa, but at the same time it is different from C_2 since he has more information. After a short training session the developer would discover that he really wants to express $C_1 \sqsubseteq C_2$ The following is an example of this antipattern in HydrOntology:
$Cascada \equiv Catarata; Disj(Cascada, Catarata);$
$Cascade \equiv Waterfall; Disj(Cascade, Waterfall);$

We propose to ask the ontology developer whether he really wants to define a synonym or a subclass-of relation. Depending on the ontology developerś answer, the equivalent axiom should be transformed into a subclass-of one or the less used concept should be suppressed according to the SOE recommendations.

~~$C_1 \equiv C_2$~~$; Disj(C_1, C_2) \Rightarrow C_1 \sqsubseteq C_2$ or C_2 is a label of C_1;

2.3 Cognitive Logical AntiPatterns (CLAP)

As aforementioned, these antipatterns are not necessarily errors, but describe common templates that ontology developers use erroneously trying to represent a different piece of knowledge.

AntiPattern SynonymOrEquivalence (SOE) $C_1 \equiv C_2$;

The ontology developer wants to express that two classes C_1 and C_2 are identical. This is not very useful in a single ontology that does not import others. Indeed, what the ontology developer generally wants to represent is a terminological synonymy relation: the class C_1 has two labels: C_1 and C_2. Usually one of the classes is not used anywhere else in the axioms defined in the ontology.

The following is an example of this antipattern in HydrOntology:
Corriente_Subterránea$\equiv$ *Rio_Subterráneo*;
$Subterranean\_Watercourse \equiv Subterranean\_River$;

The proposal for avoiding this antipattern is the following (if C_2 is the less used term in the ontology) add all the comments and labels of C_2 into C_1 and remove C_2.

$\cancel{C_1 \equiv C_2} \Rightarrow C_1.[RDFS : label|comment] = C_2.[RDFS : label|comment]$;

2.4 Guidelines

In contrast to the antipatterns already described, guidelines represent complex expressions used in an ontology component definition that are correct from a logical point of view, but in which the ontology developer could have used other simpler alternatives for encoding the same knowledge. The recommendations provided for Guidelines mainly focus on making the ontology easier to understand by ontology developers, and do not make any change with respect to the semantics or intended meaning of the ontology. We have determined that the proposed changes make the ontology easier to understand by asking a good range of ontology developers about their preferences when analysing ontologies that were not developed by them.

Guideline DisjointnessOfComplement (DOC) $C_1 \equiv not\ C_2$;

During the development process of a new ontology, it is hard to know that C_1 is the logical negation of C_2. Maybe the ontology developper will define later C_3 as a sister class of C_1 and C_2. Thus we recommend to say that C_1 and C_2 cannot share instances first, and change the definition of C_1 as a negation of C_2 if necessary at the end of the development. The following is an example of this antipattern in HydrOntology:
$Laguna\_Salada \equiv not\ Aguas\_Dulces$;
$Salt\_Lagoon \equiv not\ Fresh\_Water$;

We propose: $\cancel{C_1 \equiv not\ C_2} \Rightarrow Disj(C_1, C_2)$;

Guideline Domain&CardinalityConstraints (DCC) $C_1 \sqsubseteq \exists R.C_2$;
$C_1 \sqsubseteq (\geq 2R.\top)$; *(for example)*

Ontology developers with little background in formal logic find difficult to understand that "only" does not imply "some" [16]. This antipattern is a counterpart of that fact. Developers may forget that existential restrictions contain a cardinality constraint: $C_1 \sqsubseteq \exists R.C_2 \vDash C_1 \sqsubseteq (\geq 1R.C_2)$. Thus, when they combine existential and cardinality restrictions, they may be actually thinking about universal restrictions with those cardinality constraints.

The following is an example of this antipattern in HydrOntology:
$Aguas\_de\_Transición \sqsubseteq \exists sometida\_a\_influencia.Aguas\_Dulces \sqcap$
$\exists sometida\_a\_influencia.Aguas\_Saladas \sqcap$
$\forall sometida\_a\_influencia.(Aguas\_Dulces \sqcup Aguas\_Saladas) \sqcap$
$= 1 sometida\_a\_influencia.\top$;
$Transitional\_Water \sqsubseteq \exists is\_influenced\_by.Fresh\_Water \sqcap$
$\exists is\_influenced\_by.Salt\_Water \sqcap$
$\forall is\_influenced\_by.(Fresh\_Water \sqcup Salt\_Water) \sqcap$
$= 1 is\_influenced\_by.\top$;

We propose to transform the existential restriction into a universal one when a cardinality restriction exists.

~~$C_1 \sqsubseteq \exists R.C_2$~~; $C_1 \sqsubseteq (\geq 2R.\top); \Rightarrow C_1 \sqsubseteq \forall R.C_2$;

Guideline GroupAxioms (GA) $C_1 \sqsubseteq \forall R.C_2; C_1 \sqsubseteq (\geq 2R.\top)$; *(for example)*

For visualisation purposes, we recommend grouping all the restrictions of a class that use the same property R in a single restriction. This recommendation is to facilitate the visualisation of complex class definitions.

~~$C_1 \sqsubseteq \forall R.C_2; C_1 \sqsubseteq (\geq 2R.\top)$~~; $\Rightarrow C_1 \sqsubseteq \forall R.C_2 \sqcup (\geq 2R.\top)$;

Guideline MinIsZero (MIZ) $C_1 \sqsubseteq (\geq 0R.\top); \Rightarrow$ ~~$C_1 \sqsubseteq (\geq 0R.\top)$~~;

The ontology developer wants to remember that C_1 is the domain of the R property. This restriction has no impact on the logical model being defined and can be removed at the end of the development process. This antipattern appeared once in the HydrOntology debugging process. $Laguna\_Salada \sqsubseteq (\geq 0 es\_alimentada.\top)$;
$Salted\_Lagoon \sqsubseteq (\geq 0 fedBy.\top)$;

3 Ontology Debugging Strategy

As mentioned in the introduction, OWL ontology debugging features have been proposed in the literature with different degrees of formality ([5], [8], [20]). They allow identifying the main root for unsatisfiable classes and superfluous axioms and restrictions, and in some cases they explain them with different degrees of detail, so that the debugging process can be guided by them and can be made more efficient. However, in general these features are mainly focused on the explanations of logical entailments and are not so focused on the ontology engineering side. Hence explanations are still difficult to understand for ontology developers. Furthermore, there are no clear strategies about how an ontology developer should debug incoherent ontologies, in terms of steps to be followed in this process. Consequently, we think that there is a need to complement both types of suggestions in order to make the ontology debugging process more efficient.

Figure 1 shows graphically a usual ontology debugging lifecycle, with the roles of knowledge engineer and domain expert identified. As it happens in other disciplines (e.g., software development), the first step is to locate where the problems are, using a reasoner directly or an ontology debugging tool, which may also identify root unsatisfiable classes, one of which can be chosen. Otherwise any of the unsatisfiable classes that are in the top of the class hierarchy can be selected. Then antipatterns have to be identified for the selected class. Based on the antipattern identification, the knowledge engineer proposes recommendations for corrections, such as the ones presented in the section 2. Recommendations cannot always be automated, since they may change the intended meaning of the ontology, and should be documented. Once a change is done, new unsatisfiability checks should be performed. This is an iterative process to be followed until there are no more unsatisfiable classes in the ontology.

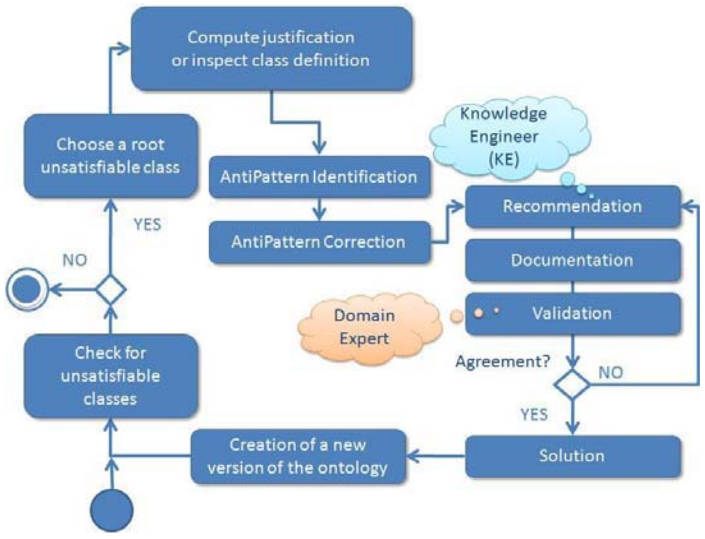

Compute justification or inspect class definition
Choose a root unsatisfiable class
AntiPattern Identification
AntiPattern Correction
Knowledge Engineer (KE)
Recommendation
Documentation
Validation
Domain Expert
Agreement?
NO
YES
NO
YES
Check for unsatisfiable classes
Creation of a new version of the ontology
Solution

Fig. 1. Global strategy for ontology debugging.

Identifying antipatterns at each step in the process may be a hard task, even for experienced ontology developers. Thus we also propose a more detailed strategy, based on the catalogue of antipatterns described in section 2. We propose to follow a specific order, based on our experience, as summarised in Figure 2.

First, we recommend solving terminological problems (SOE), checking the use of equivalence and disjoint constructors between classes (EID, DOC) and applying the guideline GA in order to make formal definitions easier to understand, grouping in the same definition all the axioms dealing with the same role. These are the easiest antipatterns to detect and they are useful to clean other ontology definitions.

Then we propose checking the root unsatisfiable classes in the ontology, using any debugging tool. At that point we can check the use of conjunction (AIO), the use of universal restrictions (OIL), and combinations of universal and existential restrictions (UE, DCC). Sometimes, unsatisfiability arises from a combination of several antipatterns, so that is the reason why there are loops in the figure.

After solving problems in root unsatisfiable classes, the branch of the class hierarchy should be checked manually from the leaf to the root to detect if the same antipattern is present in any class of the branch.

Finally, we recommend removing superfluous axioms (MIZ) to improve the clarity of the ontology. However, this could be really done at any point in time throughout the ontology debugging process.

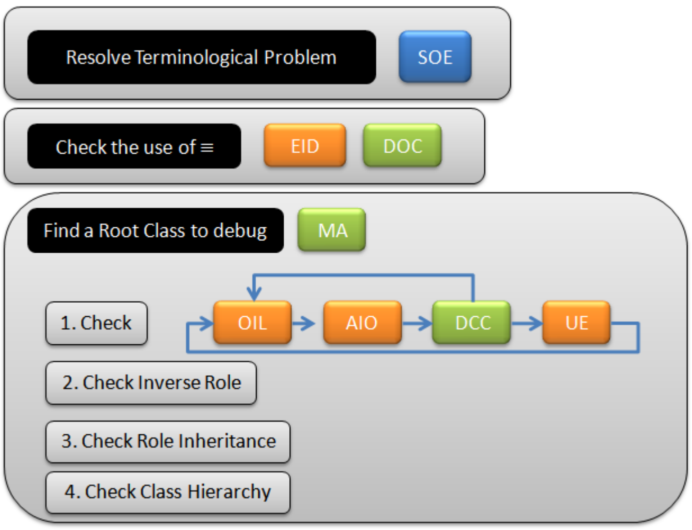


Fig. 2. Detailed debugging strategy based on antipatterns.

4 Evaluation

In order to evaluate our debugging strategy, we conducted a user study with two groups of subjects, using Protégé OWL v4 and its associated explanation workbench [5]. The ontology tested was the first version of HydrOntology, which has been used to provide examples in the previous sections. Table 1 summarises some of its characteristics.

Number of classes	Number of unsatisfiable classes	Number of object properties	Number of datatype properties	Number of class axioms	Average number of axioms per class
165	114	47	64	625	4

Table 1. The characteristics of Hydrontology

4.1 User Study

Fourteen volunteer subjects, who where postgraduate students in the Computing Science Department at the Universidad Politécnica de Madrid, were chosen for the evaluation. Most of these subjects had the same profile: they were experienced in software debugging, they had basic knowledge in description logics and OWL, and they had no knowledge about the Hydrology field.

Our initial hypotheses, which we wanted to test with this study, were:

Hypothesis 1: The subjects using our debugging strategy and our set of antipatterns will take less time to debug the ontology.

Hypothesis 2: The subjects using our set of antipatterns will take less time to find the problematic parts of axioms.

Hypothesis 3: The subjects using our set of antipatterns and their associate recommendations will provide a better solution according to the domain expert than just remove the problematic part of the axioms.

The study was conducted as follows. Each subject was given a tutorial on the debugging of DL axioms and another tutorial on using Protégé v4 with the explanation workbench [4]. None of the subjects had seen the ontology before. They were divided into two similar groups, with similar profiles and similar size. Group 2 was given an additional third tutorial about all the antipatterns, with general examples, and about our proposed debugging strategy.

The subjects in the two groups were asked to answer two surveys , where they were provided with a set of fixed questions to answer (dealing with specific classes), and in all cases they had to specify the time needed to answer each query.

The first survey was about the perceived quality of the ontology. For this survey both groups could only use Protégé to browse the ontology, without the use of any reasoner or the Protégé explanation workbench. Besides, subjects in Group 2 could also use the documentation on antipatterns that we had provided them. The questions were about which parts of the axioms of specific classes would not be taken in account by a reasoner when checking satisfiability, about existing discrepancies between the formal and the natural language definitions of classes, about how to rewrite axioms to make them more understandable, and about the existence of duplicate classes. The second survey was focused on how they would be able to solve existing unsatisfiable classes in the original ontology. They were asked to find the problematic axioms that lead to unsatisfiability in specific classes and provide some solutions. They could use reasoners Fact++ or Pellet1.5 and the Protégé explanation workbench, plus the set of antipatterns in the case of Group 2.

4.2 Analysis of the results

Survey ID	Number of queries	Number of classes involved in the survey	Number of subjects who completed the survey	Average time (in minutes) to complete the survey
S1	44	36	G1: 6 G2: 8	G1: 167 (+31) G2: 136
S2	18	25	G1: 4 G2: 8	G1: 187 G2: 225 (+38)

Table 2. Survey results and characteristics

Hypothesis 1: Concerning the first hypothesis about time, our debugging strategy doesn't seem to reduce the time needed for debugging. As shown in Table 2 the subjects using our strategy (Group 2) completed the first survey in less time than Group 1, but needed more time to complete survey 2. In any case, the differences in time were not very relevant.

However, it is important to notice that most of the subjects complained about the fact that the reasoner crashed regularly while performing debugging activities, and in those cases they found out that the availability of a catalogue of antipatterns was useful to go on working in the meantime.

Hypothesis 2: Concerning the hypothesis related to finding more quickly the errors, the results are mixed. It seems that it depends on the complexity of the antipatterns. The subjects using our set of antipatterns found more quickly the errors related to the SOE, EID, DOC and MIZ antipatterns. The explanation is that these antipatterns are easier to find in an ontology development tool user interface. However, when an unsatisfiable class contains an error which is the concatenation of several antipatterns, subjects in group 2 were not able to perform better than those in group 1. Finally, when an unsatisfiable class contained too many axioms almost none of the subjects managed to find the error. For example the class Río contains 15 axioms and only one subject per group found the error.

Hypothesis 3: Concerning the solution of errors, our recommendations helped in the debugging process. The quality of a solution is evaluated by comparing the result axiom with the one belonging to the final Hydrontology version debugged by a knowledge engineer and the domain expert. When they manage to identify an antipattern, the subjects in group 2 provided a more accurate and precise solution than those in group 1, who were mainly removing axioms randomly in order to make the class satisfiable. That is, without our recommendations the main resolution strategy is still to remove completely the problematic axioms.

5 Conclusions and future work

In this paper we have described a strategy for OWL ontology debugging that can be used in combination with existing OWL ontology debugging services in order to improve the efficiency of the debugging process by having a predefined set of suggested actions to be performed by ontology developers. We have obtained this strategy taking into account our experience in the development of DL-based ontologies and a careful analysis about how ontology developers and ontology engineers debug their ontologies nowadays.

As part of the work that we had to do in order to come up with this strategy, we have collected a list of common antipatterns that can be found in domain-expert-developed ontologies and that cause a large percentage of the unsatisfiability of classes. Besides, we have listed some antipatterns that do not have an impact on the logical consequences of the ontology being developed, but that are of importance in order to reduce the number of errors in the intended meaning of ontologies or to improve their understandability.

For the time being, our strategy is mainly manual, where debugging tools are used to detect some unsatisfiable classes or propose sets of axioms containing antipatterns, although it remains to the user to find out exactly where the antipattern is and which antipattern is applied.

We have evaluated our strategy and compared it with current practice in ontology debugging by using two groups of volunteers that have worked with an incoherent ontology in the geographical domain. As a result, we can confirm that our strategy does not reduce the debugging time but it improves the quality of debugging, that is, our proposed recommendations help finding a more appropriate solution to an error. The other conclusion of our evaluation is that users have some difficulty to find the antipatterns among all the axioms defining an unsatisfiable class.

Hence as part of our future work we are aiming at implementing additional tools that can be used in combination with existing debugging tools (e.g., the Protégé explanation workbench) to help in the identification of antipatterns. For the time being we have started applying the OPPL language [6] for this task, with promising results.

Another part of future work will be related to applying this strategy for the debugging of well-known inconsistent ontologies (e.g., TAMBIS). In this case we would be extending our work to that of experts debugging ontologies that have not been written by them. And we will also focus on how anti-patterns are usually combined together and how more complex ones can be found that can speed up even more the debugging process.

Finally, some of the explanations that we have provided for the appearance of antipatterns are related to the order in which some of the restrictions and axioms have been added to the ontology. Hence keeping a record of the changes that have been made to the ontology following well-known ontology change management systems could be useful in order to incorporate this into the antipattern detection phases and providing possible ranked solutions to them.

Acknowledgements This work is a result of collaboration between OEG, LIRIS lab and IMDEA Software. It has been done under the context of the project GeoBuddies, funded by the Spanish Ministry of Science and Technology and it was also partially funded by the COST Action C21 sponsored by the European Commission under the grant number STSM-C21-04241. The work of IMDEA Software on this paper has been partially funded by the Spanish Ministry of Industry, Tourism and Trade under the grant FIT-340503-2007-2, as part of the Morfeo EzWeb strategic singular project.

References

1. Clark P, Thompson J, Porter B.: Knowledge patterns. In Proceedings of 7th International Conference Principles of Knowledge Representation and Reasoning (KR), Breckenridge, Colorado, USA: 591-600. (2000)
2. Gamma E, Helm R, Johnson R, Vlissides J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2. (1995)
3. Guarino N and Welty C: Evaluating Ontological Decisions with OntoClean. Communications of the ACM. 45(2):61-65. New York:ACM Press, (2002).
4. Horridge M.: Understanding and Repairing Inferences. Tutorial in the 11th Intl. Protégé Conference - June 23-26, 2009 - Amsterdam, Netherlands.
5. Horridge M, Parsia B, Sattler U.: Laconic and Precise Justifications in OWL. In Proceedings of the 7th International Semantic Web Conference (ISWC), Karlsruhe, Germany; LNCS 5318: 323-338. (2008).
6. Iannone L, Rector A, Stevens R.: Embedding Knowledge Patterns into OWL. In proceedings of the 6th European Semantic Web Conference (ESWC2009), Crete, Greece. The Semantic Web: Research and Applications (2009), pp. 218-232
7. Kalyanpur A, Parsia B, Cuenca-Grau B.: Beyond Asserted Axioms: Fine-Grain Justifications for OWL-DL Entailments. Description Logics 2006
8. Kalyanpur A, Parsia B, Sirin E, Cuenca-Grau B.: Repairing Unsatisfiable Classes in OWL Ontologies. In Proceedings of the 3rd European Semantic Web Conference (ESWC), Budva, Montenegro; LNCS 4011: 170-184 (2006)
9. Koenig A.: Patterns and Antipatterns. Journal of Object-Oriented Programming 8(1):46-48. (1995)
10. Lam J, Pan JZ, Sleeman D, Vasconcelos W. A Fine-Grained Approach to Resolving Unsatisfiable Ontologies. Journal of Data Semantics (JoDS) 10:62-95. 2008
11. Laboratory of Applied Ontology: Collection of antipatterns from http://wiki.loa-cnr.it/index.php/LoaWiki:MixedDomains
12. Paslaru E, Simperl B, Popov I O, Bürger T.: ONTOCOM Revisited: Towards Accurate Cost Predictions for Ontology Development Projects. In Procs. of the 6th European Semantic Web Conference, ESWC 2009, Heraklion, Greece, June 2009, LNCS 5554: 248-262 (2009)

13. Presutti V., Gangemi A.: Content Ontology Design Patterns as practical building blocks for web ontologies. In Proceedings of the 27th International Conference on Conceptual Modeling (ER), Barcelona, Spain, LNCS 5231: 128-141(2008).
14. Presutti V, Gangemi A, David S, Aguado G, Suarez-Figueroa MC, Montiel E, Poveda M.: Neon Deliverable D2.5.1: A Library of Ontology Design Patterns available at <http://www.neon-project.org>
15. Rech J, Feldmann R L, Ras E.: Knowledge Patterns. In M. E. Jennex (Ed.), Encyclopedia of Knowledge Management (2nd Edition), IGI Global, USA, (2009).
16. Rector AL, Drummond N, Horridge M, Rogers L, Knublauch H, Stevens R, Wang H, Wroe C.: OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns. In Proceedings of the 14th International Conference Knowledge Acquisition, Modeling and Management (EKAW), Whittlebury Hall, UK. LNCS 3257: 63-81 (2004)
17. Sváb-Zamazal O, Svátek V: Analysing Ontological Structures through Name Pattern Tracking. In Proceedings of the 16th International Conference, EKAW 2008, Acitrezza, Italy, September 29 - October 2, 2008. Lecture Notes in Computer Science 5268 Springer 2008, ISBN 978-3-540-87695-3: 213-228 (2008)
18. Stuckenschmidt H.: Debugging OWL Ontologies - a Reality Check. In Proceedings of the 6th International Workshop on Evaluation of Ontology-based Tools and the Semantic Web Service Challenge (EON-SWSC-2008), Tenerife, Spain. (2008).
19. Vilches-Blázquez LM, Bernabé-Poveda MA, Suárez-Figueroa MC, Gómez-Pérez A, Rodríguez-Pascual AF: Towntology & hydrOntology: Relationship between Urban and Hydrographic Features in the Geographic Information Domain. In Ontologies for Urban Development. Studies in Computational Intelligence, vol. 61, Springer: 73-84. (2007)
20. Wang, H., Horridge M, Rector A, Drummond N, Seidenberg J.: Debugging OWL-DL Ontologies: A heuristic approach. In Proceedings of the 4th International Semantic Web Conference (ISWC), Galway, Ireland; LNCS 3729: 745-757(2005)

eXtreme Design with Content Ontology Design Patterns

Valentina Presutti and Enrico Daga and Aldo Gangemi and Eva Blomqvist

Semantic Technology Laboratory, ISTC-CNR

Abstract. In this paper, we present eXtreme Design with Content Ontology Design Patterns (XD): a collaborative, incremental, iterative method for pattern-based ontology design. We also describe the first version of a supporting tool that has been implemented and is available as a plugin for the NeOn Toolkit. XD is defined in the context of a general approach to ontology design based on patterns, which is also briefly introduce in this work.

1 Introduction

Ontology design patterns (ODPs) [7] are an emerging technology that favors the reuse of encoded experiences and good practices. ODPs are modeling solutions to solve recurrent ontology design problems. They can be of different types[1] including: *logical*, which typically provide solutions for solving problems of expressivity e.g., expressing n-ary relations in OWL; *architectural*, which describe the overall shape of the ontology (either internal or external) that is convenient with respect to a specific ontology-based task or application e.g. a certain DL family; *content*, which are small ontologies that address a specific modeling issue, and can be directly reused by importing them in the ontology under development e.g., representing roles that people can play during certain time periods; *presentation*, which provide good practices for e.g. naming conventions; etc.
With the name **eXtreme Design (XD)** we identify an approach, a family of methods and associated tools, based on the application, exploitation, and definition of ontology design patterns (ODPs) for solving ontology development issues. In this paper, we describe XD and go into details of its guidelines for ontology development with Content ODPs (CPs). Also we briefly describe the prototype of a supporting tool i.e. the XD plugin for the NeOn Toolkit.
XD adopts the notion of ontology project, a development project characterized by two main sets: (i) the *problem space*, which is composed of the actual modeling issues, here referred to as the *local problems*, that have to be addressed during the project e.g., to transform a set of microformats to an RDF dataset, to model roles that can be played by people during certain time periods; (ii) the *solution space*, which is made up of reusable modeling solutions e.g. a reengineering practice for associating microformats' attributes to a certain RDF vocabulary's

[1] http://ontologydesignpatterns.org/wiki/OPTypes

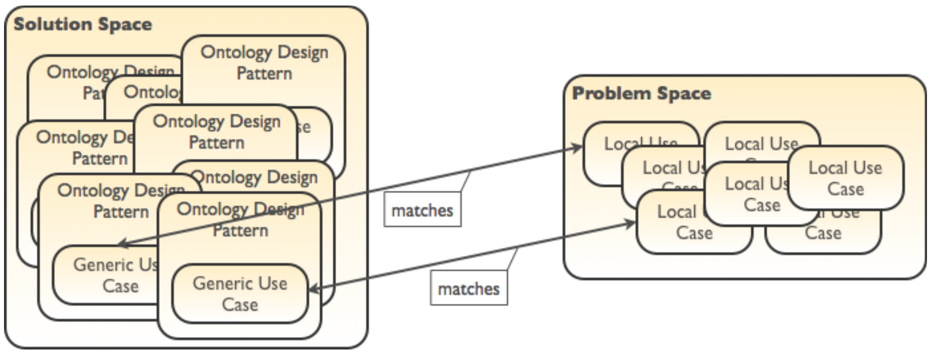


Fig. 1. The eXtreme Design approach. ODPs are associated with Generic Use Cases and compose the ontology project's *solution space*, which is used as the main knowledge source for addressing ontology design issues e.g. reengineering, evaluation, construction, etc., the ontology project's *problem space* provides descriptions of the actual issues called "Local Use Cases".

relations, a piece of an ontology that models time-indexed roles i.e. a CP.
The general approach is schematized in Figure 1. Each element in the solution space is an ODP associated with a Generic Use Case (GUC), the latter representing the problem that the ODP provides a solution for, as introduced by [6]. The elements of the problem space are called "Local Use Case" (LUC), they define the actual modeling issues that need to be addressed in order to work out the ontology project, they represent the ontology project's requirements. Under the assumption that GUCs and LUCs are represented in a compatible way e.g., both in the form of competency questions or sentences, it is possible to compare LUCs to GUCs, and if they match, the ODPs associated with the matching GUCs are selected and reused for building the final solution. Informally, a GUC matches a LUC, if the latter can be completely or partly described exactly in terms of the GUC, or as a more specific case of it; or if the LUC can be described in terms of part of the GUC.
All matching ODPs are selected and used according to specific guidelines and possibly with some tool support.
In this paper, we focus on XD guidelines for CPs, where GUCs and LUCs are expressed in the form of natural language competency questions, and ODPs are CPs. In the rest of the paper XD is used for referring to XD with CPs.
XD is partly inspired by software engineering eXtreme Programming (XP) [11], and experience factory[12, 1]. The former is an agile software development methodology the aim of which is to minimize the impact of changes at any stage of the development, and producing incremental releases based on customer requirements and their prioritization. The latter is an organizational and process approach for improving life cycles and products based on the exploitation of past experience know-how.
Although XD has similarities with the two approaches, 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 well as XP does. Also, XD adopts pair design, as opposed to pair programming. The intensive use of CPs, modular design, and collaboration are the main principles of the method. While a rigorous evaluation of the whole methodology is still in our future plans, the effectiveness of CPs in ontology design has been rigorously evaluated in [5], where XD has been used as reference development guidelines. Furthermore, initial questionnaires and informal discussions made emerge that the perception of the trainees with respect to the method is positive. The contribution of this paper is twofold: (i) a collaborative, incremental, and iterative method for pattern-based ontology design, called eXtreme Design (XD); (ii) a first version of the XD tool, a NeOn Toolkit plugin that currently supports CP repository browsing and selection, a good practice assistant, and a wizard for CPs' specialization.
The paper is structured as follows: in Section 1.1 we briefly discuss the state of art on ontology design methodologies and ontology design patterns. Section 2 discusses the principles of the method and details the XD workflow with the help of a simplified scenario taken from a real case study. Section 3 describes XD tool, a NeOn Toolkit[2] plugin for pattern-based design support.

1.1 State of the art and related work

The notion of "pattern" has proved useful in the context of design within many areas, such as architecture, software engineering, etc. So far, very few purely pattern-based methodologies have been proposed. In ontology engineering, pattern-based methods are present primarily on the logical level, where patterns support methods for ontology learning, enrichment and similar tasks like in [2]. In these methods patterns are used more or less automatically, e.g. lexico-syntactic patterns to identify ontological elements in a natural language text or to extract relations between ontology concepts. In [3], a method for constructing ontologies based on patterns is proposed, the difference between this method and XD is that the former do not consider collaboration, and that the patterns were assumed to be a non-evolving set mostly defined with a top-down approach. Another related approach is that described in [8], where competency questions have been introduced. XD takes inspiration from this work, specially for the use of competency questions as reference source for the requirement analysis. However, the methodology described in [8] do not consider modular nor pattern-based design approach, and do not address collaborative development. Many other ontology development methodologies have been proposed, an example is the DILIGENT methodology [9], which takes into account collaborative aspects, but do not consider the use of patterns and is not test-oriented. From the software engineering field we can mention the eXtreme Programming methodology, which XD is inspired by. However, the focus of XD is completely different from that of XP, although they share some base principles such as pair

[2] http://www.neon-toolkit.org

design as opposed to pair programming, test-driven development, and customer involvement.

2 Guidelines for XD with Content Ontology Design Patterns

In this section, we describe the methodological guidelines for applying XD with Content ODPs (CPs), through the definition of an iterative workflow and a case example showing an actual iteration.
The XD method with CPs is the result of the observation and consequent description of the way we (in our lab) use to develop ontologies with CPs. Since 2005, we have been developing CPs, teaching pattern-based ontology design in conference tutorials and PhD courses, and for much longer we have been using and refining this approach for our professional work.
In order to teach pattern-based design to PhD students and practitioners, we needed to provide trainees with guidelines to follow. This requirement provided us with a good occasion for defining the XD method with CPs, and also with a context for running the method with different teams, and applying possible refinement/adjustment. So far, we have identified the main principles and setting of the method, defined the iterative workflow, identified a set of requirements for tool support, started supporting tools development, and identified and started investigation of open issues.

XD principles. XD principles are inspired by those of the agile software methodology called eXtreme Programming (XP) [11]. The main idea of agile software development is to be able to incorporate changes easily, in any stage of the development. Instead of using a waterfall-like method, where you first do all the analysis, then the design, the implementation and finally the testing, the idea is to cut this process into small pieces, each containing all those elements but only for a very small subset of the problem. The solution will grow almost organically and there is no "grand plan" that can be ruined by a big change request from the customer.
The XD method is inspired by XP in many ways but its focus is different: where XP diminishes the value of careful design, this is exactly where XD has its main focus. Of course, designing software and designing ontologies is inherently different, but still there are many lessons to be learnt from programming. XD is test-driven, and applies the divide-and-conquer approach as well as XP does. Also, XD adopts pair design, as opposed to pair programming. Although we did not perform yet a formal evaluation of pair design's effectiveness, we have collected trainees feedback through informal discussions and questionnaires after the executions of XD with different trainees teams. Most of them feel to take benefit from on-the-fly brainstorming, and perceive to improve the effect of learning-by-doing with this setting. We have planned to conduct more rigorous evaluation of the method which also involves the analysis of this aspect.
The intensive use of CPs, modular design, and collaboration are the main prin-

ciples of the method. The effectiveness of CPs in ontology design has been rigorously evaluated in [5], where XD has been used as reference development guidelines.
The main principles of the XD method can be summarized as follows:

– **Customer involvement and feedback.** The development of an ontology is part of a bigger picture, where typically a software project is under development. Ideally, the customer should be involved in the ontology development and its representative should be a team, whose members are aware of all parts and needs of the project. For example, the roles that should be represented include: domain experts i.e. persons with deep knowledge of the domain to be described by the ontology; those who are in charge of maintaining the knowledge/data bases i.e. persons who know the views over the data that are usually required by users; those who control/coordinate organization processes i.e. persons who have an overall view on the entire flow of data; etc. Depending on the project characteristics, and on the complexity of the organization, the customer representative can be one person or a team. It is important that the team of designers is able to easily interact with the customer representative in order to minimize the possible number of assumptions that they have to make on the incomplete requirement descriprions i.e. assumptions on the implicit knowledge, without discussing/validating them first. Interaction with the customer representative is key for favoring the explicit expression of knowledge that is usually implicit in requirement documents, including competency questions. Furthermore, the customer representative should be able to describe what tasks the application involving the ontology is expected to solve.
– **Customer stories, Competency Questions (CQs), and contextual statements.** The ontology requirements and its tasks are described in terms of small stories by the customer representative. Designers work on those small stories and, together with the customer, transform them in the form of CQs and contextual statements. Contextual statements are accompanying assertions that explicit knowledge that is typically implicit in CQs. CQs and contextual statements will be used through the whole development, and their definition is a key phase as the designers have the challenge to help the customer in making explicit as much implicit knowledge as possible.
– **CP reuse and modular design.** If there is a CP's GUC that matches a LUC it has to be reused, otherwise a new module i.e. a CP with its GUC, is defined based on the LUC under development and shared with the team (and ideally on the Web[3]).
– **Collaboration and Integration.** Integration is a key aspect of XD as the ontology is developed in a modular way. Collaboration and constant sharing of knowledge is needed in a XD setting, in fact similar or even the same CQs and sentences can be defined for different stories. When this happens, it is important e.g. that the same CP is reused.

[3] For example on http://ontologydesignpatterns.org

- **Task-oriented design** The focus of the design is on that part of the domain of knowledge under investigation that is needed in order to address the user stories, and more generally, the tasks that the ontology is expected to address. This is opposed to the more philosophical approach of formal ontology design where the aim is to be comprehensive with respect to a certain domain.
- **Test-driven design.** Stories, CQs, and contextual statements are used in order to develop unit tests. A new story can be treated only when all unit tests associated with it have been passed. This aspect enforces the task-oriented approach of the method. It has to be noticed that in this context, "unit tests" have a completely different meaning with respect to software engineering unit tests. An ontology module developed for addressing a certain user story associated to a certain competency question, is tested e.g. (i) by encoding in the ontology[4] a sample set of facts based on the user story, (ii) defining one or a set of SPARQL queries that formally encode the competency question, (iii) associating each SPARQL query with the expected result, and (i) running the SPARQL queries against the ontology and compare actual with expected results. Unit tests for ontologies have been analyzed already in [13], where the focus is more on purely logical structures. We leave the investigation of unit test types, and their employment in XD, at future developments.
- **Pair design.** The team of designers is organized in pairs. At least one pair is in charge of integrating ontology modules.

The next section shows details of the XD iterative workflow.

2.1 XD iterative workflow

Figure 2 shows the workflow of XD with CPs. In this section we will describe the single tasks with the help of a simplified scenario coming from a real case study in the Fishery domain. XD is an incremental, iterative method for pattern-based ontology development. Before entering the details of each single task, it is worth to make few premises. The team of designers is organized in pairs that work in parallel. At least one pair is in charge of integrating the modules produced by the other pairs, in order to obtain incremental releases of the ontology. A wiki for the project is set up with a basic structure able to collect customer stories and their associated modeling choices, testing documentation, and contextual statements. The wiki will be used in order to build incrementally the project documentation. During the development, and in particular for testing purposes, an ontology module containing instances according to the customer stories is created and shared. This module is used in order to run unit tests against the ontology.

[4] According to the common way of using the term "ontology", in this context we do not distinguish between TBox and ABox, or ontology and knowledge base. Here, an ontology includes also facts.

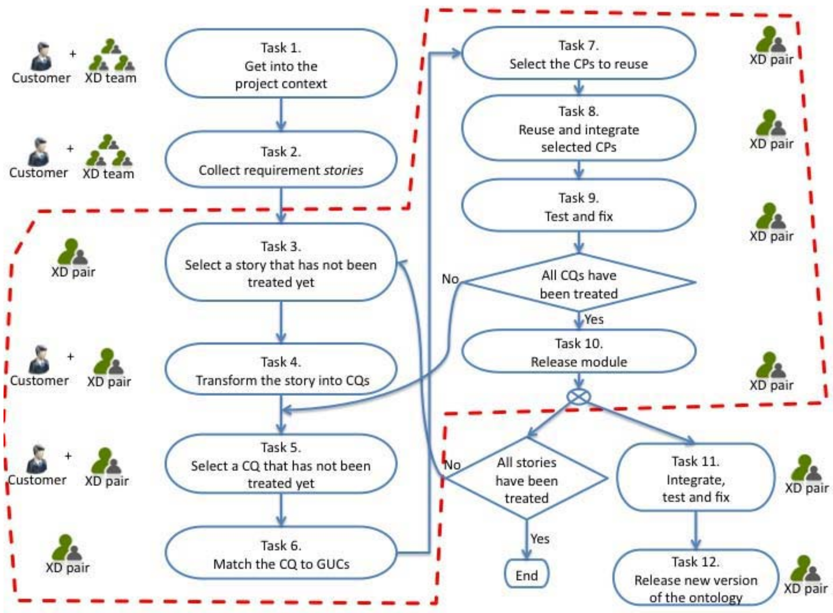

Customer
XD team
Task 1. Get into the project context
Task 2. Collect requirement stories
Task 3. Select a story that has not been treated yet
Task 4. Transform the story into CQs
Task 5. Select a CQ that has not been treated yet
Task 6. Match the CQ to GUCs
Task 7. Select the CPs to reuse
Task 8. Reuse and integrate selected CPs
Task 9. Test and fix
All CQs have been treated
No
Yes
Task 10. Release module
All stories have been treated
Task 11. Integrate, test and fix
Task 12. Release new version of the ontology
End
XD pair

Fig. 2. The XD iterative workflow.

Task 1. Get into the project context. The development starts with a group of designers and a group of domain experts i.e. the customer representative. In principle they do not know much about each other, do not have a precise idea of what will be the result of the project, are used at different terminology, and have a different background. This task has a twofold objective: (i) make the customer representative aware of the method and tools that will be applied during the project, (ii) provide the designer team with an overview of the problem, its scope, and initial terminology.
The result of this task is the setting up of a collaborative environment where customer and designers will share documentation and argument about modeling issues, including terminology i.e. a wiki for the project.

Task 2. Collect requirement stories. The customer representative is invited to write stories, possibly from real, documented scenarios, that samples the typical facts that should be stored in the resulting ontology[5]. All stories are organized in terms of priority, and possible dependencies between them are identified and made explicit[6]. Each story is described by means of a small card,

[5] We do not distinguish between TBox and ABox, ontology and knowledge base. With the term ontology we encompass both according with the current trend in the Semantic Web community.

[6] E.g., a story can be modeled only if another story already has been successfully addressed

like the one depicted in Table 1, which includes the story's title, a list of other stories which it depends on, a description in natural language, and a priority value[7]. It is important to notice that this task is not intended to be performed only once during the project. Stories can be added by the customer during the whole project life cycle. For example, if a new requirement emerges new stories can be written.

Task 3. Select a story that has not been treated yet. Each pair of designers selects a story that will be the focus of their work for the next iteration. A new wiki page for the story is created: the name of the page is the title of the story, and its content is set up based on the information that are in the card. By performing this task a pair enter a development iteration. For example, consider that a pair has selected the story described by the card in Table 1.

Table 1. A requirement story card. It includes the story's title, a list of other stories which the story depends on, a natural language description, and a priority value.

Title	Tuna observation
Depends on	Exploitation values, Tuna areas
Description	In 2004 the resource of species "Tuna" in water area 24 was observed to be fully exploited in the tropical zone at pelagic depth.
Priority	High

Task 4. Transform the story into CQs. The pair process the story and from it derive a set of CQs. In order to do that, designers could involve the customer for having feedback/clarifications. First the story is split into simple sentences, meaning that complex example sentences may be broken up into shorter sentences to increase clarity. The sentences are abstracted so that they describe a class of facts instead of a specific one. The sentences are then transformed into CQs. For example, the story "Tuna observation" is transformed to the following CQs[8], which are added in the story wiki page:

– CQ_1: What are the exploitation state and vertical distance observed in a given climatic zone for a certain resource?
– CQ_2: What resources have been observed during a certain period in a certain water area?

Additionally, the following contextual statement is derived from the discussion with the customer representative[9]:

– A resource contains one or more species.

[7] Priority values are assigned by designers based on interaction with the customer representative. The values can vary and depends on project's conventions.
[8] The elaboration of the story is a complex cognitive procedure which is not explained here. It would deserve a dedicated discussion which is out of scope of this paper.
[9] There would be additional ones, we have simplified for the sake of brevity.

– Species are associated to vertical distances. As a consequence, the vertical distance of a resource is inferred through the vertical distance of the species.

Contextual statements are listed in a dedicated wiki page, and are handled by the pair in charge of the integration task.

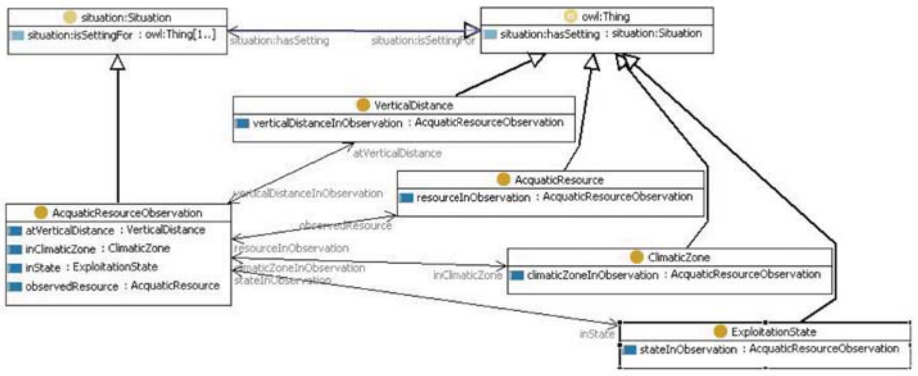
Task 5. Select a CQ that has not been treated yet. The iteration continues by selecting one of the CQs. For example, CQ_1.

Task 6. Match the CQ to GUCs. This task has the aim of identifying candidate CPs based on the CQ, which express part of the LUC. The matching procedure can be done either with some tool support e.g., keyword based searching, or manually e.g., if the designers have a good knowledge of available CPs. We here assume that designer manually perform the matching against the ontologydesignpatterns.org repository [4] of CPs. In our example, candidate CPs are: *situation*, and *time interval*All of them are available on http://ontologydesignpatterns.org. The competency question of *situation* *"What entities are in the setting of a certain situation?"* can be said to match the observation, the resource, and the parameters that are in the setting of that observation. Additionally, the *time interval* CP may be seen as partially matching the question of what period a certain observation was made, although this could also be solve with just a simple datatype property. The CP contains CQs such as: *"What is the end time of this interval?, What is the starting time of this interval?, What is the date of this time interval?"*. The result of this task is then two matching CPs.

Task 7. Select the CPs to reuse. The goal of this task is to select which of those patterns should be used for solving the modeling problem. We may decide that *time interval* adds too much extra effort, besides the needed year of observation, in which case we will only select *situation*.

Task 8. Reuse and integrate selected CPs. The term "reuse" here refers to the application of typical operation that can be applied to CPs i.e. import, specialization, and composition [7]. In our example, we specialize *situation* in order to address CQ_1. The particular situation is in our case the observation, and the thing observed is the resource. Additionally, the exploitation state, climatic zone, and vertical distance of the observation, are also involved in the setting. Thereby, we add a subclass of `situation:Situation`[10] named `AquaticResourceObservation`, and add the other entities as subclasses of `owl:Thing`. In addition, we construct subproperties of the `situation:isSettingFor` and its inverse `situation:hasSetting`, for connecting the observations to the resources and the different parameters. The result is shown with a UML diagram in Figure 3. In this case we have shown a simplified example where only one CP has been reused and specialized. In other cases, we might reuse more CPs. Each of them would be first specialized then integrated with the others. The process

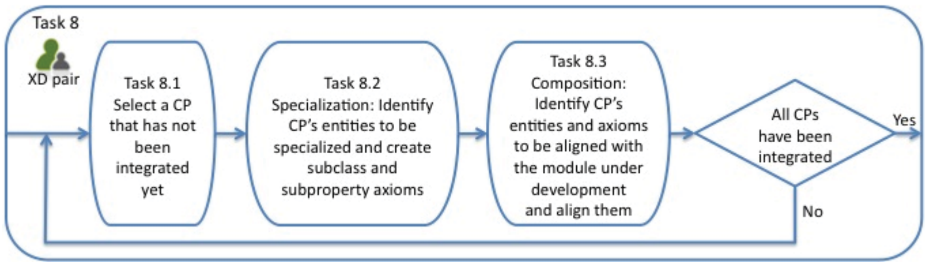
[10] The prefix "situation:" is for http://ontologydesignpatterns.org/cp/owl/situation.owl, while "Situation" is a class defined in the *situation* CP.



situation:Situation
situation:isSettingFor : owl:Thing[1..]
owl:Thing
situation:hasSetting : situation:Situation
situation:hasSetting
situation:isSettingFor
VerticalDistance
verticalDistanceInObservation : AcquaticResourceObservation
atVerticalDistance
AcquaticResource
resourceInObservation : AcquaticResourceObservation
verticalDistanceInObservation
observedResource
AcquaticResourceObservation
atVerticalDistance : VerticalDistance
inClimaticZone : ClimaticZone
inState : ExploitationState
observedResource : AcquaticResource
resourceInObservation
climaticZoneInObservation
stateInObservation
inClimaticZone
ClimaticZone
climaticZoneInObservation : AcquaticResourceObservation
inState
ExploitationState
stateInObservation : AcquaticResourceObservation

Fig. 3. The *acquatic resource observation* ontology module that specializes the *situation* CP.

that is typically performed during this task is sketched in Figure 4.



Task 8
XD pair
Task 8.1 Select a CP that has not been integrated yet
Task 8.2 Specialization: Identify CP's entities to be specialized and create subclass and subproperty axioms
Task 8.3 Composition: Identify CP's entities and axioms to be aligned with the module under development and align them
All CPs have been integrated
Yes
No

Fig. 4. The process performed in order to execute Task 8 "Reuse and integrate selected CPs".

Task 9. Test and fix. The goal of this task is to validate the resulting module with respect to the CQ just modeled. To this aim, the task is executed through the following steps: (i) the CQ is elaborated in order to derive a unit test e.g., SPARQL query; (ii) the instance module is fed with sample facts based on the story; (iii) the unit test is ran against the ontology module. If the result is not the expected one i.e. the test is not passed, the module is revised in order to fix it, and the unit test ran again until the test is passed; (iv) run all other unit tests associated with the story so far until they all pass. Notice that all unit tests are described in dedicated wiki pages that are properly linked to the associated story. If all CQs associated to the story have been addressed, the pair can pass to Task 10, otherwise they "go back" to Task 5. In our example, the unit test associated to CQ_1 is the following:

```
SELECT ?exp ?dist ?resource ?zone
WHERE {
?obs a :AquaticResourceObservation .
?obs observedResource ?resource .
?obs inClimaticZone ?zone
?obs inState ?exp .
?obs atVerticalDistance ?dist
}
```

Task 10. Release module. This task identifies the end of an iteration for a pair and its result is an ontology module. Once a whole story has been addressed, and the resulting module has been successfully tested, the new module can be released. The module is assigned with a URI and published in order to be shared by the whole team. If the module can be publicly shared, it can be published in open Web repositories such as ontologydesignpatterns.org. The module is then passed to the pair in charge of the integration. The pair of designer selects a new story if there are still some unaddressed.

Task 11. Integrate, test and fix. Once a new module is released, it has to be integrated with all the others that constitute the current version of the ontology. At least one pair is in charge of performing integration and related tests: new unit tests are defined for the integration, and all existing ones are again executed as regression tests before moving to next task. In this task, all contextual statements are taken into account and all necessary alignment axioms are defined. The module is now under the complete control and editing of the pair in charge of the integration. The products of this tasks are new unit tests and alignment axioms, all properly documented in the wiki.

Task 12. Release new version of the ontology. Once all unit tests have been passed, a new version of the ontology can be released.

3 XD tools for the NeOn toolkit

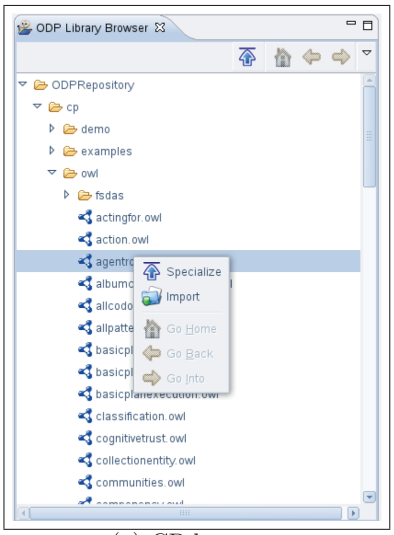
The **eXtreme Design** plugin for NeOn Toolkit[11] (XD tool) supports pattern-based ontology design. Its current version[12], focusses on XD with CPs, and supports some of the tasks described in Section 2. The main goal of XD tool is to improve the user experience with ontology design editors by providing them with a new interaction approach to ontology design. Instead of performing language-oriented operations e.g. instantiate a class, define a subclass, etc., the designer handles "pieces" of ontologies i.e. CPs, and is helped in performing modular design and applying design good practices.
The tool provide an Eclipse perspective, named "eXtreme Design", that includes

[11] http://www.neon-toolkit.org
[12] Available at http://stlab.istc.cnr.it/stlab/Download

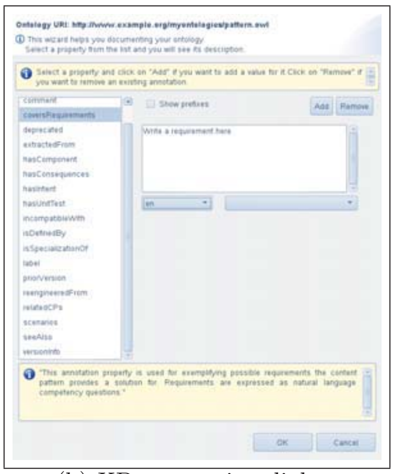
the following components: *CP browser and CP details view*, *XD annotation dialog*, *XD selector*, *XD assistant*, *XD wizards*. In the following sections, these components are described in detail.

3.1 The CP browser and CP details view

The CP browser relies on a remote connection to registries of CPs. By default, XD tool allows to browse all CPs available at ontologydesignpatterns.org. The repository of CPs is visualized according to a semantic description based on the *codolight*[13] ontology. This approach makes the XD tool able to easily add new repositories to the browser, once it is provided with a codolight-based OWL description of them. This view allows the user to browse and select CPs as shown in Figure 5(a). The CP details view shows all available annotations of the selected CP. From the CP browser, a CP can be specialized and imported in the ontology under development.



(a) CP browser.


Ontology URI: http://www.example.org/myontologies/pattern.owl

This wizard helps you documenting your ontology.
Select a property from the list and you will see its description.

Select a property and click on "Add" if you want to add a value for it. Click on "Remove" if you want to remove an existing annotation.

"This annotation property is used for exemplifying possible requirements the content pattern provides a solution for. Requirements are expressed as natural language competency questions."

(b) XD annotation dialog.

Fig. 5. XD browser and annotation dialog.

3.2 XD annotation dialog

The XD annotation dialog, shown in Figure 5, supports multilingual annotation of ontology modules or CPs. This dialog supports a number of default vocabularies, and custom ones can be added. One of the vocabularies available by

[13] http://ontologydesignpatterns.org/cpont/codo/codolight.owl

default is the CP annotation schema[14], an OWL ontology particularly suited for annotating CPs.

3.3 XD selector: pattern selection support

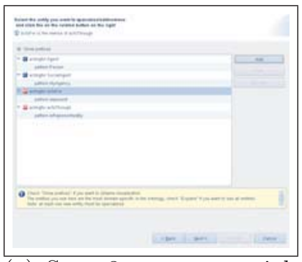
The XD selector provides the infrastructure for plugging into XD a component implementing a matching/searching algorithm that starting from a CQ gives as output a selection of candidate CPs. Currently only the APIs have been implemented, we are working at two proofs of concept components: an instance of Watson [10] specific for CPs, and a new version of OntoCase [2].

3.4 The XD assistant: support for design good practices

This component is able to provide the user with feedback related to possible mistakes and suggestions on good practices in a certain modeling situation. The XD assistant has a plugin-based architecture that make it easy to extend the help elements based on new emerging good/bad practices.
The XD assistant communicates two types of messages to the user: (i) **Warnings**: these messages are visualized when there is a strong suspect of wrong design. E.g. there is an anti-pattern in the module; (ii) **Suggestions**: these messages are visualized in order to suggest axioms currently missing in the module, that could improve the design.

3.5 XD wizards

XD also features a set of wizards. At the moment it includes a wizard for supporting CP specialization. The wizard can be accessed in the CP browser view,


(a) Step 3: create specialized entities.

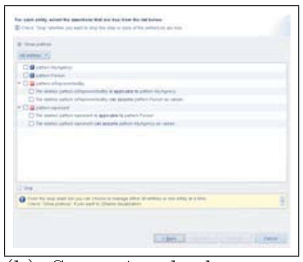

(b) Step 4: check statements that are true.

Fig. 6. Some steps of XD specialization wizard

by the command "specialized" included in the contextual menu that can be activated by right.clicking on the CP to be specialized (selected from the repository).

[14] http://ontologydesignpatterns.org/schemas/cpannotationschema.owl

The wizard guides the user through the following steps:

Step 1. The user selects the project and the target ontology (if any). Additionally, the user has to check one of three possible results that can be produced by the wizard:(i) *Create a new module/CP and import it in the target ontology*; (ii) *Create a new module/CP and store it in the indicated project*; (iii) *Merge the resulting entities in the target ontology.*

Step 2. All entity leaves are displayed to the user that is invited to select the entities to be specialized.

Step 3. For each selected entity the user creates the new specific one as shown in Figure 6(a).

Step 4. The wizard suggests possible axioms that can be added to the ontology by means of natural language statements. The users can automatically produce these additional axioms by selecting the assertions that they consider "true" in their context. This step of the wizard is depicted in Figure 6(b).

Step 5. Finally, an overview of natural language assertions corresponding to the formal axioms stated in the developed module is shown. This step has the aim of giving the users the possibility to review the result before to produce the module. They can always go back to a certain step in order to fix possible mistakes.

4 Conclusion and future work

In this paper, we have discussed **eXtreme Design (XD)**, an approach to ontology design based on the application, exploitation, and definition of ontology design patterns (ODPs). In more detail, we have presented two main contributions: a collaborative, incremental, and iterative method for pattern-based ontology design; and a first version of the XD tool, a NeOn Toolkit plugin that currently supports CP repository browsing and selection, a good practice assistant, and a wizard for CPs' specialization.
XD main principles are collaboration, integration, testing, and the extensive use of CPs. XD has been inspired by good practices typically adopted by the eXtreme Programming (XP) [11] software development methodology, such as pair programming, and customer involvement. While a rigorous evaluation of the whole methodology is still in our future plans, the effectiveness of CPs in ontology design has been rigorously evaluated in [5] where, however, XD has been used as reference development guidelines. Furthermore, initial questionnaires and informal discussions made emerge that the perception of the trainees with respect to the method is positive.
In order to ease the execution of the method some automatic support is needed. For example, matching CQs with GUCs is a complex task and automatic support

for filtering candidate CPs is necessary in order to exploit at best the evolving repositories of CPs. For addressing this type of needs, we have developed a NeOn Toolkit plugin, named "XD tool", that currently support CP browsing, annotation, and specialization. Furthermore, it provides APIs for pluggin-in components that implement matching algorithms. A lot of work is still ongoing, including the development of two components supporting matching/selection of CPs. As future work, we have planned to continue with XD tool development, and to conduct frequent user studies in order to evaluate and improve the methodology as well as to collect feedback and suggestions on the XD tool.

References

1. V. R. Basili, G. Caldiera, and D. Rombach. *Experience Factory*, pages 469–476. Wiley & Sons, 1994.
2. E. Blomqvist. Ontocase - automatic ontology enrichment based on ontology design patterns. In A. Bernstein and D. Karger, editors, *To appear in Proceedings of the 8th International Semantic Web Conference (ISWC 2009)*, 2009.
3. P. Clark and B. Porter. Building concept representations from reusable components. In *Proceedings of AAAI'97*, pages 369–376. AAAI press, 1997.
4. E. Daga, V. Presutti, and A. Salvati. http: //ontologydesignpatterns.org and evaluation wikiflow. In A. Gangemi, J. Keizer, V. Presutti, and H. Stoermer, editors, *SWAP*, volume 426 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.
5. Eva Blomqvist and Aldo Gangemi and Valentina Presutti. Experiments on Pattern-based Ontology Design. In *The Fifth International Conference on Knowledge Capture*, 2009.
6. A. Gangemi. Ontology Design Patterns for Semantic Web Content. In *Proceedings of the 4th International Semantic Web Conference*, pages 262–276. Springer, 2005.
7. V. P. A. Gangemi. Ontology design patterns. In R. S. S. Staab, editor, *Handbook of Ontologies*, International Handbooks on Information Systems. Springer, 2nd edition, 2009.
8. M. Gruninger and M. Fox. The role of competency questions in enterprise engineering, 1994.
9. S. Pinto, S. Staab, and C. Tempich. DILIGENT: Towards a fine-grained methodology for Distributed Loosely-controllled and evolvInG Engineering of oNTologies. In *Proceedings of ECAI-2004*, 2004.
10. M. Sabou, C. Baldassarre, L. Gridinoc, S. Angeletou, E. Motta, M. d'Aquin, and M. Dzbor. Watson: A gateway for the semantic web. In *ESWC 2007 poster session*, June 2007-06.
11. J. Shore and S. Warden. *The Art of Agile Development.* O'Reilly, Sebastopol, CA, USA, 2007.
12. C. G. von Wangenheim, K.-D. Althoff, and R. M. Barcia. Goal-oriented and similarity-based retrieval of software engineering experienceware. In G. Ruhe and F. Bomarius, editors, *SEKE*, volume 1756 of *Lecture Notes in Computer Science*, pages 118–141. Springer, 1999.
13. D. Vrandečić and A. Gangemi. Unit tests for ontologies. In M. Jarrar, C. Ostyn, W. Ceusters, and A. Persidis, editors, *Proceedings of the 1st International Workshop on Ontology content and evaluation in Enterprise*, LNCS, Montpellier, France, October 2006. Springer.

Part 2: Patterns

Foreword

These proceedings contain descriptions of patterns accepted at the first Workshop on Ontology Patterns held during the 11th International Semantic Web Conference near Washington DC in October 2009. Besides regular papers we proposed authors to submit ontology patterns. The ontology design patterns portal provides a natural way to describe and share ontology design patterns. The portal reviewing facility provides evaluation means ensuring quality control over the ontology design patterns described on the portal. The patterns accepted at the workshop and presented in these proceedings can thus also be found on the portal.

We distinguish two types of patterns accepted for publication: patterns accepted for discussion during the workshop, and patterns accepted for a poster presentation session also to be held during the workshop. It would for sure have been interesting to discuss all the patterns published in this proceedings as they all deserve attention and questioning. However time constraints made us select the three patterns we considered likely to raise the most lively discussions. We tried to consider for discussion patterns which, by their type (following the ODP typology: http://ontologydesignpatterns.org/wiki/OPTypes), and by the kind of problem they try to solve, are representative of the patterns accepted for publication at the workshop.

The submissions can be split in three clearly distinct categories. The first category of patterns is constituted of re-engineering patters. These patterns propose methods, or algorithms, to transform a structured or semi-structured data model into an ontology. A second categorize is constituted of evolution and inconsistency resolution patterns. These patterns are particularly useful when a revision of an ontology introduce inconsistency, either in the ontology, or in the knowledge base. Evolution patterns ensure a consistent revision of the ontology, while inconsistency resolution patterns repair an introduced inconsistency. A third category of pattern is constituted of anti-patterns. These patterns actually model bad modeling in ontologies. Anti-patterns are particularly useful for improving the quality of existing ontologies. They can arise because of no usage of a proper design methodology during the ontology construction process. These three categories of patterns are reflected both in poster presentations and in the patterns selected for discussion during the workshop.

Patterns accepted for poster presentation:

- ConceptTerms - Pierre-Yves Vandenbussche and Jean Charlet
- Negative Property Assertion Pattern (NPAs) - Olaf Noppens
- Concept Partition Pattern - Olaf Noppens
- Pattern for Re-engineering a Classification Scheme, Which Follows the Path Enumeration Data Model, to a Taxonomy - Boris Villazon-Terrazas, Mari Carmen Suarez-Figueroa, and Asuncion Gomez-Perez
- Pattern for Re-engineering a Classification Scheme, Which Follows the Adjacency List Data Model, to a Taxonomy - Boris Villazon-Terrazas, Mari Carmen Suarez-Figueroa, and Asuncion Gomez-Perez

Patterns accepted for discussion during the workshop:

- Define Hybrid Class Resolving Disjointness Due to Subsumption - Rim Djedidi and Marie-Aude Aufaure
- OnlynessIsLoneliness (OIL) - Oscar Corcho and Catherine Roussey
- Pattern for Re-engineering a Term-based Thesaurus, Which Follows the Record-based Model, to a Lightweight Ontology - Boris Villazon-Terrazas, Mari Carmen Suarez-Figueroa, and Asuncion Gomez-Perez

We would like to thank the authors who both submitted the patterns descriptions and entered patterns on the ODP portal.

We would also like to thank reviewers who take time to understand and discuss the submitted patterns. Judging the quality of a pattern is not an easy task. We therefore tried to focus the evaluation on the clarity of description of the patterns, rather than on their potential usefulness, which will probably be proven together with time and experience of usage.

The patterns chairs,

Eva Blomqvist and François Scharffe
October 2009

Define Hybrid Class Resolving Disjointness Due to Subsumption

http://ontologydesignpatterns.org/wiki/Submissions:Define_Hybrid_Class_Resolving_Disjointness_due_to_Subsumption

Rim Djedidi[1] and Marie-Aude Aufaure[2]

[1] Computer Science Department, Supélec Campus de Gif
Plateau du Moulon – 3, rue Joliot Curie – 91192 Gif sur Yvette Cedex, France
rim.djedidi@supelec.fr
[2] MAS Laboratory, SAP Business Object Chair –Centrale Paris
Grande Voie des Vignes, F-92295 Châtenay-Malabry Cedex, France
marie-aude.aufaure@ecp.fr

1 Introduction

The pattern "*Define Hybrid Class Resolving Disjointness due to Subsumption*" is proposed as a Logical Ontology Design Pattern (Logical OP) solving a problem of disjointness inconsistency caused by a subsumption relation. Further away from solving design problems where the primitives of the representation language do not directly support certain logical constructs, this pattern helps resolving a logical inconsistency triggered by a situation of disjoint classes subsuming a common sub-class. The solution presented by the pattern resolves the inconsistency while preserving existing knowledge, i.e. a resolution alternative avoiding axiom deletion.

2 Pattern

In this section, we specify the problem that the pattern deals with and the requirements covered by it; we detail the description of the solution given by this pattern and the consequences of its application; and we illustrate the pattern by an example problem and its corresponding solution.

2.1 Problem

The pattern "*Define Hybrid Class Resolving Disjointness due to Subsumption*" is proposed to solve a problem of disjointness inconsistency caused by a subsumption relation. When we need to define – for some modeling issues related to domain of interest – a class as a sub-class of two disjoint classes, a disjointness inconsistency is caused.

The problem can be illustrated by the following scenario: let's consider a class *Sub_Class* defined as a sub-class of a class *Disjoint_Class 2*; and a class

Disjoint_Class 1 disjoint with the *Disjoint_Class 2* (Fig. 1). If we need to add a sub-class relation between the *Sub_Class* and the *Disjoint_Class 1*, this generates a disjointness inconsistency:

– If the extension of the *Sub_Class* contains individuals instantiating this sub-class, the logical inconsistency will be extended to the knowledge base;
– If the *Sub_Class* is not instantiated to individuals, it will be diagnosed as an unsatisfiable class.

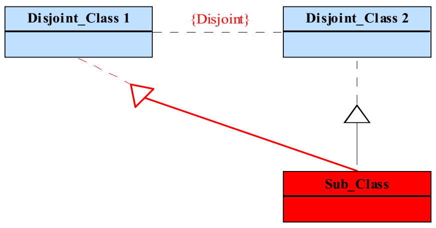


Fig. 1. Graphical illustration of the problem the pattern deals with.

To solve this inconsistency, one can think about deleting the disjointness axiom. However, this can alter the semantics expressed in the ontology, and negatively affect consistency checking and automatic evaluation of existing individuals as explained in [1].

This pattern tackles the questions of how to resolve the inconsistency caused by such kind of subsumption while preserving existing knowledge.

Intent The purpose of this pattern is to support the semantics of a subsumption defined under two disjoint classes and resolve the resulting inconsistency.

Covered Requirements The pattern solves a problem of disjointness inconsistency caused by a subsumption relation without deleting the disjointness axiom so that existing knowledge can be preserved.

2.2 Solution

The pattern resolves a disjointness inconsistency –due to a subsumption–by defining a *Hybrid Class* based on the definition of disjoint classes implicated in the inconsistency; and redistributing correctly sub-class relations between the sub-class, the hybrid class, and the most specific common super-class of the disjoint classes implicated. The definition of the *Hybrid Class* is the union (OR) of the definitions of the disjoint classes.

The application of the solution can be described by the following process (Fig. 2):

1. The pattern defines a *Hybrid Class* as a union of the definitions of the disjoint classes implicated in the inconsistency to be resolved;
2. The pattern defines a subsumption between the most specific common super-class of the disjoint classes implicated in the inconsistency, and the *Hybrid Class* created;

3. The pattern defines a subsumption between the *Hybrid Class* and the sub-class involved in the inconsistency.

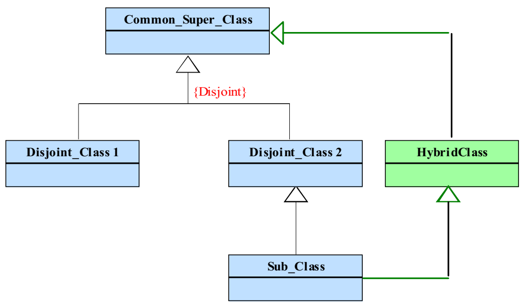


Fig. 2. Graphical representation of the proposed pattern.

Consequences The application of the pattern resolves the disjointness inconsistency (even if the involved sub-class is instantiated by individuals) and preserves existing knowledge. As a Logical OP, this pattern is independent from a specific domain of interest. However, it depends on the expressivity of the logical formalism used for the representation of the ontology. Therefore, the language of the targeted ontology should allow expressing class union.

2.3 Example

To explain pattern application, we present in this section, an example problem and its corresponding solution according to the pattern.

Example Problem Let's consider the OWL ontology O defined by the following axioms:

{Animal ⊑ Fauna-Flora, Plant ⊑ Fauna-Flora, Carnivorous-Plant ⊑ Plant, Plant ⊑ ¬Animal}

If we apply a change to the ontology defining *Carnivorous-Plant* class as a sub-class of the class *Animal*, we cause a disjointness inconsistency. The proposed pattern resolves this kind of inconsistency.

Example Solution The application of the pattern to resolve the example above is performed as follow:

1. The pattern defines a class *Animal_Plant* as a union of the definitions of the disjoint classes *Animal* and *Plant*;
2. The pattern defines a subsumption between the most specific common super-class of the disjoint classes *Fauna-Flora* and the hybrid class created *Animal_Plant*;

3. The pattern defines a subsumption between the defined hybrid class *Animal_Plant* and the sub-class *Carnivorous-Plant* involved in the inconsistency.

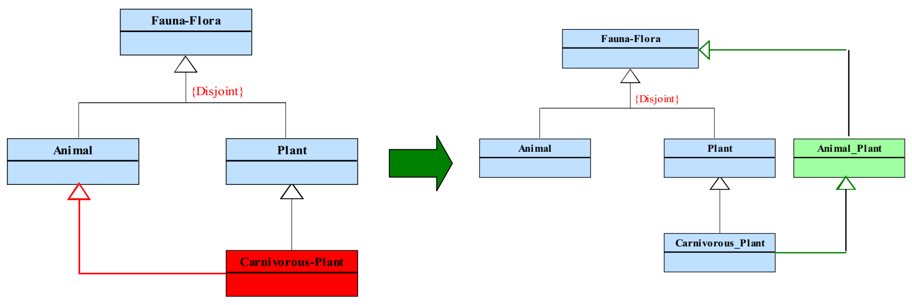


Fig. 3. Illustration of an example of problem and its corresponding solution.

3 Pattern Usage

The proposed – Logical OP – pattern "*Define Hybrid Class Resolving Disjointness due to Subsumption*" is applied as an *Alternative Resolution Pattern* in an ontology evolution approach ***ONTO-EVO^{A}L***, guided by *Change Management Patterns* (CMP) [2]. CMP patterns drive and control the change management process at three key phases: change specification, change analysis, and change resolution, by modeling three categories of patterns: *Change Patterns* classifying types of changes, *Inconsistency Patterns* classifying types of logical inconsistencies, and *Alternative Patterns* classifying types of inconsistency resolution alternatives.

4 Summary and Future Work

The purpose of this pattern is to support the semantics of a subsumption defined under two disjoint classes and resolve the resulting inconsistency without removing existing knowledge. This pattern can be extended and adapted to resolve disjointness inconsistency due to instantiation.

References

1. Völker, J., Vrandecic, D., Sure, Y., Hotho, A.: Learning Disjointness. In F., Enrico, K., Michael, May. Wolfgang (Eds.). Proceedings of the 4th European Semantic Web Conference, ESWC 2007. LNCS: Vol. 4519 pp: 175-189. (2007)

2. Djedidi, R., Aufaure, M-A.: Ontology Change Management. In: A. Paschke, H. Weigand, W. Behrendt, K. Tochtermann, T. Pellegrini (Eds.), I-Semantics 2009, Proceedings of I-KNOW '09 and I-SEMANTICS '09, ISBN 978-3-85125-060-2, pp. 611--621, Verlag der Technischen Universitt Graz. (2009).

OnlynessIsLoneliness (OIL)

http://ontologydesignpatterns.org/wiki/Submissions:OnlynessIsLoneliness_(OIL)

Oscar Corcho[1] and Catherine Roussey[23]

[1] Ontology Engineering Group, Departamento de Inteligencia Artificial, Universidad Politécnica de Madrid, Spain
ocorcho@fi.upm.es,
WWW home page:
http://www.dia.fi.upm.es/index.php?page=oscar-corcho&hl=en_US
[2] Cemagref, 24 Av. des Landais, BP 50085, 63172 Aubiére, France
catherine.roussey@cemagref.fr,
WWW home page: http://www.cemagref.fr/
[3] Université de Lyon, CNRS, Université Lyon 1, LIRIS UMR5205, Villeurbanne, France
catherine.roussey@liris.cnrs.fr,
WWW home page: http://liris.cnrs.fr/membres?idn=croussey

1 Introduction

Our work is based on the debugging process of real ontologies that have been developed by domain experts, who are not necessarily too familiar with DL, and hence can misuse DL constructors and misunderstand the semantics of some OWL expressions, leading to unwanted unsatisfiable classes. Our patterns were first found during the debugging process of a medium-sized OWL ontology (165 classes) developed by a domain expert in the area of hydrology [9]. The first version of this ontology had a total of 114 unsatisfiable classes. The information provided by the debugging systems used ([3], [5]) on (root) unsatisfiable classes was not easily understandable by domain experts to find the reasons for their unsatisfiability. And in several occasions during the debugging process the generation of justifications for unsatisfiability took several hours, what made these tools hard to use, confirming the results described in [8]. Using this debugging process and several other real ontologies debugging one, we found out that in several occasions domain experts were just changing axioms from the original ontology in a somehow random manner, even changing the intended meaning of the definitions instead of correcting errors in their formalisations.

We have identified a set of patterns that are commonly used by domain experts in their DL formalisations and OWL implementations, and that normally result in unsatisfiable classes or modelling errors ([1], [7]). Thus they are antipatterns. [6] define antipatterns as patterns that appear obvious but are ineffective or far from optimal in practice, representing worst practice about how to structure and build software. We also have made an effort to identify common alternatives for providing solutions to them, so that they can be used by domain experts to debug their ontologies.

All these antipatterns come from a misuse and misunderstanding of DL expressions by ontology developers. Thus they are all Logical AntiPatterns (LAP): they are independent from a specific domain of interest, but dependent on the expressivity of the logical formalism used for the representation.

2 Pattern

2.1 Problem

The ontology developer created a universal restriction to say that C_1 instances can only be linked with property R to C_2 instances. Next, a new universal restriction is added saying that C_1 instances can only be linked with R to C_3 instances, with C_2 and C_3 disjoint. Figure 1 illustrates this problem: grey squares represent instances of $C_2 \sqcap C_3$ that cannot exist. In general, this is because the ontology developer forgot the previous axiom in the same class or in any of the parent classes.

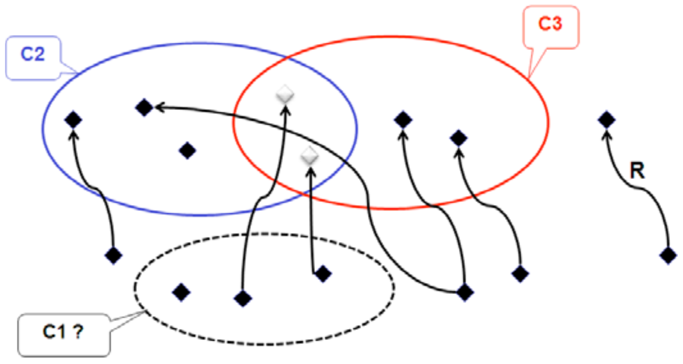


Fig. 1. A graphical representation of OIL antipattern.

$C_1 \sqsubseteq \forall R.(C_2); C_1 \sqsubseteq \forall R.(C_3); Disj(C_2, C_3);$ [4]

Notice that to be detectable, R property must have at least a value, normally specified as a (minimum) cardinality restriction for that class, or with existential restrictions.

Covers Requirements When this antipattern appears during the debugging process, you have to first explain to the domain expert the meaning of this formalisation using a schema like the one of the Figure 1. Then you could ask

[4] This does not mean that the ontology developer has explicitly expressed that C_2 and C_3 are disjoint, but that these two concepts are determined as disjoint from each other by a reasoner. We use this notation as a shorthand for $C_2 \sqcap C_3 \sqsubseteq \bot$.

him some questions to find out where is the problem. For example, you could ask:

- Should C_1 be linked with the R property to C_2?
- Should C_1 be linked with the R property to C_3?
- Does C_1 have to be linked only to C_2 with the R property?
- Does C_1 have to be linked only to C_3 with the R property?
- Are you sure that C_2 and C_3 are disjoint?

2.2 Solution

If it makes sense, we propose the domain expert to transform the two universal restrictions into only one that refers to the logical disjunction of C_2 and C_3. Another alternative solution, which is used by most part of automatic debugging tool is to remove one of the axioms.

~~$C_1 \sqsubseteq \forall R.C_2; C_1 \sqsubseteq \forall R.C_3$~~$; Disj(C_2, C_3); \Rightarrow C_1 \sqsubseteq \forall R.(C_2 \sqcup C_3);$

2.3 Example

The following section describes two definitions from HydrOntology where this antipattern can be found and their English translations. Notice that in each example, the antipattern corresponds to a part of the class definition.

Example Problem about Transitional Water

$Aguas\_de\_Transición \sqsubseteq \forall está\_próxima.Aguas\_Marinas \sqcap \forall está\_próxima.Desembocadura \sqcap = 1 está\_próxima.\top;$

$Transitional\_Water \sqsubseteq \forall is\_nearby.Sea\_Water \sqcap \forall is\_nearby.River\_Mouth \sqcap = 1 is\_nearby.\top;$

Example Solution about Transitional Water

$Aguas\_de\_Transición \sqsubseteq \forall está\_próxima.(Aguas\_Marinas \sqcup Desembocadura) \sqcap = 1 está\_próxima.\top;$

$Transitional\_Water \sqsubseteq \forall is\_nearby.(Sea\_Water \sqcup River\_Mouth) \sqcap = 1 is\_nearby.\top$

Example Problem about Wet Zone

$Zona\_Humeda \sqsubseteq \forall Humedal \sqcap \forall es\_inundada.Aguas\_Marinas \sqcap \forall es\_inundada.Aguas\_Superficiales \sqcap \geq 1 es\_inundada.\top;$

$Wet\_Zone \sqsubseteq \forall Wetlands \sqcap \forall are\_inundated.Sea\_Water \sqcap \forall are\_inundated.Surface\_Water \sqcap \geq 1 are\_inundated.\top;$

Example Solution about Wet Zone

$Zona\_Humeda \sqsubseteq \forall Humedal \sqcap \forall es\_inundada.(Aguas\_Marinas \sqcup Aguas\_Superficiales) \sqcap \geq 1 es\_inundada.\top;$

$Wet\_Zone \sqsubseteq \forall Wetlands \sqcap \forall are\_inundated.(Sea\_Water \sqcup Surface\_Water) \sqcap \geq 1 are\_inundated.\top;$

2.4 Related Resources and Pattern Usage

All the information related to the debugging of the Hydrontology ontology can be found in urlhttp://www.dia.fi.upm.es/ ocorcho/OWLDebugging/. The debugging strategy using this antipattern is described in [2]. Other antipatterns found during the debugging task are defined in [1] and [7]

3 Summary and Future Work

This antipattern can be found in ontologies and may cause inconsistency problems. We provide a solution to it, so that it can be used by domain experts to debug their ontologies. In the future, we aim at implementing additional tools to help in the identification of antipatterns in well-known inconsistent ontologies (e.g., TAMBIS). For the time being we have started applying the OPPL language [4] for this task, with promising results.

References

1. Corcho O., Roussey C., Vilches Blazquez L.M.: Catalogue of Anti-Patterns for formal Ontology debugging. In Proceedings of Construction d'ontologies : vers un guide des bonnes pratiques, AFIA 2009, Hammamet, Tunisie. (2009).
2. Corcho O., Roussey C., Vilches Blazquez L.M.: Pattern-based OWL Ontology Debugging Guidelines. In Proceedings of 1st Workshop on Ontology Patterns (WOP2009), Washington DC, USA. (2009).
3. Horridge M, Parsia B, Sattler U.: Laconic and Precise Justifications in OWL. In Proceedings of the 7th International Semantic Web Conference (ISWC), Karlsruhe, Germany; LNCS 5318: 323-338. (2008).
4. Iannone L, Rector A, Stevens R.: Embedding Knowledge Patterns into OWL. In proceedings of the 6th European Semantic Web Conference (ESWC2009), Crete, Greece. The Semantic Web: Research and Applications (2009), pp. 218-232
5. Kalyanpur A, Parsia B, Sirin E, Cuenca-Grau B.: Repairing Unsatisfiable Classes in OWL Ontologies. In Proceedings of the 3rd European Semantic Web Conference (ESWC), Budva, Montenegro; LNCS 4011: 170-184 (2006)
6. Koenig A.: Patterns and Antipatterns. Journal of Object-Oriented Programming 8(1):46-48. (1995)
7. Roussey C., Corcho O., Vilches Blazquez L.M.: A Catalogue of OWL Ontology AntiPatterns. In Proceedings of the Fifth International Conference on Knowledge Capture KCAP 2009, Yolanda Gil, Natasha Noy ed. Redondo Beach, California, USA. ISBN 978-1-60558-658-8. pp. 205-206 (2009)
8. Stuckenschmidt H.: Debugging OWL Ontologies - a Reality Check. In Proceedings of the 6th International Workshop on Evaluation of Ontology-based Tools and the Semantic Web Service Challenge (EON-SWSC-2008), Tenerife, Spain. (2008).
9. Vilches-Blázquez LM, Bernabé-Poveda MA, Suárez-Figueroa MC, Gómez-Pérez A, Rodrguez-Pascual AF: Towntology & hydrOntology: Relationship between Urban and Hydrographic Features in the Geographic Information Domain. In Ontologies for Urban Development. Studies in Computational Intelligence, vol. 61, Springer: 73-84. (2007)

Pattern for Re-engineering a Term-based Thesaurus, Which Follows the Record-based Model, to a Lightweight Ontology

http://ontologydesignpatterns.org/wiki/Submissions:Term-based_-_record-based_model_-_thesaurus_to_lightweight_ontology

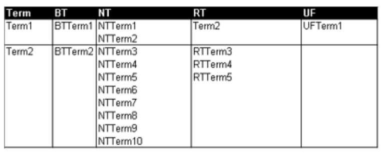
Boris Villazón-Terrazas[1], Mari Carmen Suárez-Figueroa[1], and Asunción Gómez-Pérez[1]

Ontology Engineering Group, Departamento de Inteligencia Artificial, Facultad de Informática, Universidad Politécnica de Madrid, Spain
`{bvillazon,mcsuarez,asun}@fi.upm.es`,
WWW home page: `http://www.oeg-upm.net/`

1 Introduction

This pattern for re-engineering non-ontological resources (PR-NOR) fits in the Schema Re-engineering Category proposed by [3]. The pattern defines a procedure that transforms the term-based thesaurus components into ontology representational primitives. This pattern comes from the experience of ontology engineers in developing ontologies using thesauri in several projects (SEEMP[1], NeOn[2], and Knowledge Web[3]). The pattern is included in a pool of patterns, which is a key element of our method for re-engineering non-ontological resources into ontologies [2]. The patterns generate the ontologies at a conceptualization level, independent of the ontology implementation language.

2 Pattern

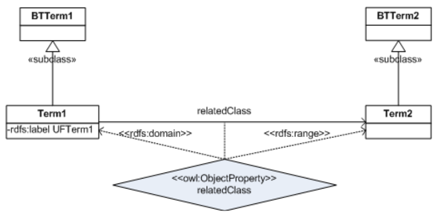
Problem	
Re-engineering a term-based thesaurus, which follows the record-based model, to design a lightweight ontology.	
Non-Ontological Resource	
A non-ontological resource holds a term-based thesaurus which follows the record-based model. A thesaurus represents the knowledge of a domain with a collection of terms and a limited set of relations between them. The record-based data model [4] is a denormalized structure, uses a record for every term with the information about the term, such as synonyms, broader, narrower and related terms.	
Applicability	
The semantics of the relation between narrower and broader terms are *subClassOf*.	

Term	BT	NT	RT	UF
Term1	BTTerm1	NTTerm1 NTTerm2	Term2	UFTerm1
Term2	BTTerm2	NTTerm3 NTTerm4 NTTerm5 NTTerm6 NTTerm7 NTTerm8 NTTerm9 NTTerm10	RTTerm3 RTTerm4 RTTerm5	

[1] http://www.seemp.org
[2] http://www.neon-project.org
[3] http://knowledgeweb.semanticweb.org

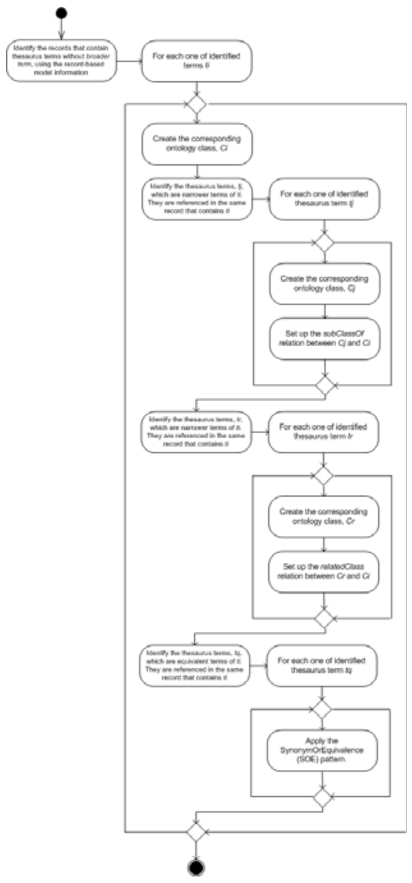
Ontology Generated

The ontology generated will be based on the lightweight ontology architectural pattern (AP-LW-01) [5].
Each thesaurus term is mapped to a class. A *subClassOf* relation is defined between the new classes for the BT/NT relation. A *relatedClass* relation is defined between the new classes for the RT relation. For the UF/USE relations the SynonymOrEquivalence (SOE) pattern [1] is applied.



Process - Solution

1. Identify the records that contain thesaurus terms without a *broader term.*
2. For each one of the above identified thesaurus terms t_i:
   - 2.1. Create the corresponding ontology class, C_i class, if it is not created yet.
   - 2.2. Identify the thesaurus term, t_j, which are narrower terms of t_i. They are referenced in the same record that contains t_i.
   - 2.3. For each one of the above identified thesaurus term t_j:
     - 2.3.1. Create the corresponding ontology class, C_j class, if it is not created yet.
     - 2.3.2. Set up the *subClassOf* relation between C_j and C_i
     - 2.3.3. Repeat from step 2.2 for c_j as a new t_i
   - 2.4. Identify the thesaurus term, t_r, which are related terms of t_i. They are referenced in the same record that contains t_i.
   - 2.5. For each one of the above identified thesaurus term t_r:
     - 2.5.1. Create the corresponding ontology class, C_r class, if it is not created yet.
     - 2.5.2. Set up the *relatedClass* relation between C_r and C_i
     - 2.5.3. Repeat from step 2.4 for t_r as a new t_i
   - 2.6. Identify the thesaurus term, t_q, which are equivalent terms of t_i. They are referenced in the same record that contains t_i.
   - 2.7. For each one of the above identified thesaurus term t_q:
     - 2.7.1. Apply the SynonymOrEquivalence (SOE) pattern.

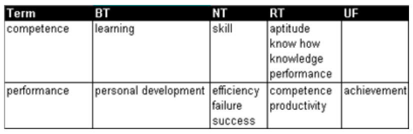

Identify the records that contain thesaurus terms without broader term, using the record-based model information
For each one of identified terms ti
Create the corresponding ontology class, Ci
Identify the thesaurus terms, tj, which are narrower terms of ti. They are referenced in the same record that contains ti
For each one of identified thesaurus term tj
Create the corresponding ontology class, Cj
Set up the subClassOf relation between Cj and Ci
Identify the thesaurus terms, tr, which are narrower terms of ti. They are referenced in the same record that contains ti
For each one of identified thesaurus term tr
Create the corresponding ontology class, Cr
Set up the relatedClass relation between Cr and Ci
Identify the thesaurus terms, tq, which are equivalent terms of ti. They are referenced in the same record that contains ti
For each one of identified thesaurus term tq
Apply the SynonymOrEquivalence (SOE) pattern.

Example

Suppose that someone wants to build a lightweight ontology based on the European Training Thesaurus (ETT), which is a term-based thesaurus and it follows the record-based model.

Non-Ontological Resource

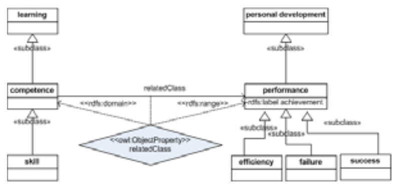
The European Training Thesaurus (ETT) constitutes the controlled vocabulary of reference in the field of vocational education and training (VET) in Europe. The relation semantics between the sub-ordinate and the super-ordinate concepts is *subClassOf*. This classification scheme is available at http://libserver.cedefop.europa.eu/ett/en/



Term	BT	NT	RT	UF
competence	learning	skill	aptitude know how knowledge performance	
performance	personal development	efficiency failure success	competence productivity	achievement

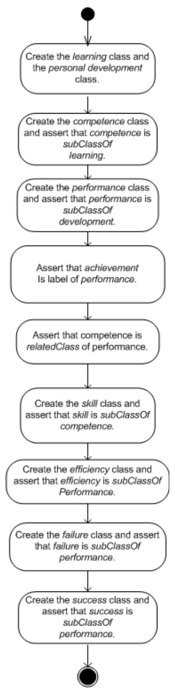
Ontology Generated

The ontology generated will be based on the lightweight ontology architectural pattern (AP-LW-01) [5].
Each thesaurus term is mapped to a class. A *subClassOf* relation is defined between the new classes for the BT/NT relation. A *relatedClass* relation is defined between the new classes for the RT relation. For the UF/USE relations the SynonymOrEquivalence (SOE) pattern [1] is applied.



Process - Solution

1. Create the `learning` class and the `personal development` class.
2. Create the `competence` class and assert that `competence` is *subClassOf* `learning`.
3. Create the `performance` class and assert that `performance` is *subClassOf* `development`.
4. Assert that `achievement` is label of the `performance` class.
5. Assert that `competence` is *relatedClass* of `performance`.
6. Create the `skill` class and assert that `skill` is *subClassOf* `competence`.
   6.1. Create the `efficiency` class and assert that `efficiency` is *subClassOf* `performance`.
   6.2. Create the `failure` class and assert that `failure` is *subClassOf* `performance`.
   6.3. Create the `success` class and assert that `success` is *subClassOf* `performance`.


Create the *learning* class and the *personal development* class.
Create the *competence* class and assert that *competence* is *subClassOf learning.*
Create the *performance* class and assert that *performance* is *subClassOf development.*
Assert that *achievement* Is label of *performance.*
Assert that competence is *relatedClass* of performance.
Create the *skill* class and assert that *skill* is *subClassOf competence.*
Create the *efficiency* class and assert that *efficiency* is *subClassOf Performance.*
Create the *failure* class and assert that *failure* is *subClassOf performance.*
Create the *success* class and assert that *success* is *subClassOf performance.*

Related Resources

This pattern is related to the architectural pattern AP-LW-01 [5] for modelling a lightweight ontology.

3 Pattern Usage

This pattern is being applied to re-engineer the European Training Thesaurus (ETT)[4] into a Education Ontology[5], within the context of the SEEMP project. It contains over 2500 terms (1550 are descriptors, and 950 non descriptors). This term-based thesaurus is modelled following the record-based data model.

4 Summary and Future Work

We have presented a pattern for transforming a term-based thesaurus, which is modelled following a record-based data model, into a lightweight ontology. The pattern is included in a pool of patterns, which is a key element of our method for re-engineering non-ontological resources into ontologies [2].

We plan to develop software libraries within a framework that implement the transformation process suggested by the pattern. Moreover, we will include external resources to improve the quality of the resultant ontologies. Finally, we need to calculate how much effort do we save re-engineering classification schemes using patterns compared with re-engineering classification schemes without them.

Acknowledgments. This work has been partially supported by the European Comission projects NeOn(FP6-027595) and SEEMP(FP6-027347), as well as by an R+D grant from the UPM.

References

1. C. Roussey and O. Corcho. SynonymOrEquivalence (SOE) Pattern. *http://ontologydesignpatterns.org*, 2009.
2. A. García, A. Gómez-Pérez, M. C. Suárez-Figueroa, and B. Villazón-Terrazas. A Pattern Based Approach for Re-engineering Non-Ontological Resources into Ontologies. In *Proceedings of the 3rd Asian Semantic Web Conference (ASWC2008).* Springer-Verlag, 2008.
3. V. Presutti, A. Gangemi, S. David, G. Aguado de Cea, M. C. Surez-Figueroa, E. Montiel-Ponsoda, and M. Poveda. NeOn Deliverable D2.5.1. A Library of Ontology Design Patterns: reusable solutions for collaborative design of networked ontologies. In *NeOn Project. http://www.neon-project.org*, 2008.
4. D. Soergel. Data models for an integrated thesaurus database. *Comatibility and Integration of Order Systems*, 24(3):47–57, 1995.
5. M. C. Suárez-Figueroa, S. Brockmans, A. Gangemi, A. Gómez-Pérez, J. Lehmann, H. Lewen, V. Presutti, and M. Sabou. Neon modelling components. Technical report, NeOn project deliverable D5.1.1, 2007.

[4] http://libserver.cedefop.europa.eu/ett/en/
[5] The ontology will be available at http://droz.dia.fi.upm.es/hrmontology/

Pattern for Re-engineering a Classification Scheme, Which Follows the Path Enumeration Data Model, to a Taxonomy

http://ontologydesignpatterns.org/wiki/Submissions:Classification_scheme_-_path_enumeration_model_-_to_Taxonomy

Boris Villazón-Terrazas[1], Mari Carmen Suárez-Figueroa[1], and Asunción Gómez-Pérez[1]

Ontology Engineering Group, Departamento de Inteligencia Artificial, Facultad de Informática, Universidad Politécnica de Madrid, Spain
{bvillazon,mcsuarez,asun}@fi.upm.es,
WWW home page: http://www.oeg-upm.net/

1 Introduction

This pattern for re-engineering non-ontological resources (PR-NOR) fits in the Schema Re-engineering Category proposed by [3]. The pattern defines a procedure that transforms the classification scheme components into ontology representational primitives. This pattern comes from the experience of ontology engineers in developing ontologies using classification schemes in several projects (SEEMP[1], NeOn[2], and Knowledge Web[3]). The pattern is included in a pool of patterns, which is a key element of our method for re-engineering non-ontological resources into ontologies [2]. The patterns generate the ontologies at a conceptualization level, independent of the ontology implementation language.

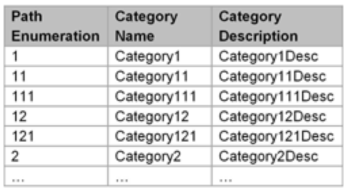
2 Pattern

Problem

Re-engineering a classification scheme, which follows the path enumeration model, to design a taxonomy.

Non-Ontological Resource

A non-ontological resource holds a classification scheme which follows the path enumeration model. A classification scheme is a rooted tree of concepts, in which each concept groups entities by some particular degree of similarity.
The semantics of the hierarchical relation between parents and children concepts may vary depending of the context. The path enumeration data model [1] for classification schemes take advantage of that there is one and only one path from the root to every item in the classification. The path enumeration model stores that path as string by concatenating either the edges or the keys of the classification scheme items in the path.



Path Enumeration	Category Name	Category Description
1	Category1	Category1Desc
11	Category11	Category11Desc
111	Category111	Category111Desc
12	Category12	Category12Desc
121	Category121	Category121Desc
2	Category2	Category2Desc
...	...	...

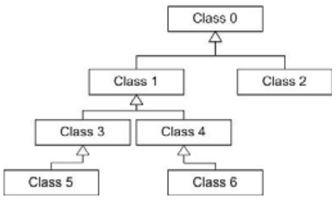
[1] http://www.seemp.org
[2] http://www.neon-project.org
[3] http://knowledgeweb.semanticweb.org

Applicability

The semantics of the relation between parent and children items are *subClassOf*.
There is not multi-inheritance nor cyclic relations.

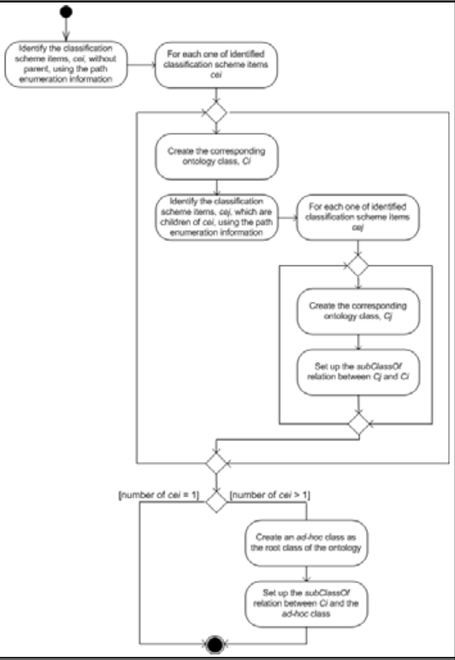
Ontology Generated

The ontology generated will be based on the taxonomy architectural pattern (AP-TX-01) [4].
Each category in the classification scheme is mapped to a class, and the semantics of the relationship between children and parent categories are mapped to *subClassOf* relations.



Process - Solution

1. Identify the classification scheme items whose their path enumeration values have the shortest length, i.e. classification scheme items without parents.
2. For each one of the above identified classification scheme items ce_i:
   - 2.1. Create the corresponding ontology class, C_i class.
   - 2.2. Identify the classification scheme items, ce_j, which are children of ce_i, by using the path enumeration values.
   - 2.3. For each one of the above identified classification scheme items ce_j:
     - 2.3.1. Create the corresponding ontology class, C_j class.
     - 2.3.2. Set up the *subClassOf* relation between C_j and C_i.
     - 2.3.3. Repeat from step 2.2 for ce_j as a new ce_i.
3. If there are more than one classification scheme items without parent ce_i
   - 3.1. Create an *ad-hoc* class as the root class of the ontology.
   - 3.2. Set up the *subClassOf* relation between C_i class and the root class.

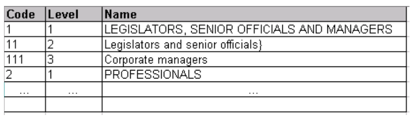

Identify the classification scheme items, cei, without parent, using the path enumeration information
For each one of identified classification scheme items cei
Create the corresponding ontology class, Ci
Identify the classification scheme items, cej, which are children of cei, using the path enumeration information
For each one of identified classification scheme items cej
Create the corresponding ontology class, Cj
Set up the subClassOf relation between Cj and Ci
[number of cei = 1]
[number of cei > 1]
Create an ad-hoc class as the root class of the ontology
Set up the subClassOf relation between Ci and the ad-hoc class

Example

Suppose that someone wants to build an ontology based on the International Standard Classification of Occupations (for European Union purposes) ISCO-88 (COM). This classification scheme follows the path enumeration data model.

Non-Ontological Resource

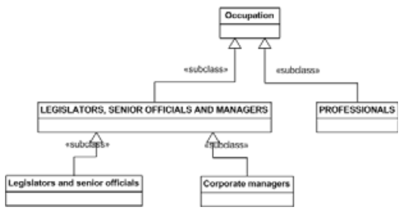
The International Standard Classification of Occupations (for European Union purposes), 1988 version: ISCO-88 (COM) published by Eurostat is modelled with the path enumeration data model. This classification scheme is available at http://ec.europa.eu/eurostat/ramon/



Code	Level	Name
1	1	LEGISLATORS, SENIOR OFFICIALS AND MANAGERS
11	2	Legislators and senior officials}
111	3	Corporate managers
2	1	PROFESSIONALS
...	...	...

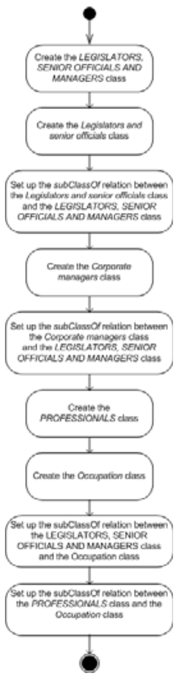
Ontology Generated

The ontology generated will be based on the taxonomy architectural pattern (AP-TX-01) [4]. Each category in the classification scheme is mapped to a class, and the semantics of the relationship between children and parent categories are mapped to *subClassOf* relations.



Process - Solution

1. Create the `LEGISLATORS, SENIOR OFFICIALS AND MANAGERS` class.
   1.1. Create the `Legislators and senior officials` class, and set up the *subClassOf* relation between the `Legislators and senior officials` class and the `LEGISLATORS, SENIOR OFFICIALS AND MANAGERS` class.
   1.2. Create the `Corporate managers` class, and set up the *subClassOf* relation between the `Corporate managers` class and the `LEGISLATORS, SENIOR OFFICIALS AND MANAGERS` class.
2. Create the `PROFESSIONALS` class.
3. Create the `Occupation` class.
4. Set up the *subClassOf* relation between the `LEGISLATORS, SENIOR OFFICIALS AND MANAGERS` class and the `Occupation` class.
5. Set up the *subClassOf* relation between the `PROFESSIONALS` class and the `Occupation` class.


Create the LEGISLATORS, SENIOR OFFICIALS AND MANAGERS class
Create the Legislators and senior officials class
Set up the subClassOf relation between the Legislators and senior officials class and the LEGISLATORS, SENIOR OFFICIALS AND MANAGERS class
Create the Corporate managers class
Set up the subClassOf relation between the Corporate managers class and the LEGISLATORS, SENIOR OFFICIALS AND MANAGERS class
Create the PROFESSIONALS class
Create the Occupation class
Set up the subClassOf relation between the LEGISLATORS, SENIOR OFFICIALS AND MANAGERS class and the Occupation class
Set up the subClassOf relation between the PROFESSIONALS class and the Occupation class

Related Resources

This pattern is related to the architectural pattern TX-AP-01 [4] for modelling a taxonomy.

3 Pattern Usage

This pattern was applied to re-engineer the ISCO-88(COM)[4], International Standard Classification of Occupations (for European Union purposes), into a Occupation Ontology[5], within the context of the SEEMP project. This standard is a classification scheme which consists of 520 occupations. ISCO-88(COM) is modelled following the path enumeration data model. Because of the number of occupations of the ISCO-88(COM) standard, it was not practical to create the

[4] Available at http://ec.europa.eu/eurostat/ramon/

[5] The ontology is available at http://droz.dia.fi.upm.es/hrmontology/

ontology manually. Therefore, we created an *ad-hoc* wrapper, implemented in Java, that reads the data from the resource implementation and automatically creates the corresponding elements of the new ontology following the suggestion given by the pattern.

4 Summary and Future Work

We have presented a pattern for transforming a classification scheme, which is modelled following the path enumeration data model, into a taxonomy. The pattern is included in a pool of patterns, which is a key element of our method for re-engineering non-ontological resources into ontologies [2].

We plan to develop software libraries within a framework that implement the transformation process suggested by the pattern. Moreover, we will include external resources to improve the quality of the resultant ontologies. Finally, we need to calculate how much effort do we save re-engineering classification schemes using patterns compared with re-engineering classification schemes without them.

Acknowledgments. This work has been partially supported by the European Comission projects NeOn(FP6-027595) and SEEMP(FP6-027347), as well as by an R+D grant from the UPM.

References

1. D. Brandon. Recursive database structures. *Journal of Computing Sciences in Colleges*, 2005.
2. A. García, A. Gómez-Pérez, M. C. Suárez-Figueroa, and B. Villazón-Terrazas. A Pattern Based Approach for Re-engineering Non-Ontological Resources into Ontologies. In *Proceedings of the 3rd Asian Semantic Web Conference (ASWC2008)*. Springer-Verlag, 2008.
3. V. Presutti, A. Gangemi, S. David, G. Aguado de Cea, M. C. Surez-Figueroa, E. Montiel-Ponsoda, and M. Poveda. NeOn Deliverable D2.5.1. A Library of Ontology Design Patterns: reusable solutions for collaborative design of networked ontologies. In *NeOn Project. http://www.neon-project.org*, 2008.
4. M. C. Suárez-Figueroa, S. Brockmans, A. Gangemi, A. Gómez-Pérez, J. Lehmann, H. Lewen, V. Presutti, and M. Sabou. Neon modelling components. Technical report, NeOn project deliverable D5.1.1, 2007.

Pattern for Re-engineering a Classification Scheme, Which Follows the Adjacency List Data Model, to a Taxonomy

http://ontologydesignpatterns.org/wiki/Submissions:Classification_scheme_-_adjacency_list_model_-_to_Taxonomy

Boris Villazón-Terrazas[1], Mari Carmen Suárez-Figueroa[1], and Asunción Gómez-Pérez[1]

Ontology Engineering Group, Departamento de Inteligencia Artificial, Facultad de Informática, Universidad Politécnica de Madrid, Spain
{bvillazon,mcsuarez,asun}@fi.upm.es,
WWW home page: http://www.oeg-upm.net/

1 Introduction

This pattern for re-engineering non-ontological resources (PR-NOR) fits in the Schema Re-engineering Category proposed by [3]. The pattern defines a procedure that transforms the classification scheme components into ontology representational primitives. This pattern comes from the experience of ontology engineers in developing ontologies using classification schemes in several projects (SEEMP[1], NeOn[2], and Knowledge Web[3]). The pattern is included in a pool of patterns, which is a key element of our method for re-engineering non-ontological resources into ontologies [2]. The patterns generate the ontologies at a conceptualization level, independent of the ontology implementation language.

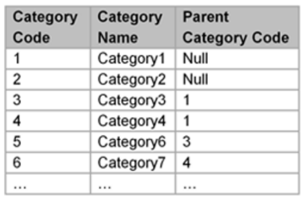
2 Pattern

Problem

Re-engineering a classification scheme, which follows the adjacency list model, to design a taxonomy.

Non-Ontological Resource

A non-ontological resource holds a classification scheme which follows the adjacency list model. A classification scheme is a rooted tree of concepts, in which each concept groups entities by some particular degree of similarity.
The semantics of the hierarchical relation between parents and children concepts may vary depending of the context. The adjacency list data model [1] for hierarchical classifications proposes to create an entity which holds a list of items with a linking column associated to their parent items.



Category Code	Category Name	Parent Category Code
1	Category1	Null
2	Category2	Null
3	Category3	1
4	Category4	1
5	Category6	3
6	Category7	4
...	...	...

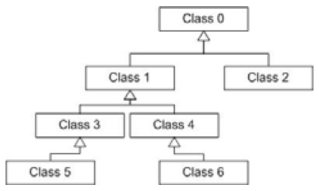
[1] http://www.seemp.org
[2] http://www.neon-project.org
[3] http://knowledgeweb.semanticweb.org

Applicability

The semantics of the relation between parent and children items are *subClassOf*.
There is not multi-inheritance nor cyclic relations.

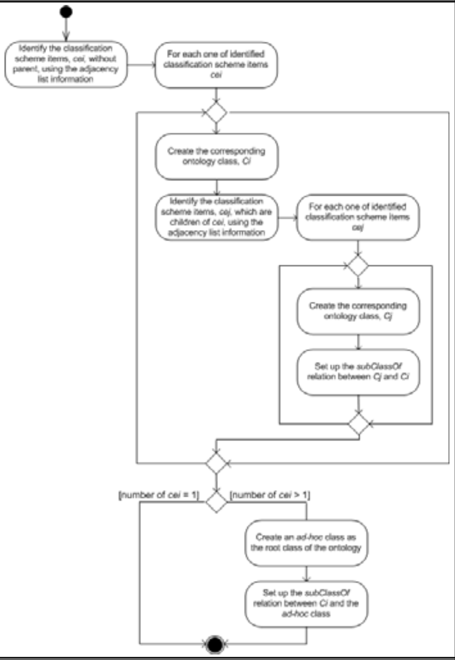
Ontology Generated

The ontology generated will be based on the taxonomy architectural pattern (AP-TX-01) [4].
Each category in the classification scheme is mapped to a class, and the semantics of the relationship between children and parent categories are mapped to *subClassOf* relations.



Process - Solution

1. Identify the classification scheme items which do not have a parent key value, i.e. classification scheme items without parents.
2. For each one of the above identified classification scheme items ce_i:
   - 2.1. Create the corresponding ontology class, C_i class.
   - 2.2. Identify the classification scheme items, ce_j, which are children of ce_i, by using the parent key values.
   - 2.3. For each one of the above identified classification scheme items ce_j:
     - 2.3.1. Create the corresponding ontology class, C_j class.
     - 2.3.2. Set up the *subClassOf* relation between C_j and C_i.
     - 2.3.3. Repeat from step 2.2 for ce_j as a new ce_i.
3. If there are more than one classification scheme items without parent ce_i
   - 3.1. Create an *ad-hoc* class as the root class of the ontology.
   - 3.2. Set up the *subClassOf* relation between C_i class and the root class.

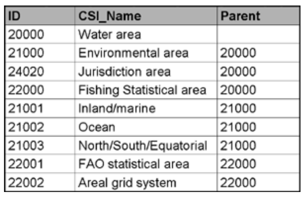

Identify the classification scheme items, cei, without parent, using the adjacency list information
For each one of identified classification scheme items cei
Create the corresponding ontology class, Ci
Identify the classification scheme items, cej, which are children of cei, using the adjacency list information
For each one of identified classification scheme items cej
Create the corresponding ontology class, Cj
Set up the subClassOf relation between Cj and Ci
[number of cei = 1]
[number of cei > 1]
Create an ad-hoc class as the root class of the ontology
Set up the subClassOf relation between Ci and the ad-hoc class

Example

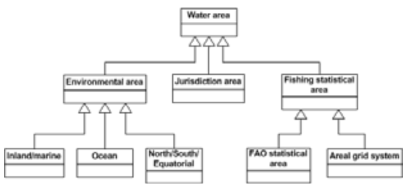
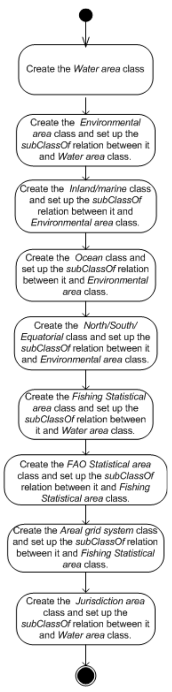
Suppose that someone wants to build an ontology based on the water areas classification published by FAO. This classification scheme follows the adjacency list data model.

Non-Ontological Resource

The FAO classification for water areas groups them according to some different criteria as environment, statistics, and jurisdiction, among others. This classification scheme is available at http://www.fao.org/figis/servlet/RefServlet



ID	CSI_Name	Parent
20000	Water area	
21000	Environmental area	20000
24020	Jurisdiction area	20000
22000	Fishing Statistical area	20000
21001	Inland/marine	21000
21002	Ocean	21000
21003	North/South/Equatorial	21000
22001	FAO statistical area	22000
22002	Areal grid system	22000

Ontology Generated	
The ontology generated will be based on the taxonomy architectural pattern (AP-TX-01) [4]. Each category in the classification scheme is mapped to a class, and the semantics of the relationship between children and parent categories are mapped to *subClassOf* relations.	 Water area Environmental area Jurisdiction area Fishing statistical area Inland/marine Ocean North/South/ Equatorial FAO statistical area Areal grid system
Process - Solution	
1. Create the `Water area` class. 2. Create the `Environmental area` class, and set up the *subClassOf* relation between the `Environmental` area class and the `Water area` class. 2.1. Create the `Inland/marine` class, and set up the *subClassOf* relation between the `Inland/marine` class and the `Environmental area` class. 2.2. Create the `Ocean` class, and set up the *subClassOf* relation between the `Ocean` class and the `Environmental area` class. 2.3. Create the `North/South/Equatorial` class, and set up the *subClassOf* relation between the `North/South/Equatorial` class and the `Environmental area` class. 3. Create the `Fishing Statistical area` class, and set up the *subClassOf* relation between the `Fishing Statistical area` class and the `Water area` class. 3.1. Create the `FAO statistical area` class, and set up the *subClassOf* relation between the `FAO statistical area` class and the `Fishing Statistical area` class. 3.2. Create the `Areal grid system` class, and set up the *subClassOf* relation between the `Areal grid system` class and the `Fishing Statistical area` class. 4. Create the `Jurisdiction area` class, and set up the *subClassOf* relation between the `Jurisdiction area` class and the `Water area` class.	 Create the *Water area* class Create the *Environmental area* class and set up the *subClassOf* relation between it and *Water area* class. Create the *Inland/marine* class and set up the *subClassOf* relation between it and *Environmental area* class. Create the *Ocean* class and set up the *subClassOf* relation between it and *Environmental area* class. Create the *North/South/ Equatorial* class and set up the *subClassOf* relation between it and *Environmental area* class. Create the *Fishing Statistical area* class and set up the *subClassOf* relation between it and *Water area* class. Create the *FAO Statistical area* class and set up the *subClassOf* relation between it and *Fishing Statistical area* class. Create the *Areal grid system* class and set up the *subClassOf* relation between it and *Fishing Statistical area* class. Create the *Jurisdiction area* class and set up the *subClassOf* relation between it and *Water area* class.
Related Resources	
This pattern is related to the architectural pattern TX-AP-01 [4] for modelling a taxonomy.	

3 Pattern Usage

This pattern was applied to re-engineer the ISTAT[4], geography italian standard, into a Geography Ontology[5], within the context of the SEEMP project. This standard is a classification scheme which consists of 4 divisions, 20 regions and 106 provinces. ISTAT is modelled following the adjacency list data model. Because of the number of divisions, regions and provinces of the ISTAT standard, it was not practical to create the ontology manually. Therefore, we created an

[4] http://www.istat.it/

[5] The ontology is available at http://droz.dia.fi.upm.es/hrmontology/

ad-hoc wrapper, implemented in Java, that reads the data from the resource implementation and automatically creates the corresponding elements of the new ontology following the suggestion given by the pattern.

4 Summary and Future Work

We have presented a pattern for transforming a classification scheme, which is modelled following the adjacency list data model, into a taxonomy. The pattern is included in a pool of patterns, which is a key element of our method for re-engineering non-ontological resources into ontologies [2].

We plan to develop software libraries within a framework that implement the transformation process suggested by the pattern. Moreover, we will include external resources to improve the quality of the resultant ontologies. Finally, we need to calculate how much effort do we save re-engineering classification schemes using patterns compared with re-engineering classification schemes without them.

Acknowledgments. This work has been partially supported by the European Comission projects NeOn(FP6-027595) and SEEMP(FP6-027347), as well as by an R+D grant from the UPM.

References

1. D. Brandon. Recursive database structures. *Journal of Computing Sciences in Colleges*, 2005.
2. A. García, A. Gómez-Pérez, M. C. Suárez-Figueroa, and B. Villazón-Terrazas. A Pattern Based Approach for Re-engineering Non-Ontological Resources into Ontologies. In *Proceedings of the 3rd Asian Semantic Web Conference (ASWC2008)*. Springer-Verlag, 2008.
3. V. Presutti, A. Gangemi, S. David, G. Aguado de Cea, M. C. Surez-Figueroa, E. Montiel-Ponsoda, and M. Poveda. NeOn Deliverable D2.5.1. A Library of Ontology Design Patterns: reusable solutions for collaborative design of networked ontologies. In *NeOn Project. http://www.neon-project.org*, 2008.
4. M. C. Suárez-Figueroa, S. Brockmans, A. Gangemi, A. Gómez-Pérez, J. Lehmann, H. Lewen, V. Presutti, and M. Sabou. Neon modelling components. Technical report, NeOn project deliverable D5.1.1, 2007.

Negative Property Assertion Pattern (NPAs)

http://
ontologydesignpatterns.org/wiki/Submissions:NegativePropertyAssertions

Olaf Noppens

Inst. of Artificial Intelligence
Ulm University
Germany
olaf.noppens@uni-ulm.de

1 Introduction

The basic building blocks of Description Logics (DLs) in general are concept and role constructors (e. g. , intersection, union, nominals, negation etc.), terminological as well as individual axioms. Combining axioms, meaningful statements can be expressed. Additional axiom constructors (so-called syntactical sugar) has been introduced in most ontology language (such as OWL and OWL 2 [1]) in order to simplify ontology modeling, understanding and maintenance. These syntactical sugar axioms can be reduced to basic axioms. A simple example is disjointness of concepts that can be reduced to GCI axioms. However, even experts have often difficulties to understand the reduced forms in comparison to syntactical sugar forms. From the modeling perspective, patterns can support the user in modeling logical meaning that is not directly supported in an ontology language. For instance, OWL 2 provides a syntactical form for modeling *negative property assertions* whereas using the predecessor OWL 1 one have to model it by a more or less complicated inclusion axiom (or, alternatively, disjoint axiom).

The motivation of this pattern is to model *negative property assertions* (NPAs) in ontology languages such as OWL 1 that do not provide a special construct for it. It is worth mentioning that not all knowledge base systems can be migrated to OWL 2 for several reasons. On the other hand, NPAs modeled according to this pattern can be migrated to OWL 2 using the newly introduced constructor. A negative property assertion as defined in the upcoming OWL 2 states that a given individual i is *never* connected to a given individual j by a given property expression P. In other words, asserting that i is connected to j by P results in an inconsistent ontology. In this sense this assertion can be considered as a constraint that should not be violated. In contrast, considering an ontology where it cannot be inferred that i is connected to j by p does not necessarily mean that there cannot be such a connection – in fact, it is merely not modeled.

2 Pattern

In this section we describe the Negative Property Assertion Pattern in detail.

2.1 Problem

Prior to OWL 2 it is difficult to model *negative property assertions* and, if they are contained in an ontology, difficult to understand because OWL 1 does not provide a specialized constructor for it. In addition, with help of this pattern one can also migrate OWL 1 ontologies containing axioms describing NPAs into the NPA constructor of OWL 2.

Intent This patterns allows to express NPAs in ontologies written in an ontology language such as OWL 1 that does not allow NPA directly. The pattern can also be understood as a transformation rule for transforming axioms modeling NPAs into the direct NPA axiom constructor of OWL 2.

2.2 Solution

A negative property assertion that states that an individual $Individual_1$ is not connected to an individual $Individual_2$ by a property *property* can be modeled as the following inclusion axiom: $\{Individual_1\} \sqsubseteq \neg\ (\exists property\ \{Individual_2\})$ [1]. Here we are using the German DL syntax where $\{x\}$ denotes an enumeration class with a single individual x (aka. 'one-of'). A graphical representation of this pattern is given in Figure 1.

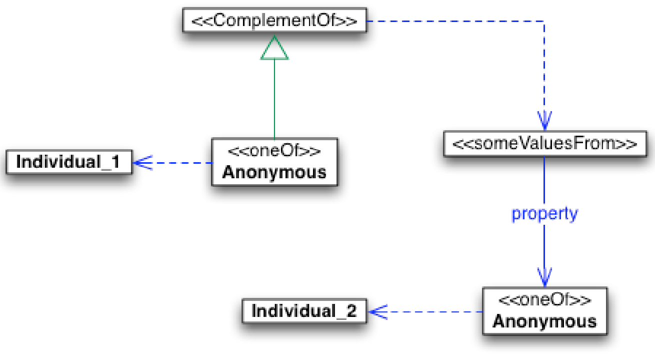


Fig. 1. Graphical representation of the pattern using the UML-based notation proposed in [2].

In the following we proof the equivalence between the axiom produced by the pattern and NPA as defined in OWL 2.

[1] Note that this can also be modeled with help of a disjointness axiom. However, not all languages provide a special disjointness constructor.

Proof Let C and D be concepts. Then C and D are disjoint if, and only if, C is subsumed by the complement of D, i. e., ($C \sqsubseteq \neg D$). The equivalence is correct because of the duality of disjointness, equivalence, and unsatisfiability: C is subsumed by D if, and only if, $C \sqcup \neg D$ is unsatisfiable, and the intersection of C and D is unsatisfiable if, and only if, C and D are disjoint.
One also reminds that the extension of the concept $\exists\, prop.C$ is the set of individuals i which are connected to an individual j that is in the extension of the concept C, by the property $prop$.
Let $NPA\,(p\,a\,b)$ be a negative property assertion axiom, i. e., the individual a is not related to b by the property p. Then the extension of $\exists\, p.\,\{b\}$ which contains all individuals that are connected to b by p must not contain a. This is true, if, and only if, $\{a\}$ is disjoint to $\exists\, p.\,\{b\}$.

Consequences Applying this pattern to an ontology will add the logical meaning of a NPA to the ontology. There are not any restrictions or limitations of the solution besides that nominals and unrestricted existential quantification must be supported in the target ontology language.

2.3 Example

Consider a social network containing facts about people and their relationships. Let `Adam` and `Eve` be two persons and `like` a property ('A likes B'). Furthermore we know that `Adam` does not like `Eve` but we have no dislike relationship. Moreover, our language (such as OWL 1) does not have any NPA axiom constructor.

The sample ontology is interpreted with respect to the open-world semantics, i. e., one can not infer the dislike merely from the lack of a property assertion axiom `Adam like Eve`. Then this fact can be expressed with the following axiom in OWL 1: $\{Adam\} \sqsubseteq \neg\ (\exists likes\ \{Eve\})$.

3 Pattern Usage

The NPA pattern is useful in all situations where one has to model a negative property assertion but the language does not allow it directly. One real-world example here is the one mentioned in Section 2.3 where we had modeled a social network structure and need a possibility to express that one person does not like another and to ensure that one person does not know another person. With help of NPAs we have the possibility to distinguish between 'Person A does not know Person" and "We do not know whether Person A knows Person B or not".

4 Summary

The *Negative Property Assertion Pattern* allows to express negative property assertions in languages such as OWL 1 that do not a build-in constructor for this kind of assertions. In addition, when migrating OWL 1 ontologies to OWL 2 ontologies this pattern can be used to find negative property assertions and to transform them into the special NPA constructor of OWL 2.

References

1. Motik, B., Patel-Schneider, P.F., Parsia, B.: OWL 2 Structural Specification and Functional-Style Syntax. W3C Candidate Recommendation 11 June 2009 (June 2009)
2. Borckmans, S., Volz, R., Eberhart, A., Löffler, P.: Visual Modeling of OWL DL Ontologies Using UML. 198–213

ConceptTerms

http://ontologydesignpatterns.org/wiki/Submissions:ConceptTerms

Pierre-Yves Vandenbussche[1,2] and Jean Charlet[1,3]

[1] INSERM UMRS 872, éq.20, 15, rue de l'école de médecine, 75006 Paris, France
[2] MONDECA, 3, cité Nollez, 75018 Paris, France
[3] DSI AP-HP, Paris, FRANCE
`pierre-yves.vandenbussche@etu.jussieu.fr`
`jean.charlet@spim.jussieu.fr`

1 Introduction

The integration of domain terminological resources is a major issue for their full utilization. This integration is made difficult by the nature of these resources and their formal representation heterogeneity. In order to tackle this integration problem, we foster a solution based on metamodeling: the unfied metamodel for structured vocabularies. In this abstract, we sketch out assets for using Ontology Design Patterns (ODPs) in a unified metamodel by reusing n-ary relation pattern and describing new ODP for concept-terms representation. This ConceptTerms ODP allows designers to represent jointly conceptual and linguistic part of a vocabulary. The pattern purpose is not to encompass all linguistic complexity as Linginfo or LMF does, but to describe linguistic information in more details than SKOS which names concept with simple labels.

2 Pattern

2.1 Problem

Intent ConceptTerms pattern allows designers to represent jointly conceptual and linguistic part of a vocabulary. The pattern purpose is not to encompass all linguistic complexity as Linginfo or LMF does, but to describe linguistic information in more details than SKOS which names concept whith simple labels.

Domains Linguistic, Vocabulary.

Covers Requirements What preferred terms and synonyms (simple non preferred term) have a concept? What is the preferred term of a concept in a specific language? By which preferred term a coumpound non preferred term is composed?

2.2 Solution

A concept is named in a particular language by a preferred term and a set of simple non preferred terms (multilinguism). Those terms artifacts specialize the term entity which owns common properties. This list of properties may be extended depending on vocabulary specific needs. This pattern suits for various vocabularies (thesaurus, terminology, taxonomy) and has been applied to GEMET, Eurovoc, CIM10 among other. Modeling takes into account: - the possibility to extend the current pattern in order to add some more precise linguistic information (for instance represent translation relation between two terms since term is a class) - minimal linguistic artifacts necessary for vocabulary resource access by providing a preferred Term to name a concept and some synonyms which are Simple non preferred terms.

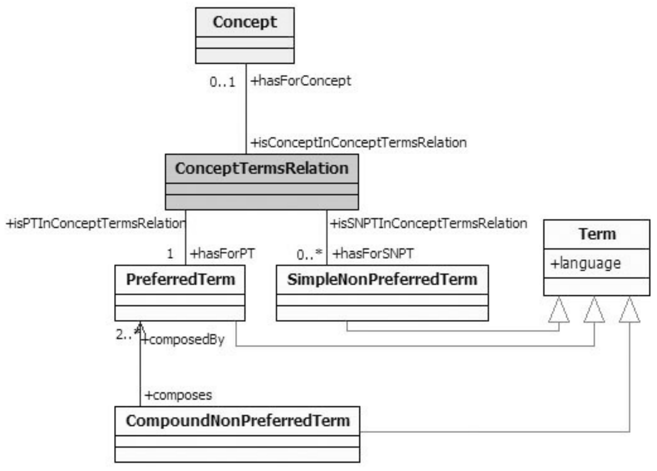


Fig. 1. The ConceptTerms Content ODP's graphical representation in UML.

Consequences Compare to labels on a concept class, this solution has a higher data load.

2.3 Example

Used for vocabulary representation. For example in Eurovoc (http://europa.eu/eurovoc/), a concept has a preferred term *social sciences* in english and a simple non preferred term (i.e. synonyms) *humanities* in the same language whereas the same concept has a preferred term *sciences sociales* in french and a simple non preferred term *sciences humaines* in this language. If we wanted to add a translation relation between terms we could state that *social sciences* english term is a translation of *sciences sociales* french term. If we consider a second preferred term in english *award* which names a concept, in a particular information retrieval context, we could define a coumpound non preferred term *social sciences awards* which is related to preferred terms *social sciences* and *award* .

2.4 Related Resources

In BS8723 model, triangular-shaped relations are defined between a *thesaurus concept*, a *preferred term* and some *simple non preferred terms.* We are convinced that maintaining this model can be optimized by reifying those relations in a single relation class. That is why we defines the *Concept-Terms relation* which reusing N-ary pattern in order to represent terms on a concept. Between all terms, we distinguish a preferred term and some synonyms (simple non preferred terms).

3 Pattern Usage

This pattern suits for various vocabularies (thesaurus, terminology, taxonomy) and has been applied to Eurovoc authoring management.

4 Summary and Future Work

This pattern allows ontology designers to represent jointly conceptual and linguistic part of a vocabulary. Therefore, it is a complementary pattern to linguistic ones.

Concept Partition Pattern

http:// ontologydesignpatterns.org/wiki/Submission:Partition

Olaf Noppens

Inst. of Artificial Intelligence
Ulm University
Germany
olaf.noppens@uni-ulm.de

1 Introduction

The *Partition Pattern* is a logical pattern that introduces axioms which model a partition of concepts. A partition is a general structure which is divided into several disjoint parts. With respect to ontologies the structure is a concept which is divided into several pair-wise disjoint concepts. This pattern reflects the simplest case where a named concept is defined as a partition of concepts.

2 Pattern

2.1 Problem

The *Partition Pattern* describes how to model a partition, i. e., a named concept which is divided into several disjoint concepts. Applying this pattern to an ontology will introduce the necessary axioms.

2.2 Solution

Let P be a named concept that is the partition which is divided into several concepts C_i. Then the partition is defined by introducing the following axioms (expressed in the standard DL syntax) as also be shown in Figure 1:
$P \equiv C_0 \sqcup C_1 \sqcup \cdots \sqcup C_n$ and $C_i \sqcap C_j = \bot$ for $0 \leq i, j \leq n, i \neq j$.
Note that some ontology languages such as OWL 2 [1] provides disjointness axioms as syntactical sugar to make the definition of pair-wise disjointness easier. In OWL 2 the pattern can be translated into the following axioms (expressed in OWL 2 abstract syntax):

```
EquivalentClasses(P ObjectUnionOf(C0, C1, ..., Cn))
DisjointClasses(C0, C1, ..., Cn)
```

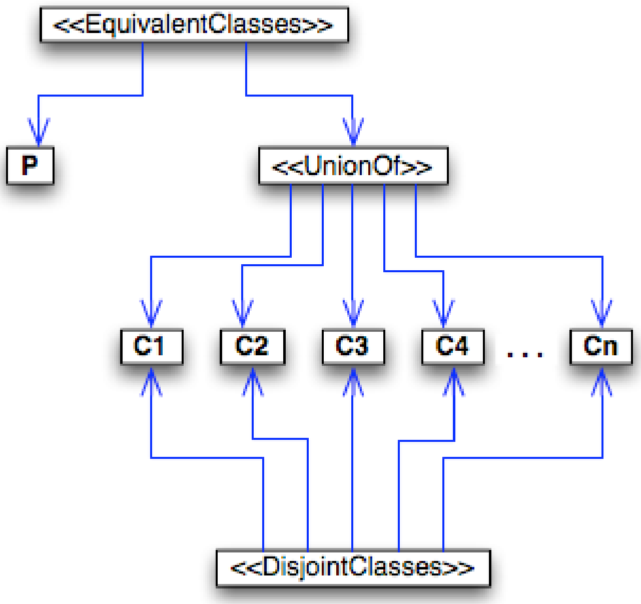


Fig. 1. Graphical representation of the *Partition Pattern*. Here, *EquivalentClasses* and *DisjointClasses* are axioms and *ObjectUnionOf* is a class constructor building a disjunction over an arbitrary number of concepts C_0, ..., C_n.

2.3 Example

Consider a world where only humans and animals live. Then the inhabitants of this world are partitioned into humans and animals. The following two axioms

```
EquivalentClasses(Inhabitant ObjectUnionOf(Human, Animal))
DisjointClasses(Human, Animal)
```

describe the partitions of inhabitants into human and animals.

3 Pattern Usage

In an ontology about family relationship we defined concepts such as `Person`, `Aunt` and `ParentOfSon` which are characterized by a relationships such as `hasChild` (resp. the inverse relationship `hasParent`), `hasSibling`, `married-with` as well as by the gender of people (`Male` respectively `Female`). There are a lot of similar ontologies about family relationships. Our version can be downloaded at http://www.informatik.uni-ulm.de/ki/Noppens/generation.owl.

```
EquivalentClasses(Parent-Of-Son
     ObjectSomeValuesFrom (has-Child Male) )
EquivalentClasses(Parent-Of-Daughter
     ObjectSomeValuesFrom(has-Child Female) )
EquivalentClasses(Aunt ObjectIntersectionOf(Uncle-Or-Aunt Female))
EquivalentClasses(Uncle-Or-Aunt
     ObjectIntersectionOf(Person
         ObjectSomeValuesFrom(has-Sibling Parent)))
```

The concept `Gender` is partitioned in `Male` and `Female`. Applying this pattern results in the following axioms:

```
EquivalentClasses(Gender, ObjectUnionOf(Male  Female))
DisjointClasses(Male Female)
```

4 Summary and Future Work

The *Partition Pattern* describes how to model a partition. The pattern reflects the simplest case where a *named* concept is the partition of (arbitrary) concepts. Future work will be concerned with a more general variant of this pattern: in some cases, the partition concept is not explicitly named (i. e., is not a named concept) but implicitly used, for instance, as value range of quantifications. In other words, no equivalent class axiom will be used but the value range is the union of the pair-wise disjoint parts of the partition.

References

1. Motik, B., Patel-Schneider, P.F., Parsia, B.: OWL 2 Structural Specification and Functional-Style Syntax. W3C Candidate Recommendation 11 June 2009 (June 2009)

Part 3:

Short Papers (Posters)

Pattern Definitions and Semantically Annotated Instances

Ivan Perez[1] and Oscar Corcho[2]

[1] IMDEA Software*. Universidad Politécnica de Madrid, Spain.
`ivan.perez@imdea.org`
[2] Ontology Engineering Group. Departamento de Inteligencia Artificial.
Universidad Politécnica de Madrid, Spain.
`ocorcho@fi.upm.es`

Abstract Ontology design patterns are normally instantiated by replicating and adapting the pattern concepts and roles. The relation between pattern definitions and their instantiations is documented in natural language. The use of parametric ontologies or pattern-reuse modifications to OWL-DL has been suggested before, but so far only practical aspects have been analysed, leaving the formal semantics of these extensions as future work. In this work we present formal definitions for ontology pattern and pattern instantiations, together with the semantics of instantiation. We propose the use of semantic annotations to describe and generate OWL pattern instantiations, without the need for explicit ontology replication, and provide tools to support this process.

1 Introduction

Ontology patterns represent knowledge that is subject to appear frequently in different ontologies, normally with different names. Relations such as "being part of something", "participating in an event" and "n-ary relations" are examples of such frequent type of knowledge.

Pattern instantiation is normally done in two different ways: (1) importing the pattern axioms and establishing mappings (equivalences) with existing elements in our ontology, or (2) replicating the pattern by changing the names of its elements. The former leads to representation and reasoning problems in case we use the pattern twice in the same ontology, as described in section 2. The latter is error prone and loses the relation between the instantiations and the pattern, which is in some cases only described with comments in natural language.

Ad-hoc pattern definition and instantiation languages have been proposed for this purpose. However, we propose using the ontology language expressiveness, by means of semantic annotations, to support this pattern instantiation process.

This paper is organised as follows. In section 2 we introduce some of the problems found when trying to reuse ontology design patterns. In section 3

* The work of IMDEA Software on this paper has been partially funded by the Spanish Ministry of Industry, Tourism and Trade under the grant FIT-340503-2007-2, as part of the Morfeo EzWeb strategic singular project.

we define formally the concepts of pattern and pattern instance. In section 4 we describe our tools for supporting pattern reuse. Finally, section 5 presents conclusions, related work and further research topics in this area.
Notation. We use the notation $\textsc{Concept}$ for concepts or classes, $\mathsf{property}$ for properties or roles, and *individual* for individuals. The notation used for DL formulas, unless stated otherwise, follows the one presented in [1].

2 Problem description

Let us start with an example where we want to represent cities as part of provinces, which we will use as an ontological pattern. We use the concepts $\textsc{City}$ and $\textsc{Province}$, the property $\mathsf{isPartOf}$, and the following axioms[3]:

$$\begin{aligned} \top &\sqsubseteq \forall \mathsf{isPartOf}^{-1}.\textsc{City} \\ \top &\sqsubseteq \forall \mathsf{isPartOf}.\textsc{Province} \\ \textsc{City} &\sqsubseteq =1\ \mathsf{isPartOf}.\top \end{aligned}$$

If we add the concepts $\textsc{GermanCity}$ and $\textsc{GermanProvince}$, we can reuse the above axiom to relate them. With the following axioms, we can use the relation $\mathsf{isPartOf}$ also for German cities and provinces:

$$\begin{aligned} \textsc{GermanCity} &\sqsubseteq \textsc{City} \\ \textsc{GermanProvince} &\sqsubseteq \textsc{Province} \end{aligned}$$

These axioms do not guarantee that a $\textsc{GermanCity}$ belongs to ($\mathsf{isPartOf}$) a $\textsc{GermanProvince}$. This situation is more clear if we add new subconcepts of $\textsc{City}$ and $\textsc{Province}$, for instance, $\textsc{FrenchCity}$ and $\textsc{FrenchProvince}$:

$$\begin{aligned} \textsc{FrenchCity} &\sqsubseteq \textsc{City} \\ \textsc{FrenchProvince} &\sqsubseteq \textsc{Province} \end{aligned}$$

According to our axioms, it could be the case that a $\textsc{FrenchCity}$ is part of a $\textsc{GermanProvince}$. To solve this, we may add the following axioms:

$$\begin{aligned} \textsc{GermanCity} &\equiv \forall \mathsf{isPartOf}.\textsc{GermanProvince} \\ \textsc{FrenchCity} &\equiv \forall \mathsf{isPartOf}.\textsc{FrenchProvince} \end{aligned}$$

This solution is correct for our case. Note, however, that we have proposed an ad-hoc solution to a simple problem of multiple instantiation of one pattern. In larger ontologies, it may be easier to simply duplicate the original pattern using different names for each element in the pattern:

[3] We use this adhoc "pattern" to simplify our explanation, since we have not found any well-known ontology pattern that contained these axioms and could be used instead. Potential candidate patterns like Componency [12], Collection [4], Collection Entity [11] or Classification [10] lacked some of the concepts or axioms.

$$\begin{aligned}\top &\sqsubseteq \forall \mathsf{gIsPartOf}^{-1}.\textsc{GermanCity}\\ \top &\sqsubseteq \forall \mathsf{gIsPartOf}.\textsc{GermanProvince}\\ \textsc{GermanCity} &\sqsubseteq {=1}\ \mathsf{gIsPartOf}.\textsc{GermanProvince}\end{aligned}$$

with a similar description for the concepts FRENCHCITY and FRENCHPROVINCE, and the role fIsPartOf. To document the relation between the pattern and its instances, we may keep the subclass relations above and add the following axioms:

$$\begin{aligned}\mathsf{gIsPartOf} &\sqsubseteq \mathsf{isPartOf}\\ \mathsf{fIsPartOf} &\sqsubseteq \mathsf{isPartOf}\end{aligned}$$

but this adds one new axiom for each element that is part of an instance of a pattern, and that relation is often just documented in natural language (as an `rdfs:comment` associated to the role).

3 Pattern definitions and instantiations

Two key elements are needed in ontologies to support patterns: a means to define the patterns and a way to use them. Before going into details, we introduce a few elementary definitions that will be used along the paper. We use C for a set of concept names, R for a set of role names, and I for individual names. We consider a DL, with a set of symbols S, as a language over the union of all four sets C, R, I and S. DL sentences are formulas, and an ontology is a finite set of formulas of a particular DL. We consider C^+, R^+, I^+ and S^+ pairwise disjoint[4]. Finally, given a set W, a list of symbols $w_1, \ldots, w_n \in W$ all different, and a second list of symbols $w'_1, \ldots, w'_n \in W$, we define the substitution function from $w_1, \ldots, w_n \in W$ to $w'_1, \ldots, w'_n$ respectively, as the function $f \subseteq W \times W$ such that:

$$f(x) = \begin{cases} w'_i & \text{if there is an } i \in \{1, \ldots, n\} \text{ such that } w_i = x \\ x & \text{otherwise} \end{cases}$$

We refer to that substitution function as the mapping $[w_1 \mapsto w'_1, \ldots, w_n \mapsto w'_n]$ or $[f]$. Given a substitution function f, and given a string w of elements in W of length n (that is, $w : [n] \to W$)[5], we define the substitution of w under the total substitution function $f : W \to W$, and we represent it as $w[f]$, as the string $w' : [n] \to W$ such that $\forall i \in [n]\ w'(i) = f(w(i))$.

3.1 Pattern definitions

Ontology patterns are defined as DL models or UML diagrams, plus descriptions in natural language. If we use DL, we have no standard way of establishing

[4] We use the notation X^+ for the Kleene plus, that is, the Kleene closure of the set X without the empty string. More details can be found in [3].

[5] We follow the notation for strings and alphabets in [3]. $[n]$, where n is a natural number, represents the subset of natural numbers from 1 to n.

which parts of the model are meant to be substituted. In case we use UML, the semantics in ontologies when the pattern is instantiated is not clear.

In our proposal, we also use DL models to define patterns. However, we add an interface to the pattern, that is, a definition of which parts are instantiable. This allows us to establish if an element in a pattern is not meant to be substituted or instantiated. This can happen because a name is just presented to simplify the axioms, or if the element in question represents more general knowledge.

As an example, we will use the model in section 2, with concepts City and Province, the property isPartOf, and the following axioms:

$$\begin{aligned}\top &\sqsubseteq \forall \mathsf{isPartOf}^{-1}.\textsc{City}\\ \top &\sqsubseteq \forall \mathsf{isPartOf}.\textsc{Province}\\ \textsc{City} &\sqsubseteq {=1}\ \mathsf{isPartOf}.\top\end{aligned}$$

This model and the interface $\{\textsc{City}, \textsc{Province}, \mathsf{isPartOf}\}$ is a pattern definition.

Definition 1. (Ontological knowledge pattern)
Given an ontology O, and given the sets $C' \subseteq C$, $R' \subseteq R$ and $I' \subseteq I$. The tuple $\langle O, C', R', I' \rangle$ is an ontological knowledge pattern.

Unless stated otherwise, we will normally refer to ontology patterns as pattern definitions or simply patterns. If $\langle O, C', R', I' \rangle$ is a pattern, we will refer to C', R' and I' as the interface of the pattern.

3.2 Pattern instantiations

A model is an instantiation of a pattern with a parameter/value assignation if the model has exactly the same axioms as the pattern, where the parameters have been substituted with their corresponding values. Note that the values must be of the same kind, that is, we can substitute or instantiate a concept with a concept, a role with a role and an individual with an individual.

Given the pattern definition above, we can instantiate it with the assignations FrenchCity to City and FrenchProvince to Province, having as a result the following model:

$$\begin{aligned}\top &\sqsubseteq \forall \mathsf{isPartOf}^{-1}.\textsc{FrenchCity}\\ \top &\sqsubseteq \forall \mathsf{isPartOf}.\textsc{FrenchProvince}\\ \textsc{FrenchCity} &\sqsubseteq {=1}\ \mathsf{isPartOf}.\top\end{aligned}$$

The following is a formal definition of pattern instantiation:

Definition 2. (Pattern instantiation)
Let $\langle O, C', R', I' \rangle$ be a pattern, where O is an ontology over some description logic D. Let $c_1, \ldots, c_m \in C', k_1, \ldots, k_m \in C$ be concepts, $r_1, \ldots, r_n \in R'$, $s_1, \ldots, s_n \in R$ be roles, and $i_1, \ldots, i_p \in I', j_1, \ldots, j_p \in I$ be individuals.

We say that O' is an instance of the pattern $\langle O, C', R', I' \rangle$ with the mappings $c_1 \mapsto k_1, \ldots, c_m \mapsto k_m, r_1 \mapsto s_1, \ldots, r_n \mapsto s_n, i_1 \mapsto i_1, \ldots, i_p \mapsto j_p$ if $O' = \{o[c_1 \mapsto k_1, \ldots, c_m \mapsto k_m, r_1 \mapsto s_1, \ldots, r_n \mapsto s_n, i_1 \mapsto i_1, \ldots, i_p \mapsto j_p] \mid o \in O\}$ and O' is also an ontology over D.

3.3 Rebasing ontologies to avoid name clashing

We now address a practical issue commonly found when *partially* instantiating the same pattern twice as part of the same ontology. Imagine that we instantiate the city pattern for French and German cities, and do not want to name isPartOf differently in each case. The following axioms would be part of the result:

$$\top \sqsubseteq \forall \mathsf{isPartOf}^{-1}.\textsc{GermanCity}$$
$$\top \sqsubseteq \forall \mathsf{isPartOf}^{-1}.\textsc{FrenchCity}$$

which implies that either isPartOf is empty or GermanCity and FrenchCity are related (one is a subclass of the other).

Finding these problems may not be so obvious. Besides, name clashes may occur in non-instantiable elements. The pattern designer may document or avoid these issues, but it is the responsibility of the pattern user to make sure that no inconsistencies are introduced with multiple instantiations of a pattern.

Name clashes are not necessarily a mistake from a formal point of view, so these are only guidelines to pattern design. However, we will help avoid these situations by means of namespace (or URI) translations.

URIs and ontology rebasing. When coded in OWL, ontologies have a base URI and all entities local to it have that URI as a prefix of their complete names. By changing the base URI of an ontology, we are effectively renaming all the local entities at once, thus avoiding all name clashes of non-instanced local names.

In our formal definitions, we consider the unqualified name sets C_u, R_u and I_u, respectively, for concepts, names and individuals. We also consider a set M of namespaces, such that $C = M \times C_u$, $R = M \times R_u$, and $I = M \times I_u$. Like before, C_u, R_u and I_u are pairwise disjoint.

Definition 3. (Ontology rebasing)
Let O be an ontology, and $m, m' \in M$ two namespaces. We define the rebasing of O from m to m', and we represent it as $O^{m \mapsto m'}$, as the ontology $\{o[f] \mid o \in O\}$ where f is the substitution function defined by the relation $\{(x, y) \in C \cup R \cup I \times C \cup R \cup I \mid \exists z \in (C_u \cup R_u \cup I_u)\ x = \langle m, z \rangle \wedge y = \langle m', z \rangle\}$.

The previous definition of pattern instantiation is now extended as follows:

Definition 4. (Pattern instantiation)
We say that an ontology O is an instantiation of the pattern P with the namespace change from m to m' and the mappings $[e_1 \mapsto d_1, \ldots, e_n \mapsto d_n]$ if $O = O'^{m \to m'}$ and O' is an instantiation of the pattern P with the mappings $[e_1 \mapsto d_1, \ldots, e_n \mapsto d_n]$.

For example, assume we use `http://foo/Cities` as the base URI of the cities pattern and `http://foo/Cities#isPartOf` as the qualified name of isPartOf. If we set `http://foo/FCities` and `http://foo/GCities` as the base URIs of the instances for French and German cities respectively, applying the new pattern instance definition with the same mappings as before would give us the following axioms instead, where no name clashes occur:

$$\top \sqsubseteq \forall \texttt{http://foo/GCities\#}\mathsf{isPartOf}^{-1}.\texttt{http://foo/GCities\#}\textsc{GermanCity}$$
$$\top \sqsubseteq \forall \texttt{http://foo/FCities\#}\mathsf{isPartOf}^{-1}.\texttt{http://foo/FCities\#}\textsc{FrenchCity}$$

4 Tool support

We provide tool support for the processes of pattern definition and instantiation.

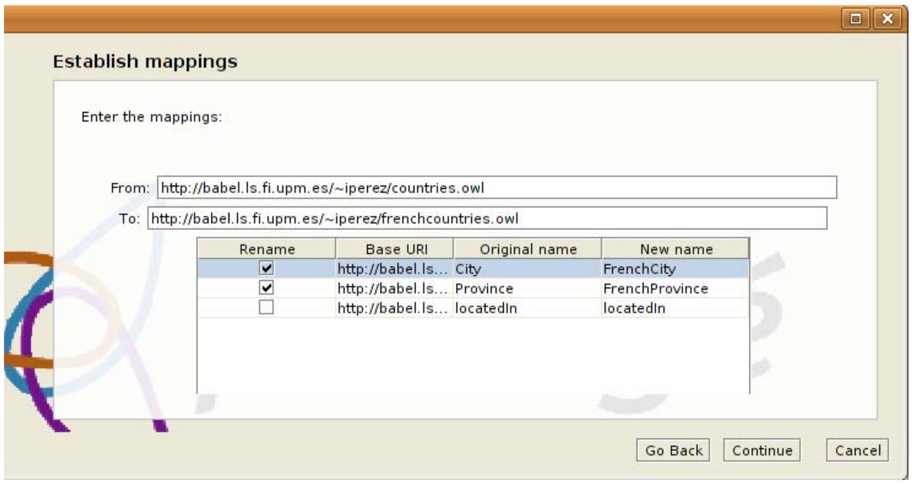
Pattern definition. Patterns are defined as ontologies with an interface, hence to define patterns we simply need to support *interface declaration*. For this purpose, we have developed:

- An ontology [7] with the annotation property exportable, which can be set for any class, property or individual, and is `true` if the element is part of the interface (instantiable) or `false` otherwise. Its default value is assumed to be `true`, so that users do not need to annotate all the elements in a pattern.
- A Protégé plugin [6] to assign values for this annotation property.

Pattern instantiation and use. Defining pattern instances means identifying the pattern to be used, establishing a parameter/value map and a URI rebasing. This task is supported with the following elements:

- An ontology [9] to describe instantiations. It contains concepts to represent mappings between entities and URI rebases.
- A Protégé plugin [6] that eases the creation of these instantiation A-Boxes. It also allows the user to apply a particular instantiation.
- An ontology [8] with an annotation property called `isPatternInstance`, that indicates that an ontology is the result of applying a particular pattern instance definition as defined in the first step.

The new Protégé plugin is accessible under the menu Tools, as an import *wizard*. The process is divided in three steps. First, the location of a pattern definition (an ontology) is provided. Second, the plugin shows all the elements in that pattern that are *exportable*. The user can then introduce the main information about the instantiation, that is, the new names for each entity that will be instantiated, and the URI rebase. Third, the plugin allows the user to select a location where the instantiation will be saved. This will create an ontology with all the information of this particular instantiation (the entity mappings and the URI rebase), located in a different file. A screenshot is shown in Fig. 1.


Establish mappings

Enter the mappings:

From: http://babel.ls.fi.upm.es/~iperez/countries.owl

To: http://babel.ls.fi.upm.es/~iperez/frenchcountries.owl

Rename	Base URI	Original name	New name
✓	http://babel.ls...	City	FrenchCity
✓	http://babel.ls...	Province	FrenchProvince
☐	http://babel.ls...	locatedIn	locatedIn

Figure 1. Screenshot of the instantiation wizard plugin running in Protégé 4

Once the process is finished, the plugin loads the pattern, applies the URI rebase and entity mapping, and adds all the axioms into the currently active ontology. It also annotates the ontology with the property `isPatternInstance`, using the base URI of the pattern instance definition as value.

5 Conclusions and future work

In this paper we have shown how patterns can be defined as ontologies with an interface (as if they were parametric ontologies), and how pattern instantiations can be formalised as a substitution of the parameters in a pattern ontology, providing formal definitions for both concepts. We have also analysed some of the problems derived from multiple pattern instantiation, and suggested ontology rebasing as a way to avoid name clashes.

Some notable work in pattern definition is presented in [5], where patterns are presented as part of an ontology modification language (OPPL), not focusing on pattern reuse specifically. There are also some relationships with Package-Based Description Logics [2], where three views to entities are provided: public, protected and private. The authors of Package-Based Description Logics state that this change introduces parameterism in ontologies, although its expressiveness is not explored.

The current OWL syntax and tool support is built completely around annotation properties. Protégé plugins are based on the OWLAPI library, and they are currently in an early beta-testing state of development. In future versions of the syntax, we plan to keep annotation properties to indicate if an element is instantiable in a pattern, and add support for pattern imports to the modules mechanism in OWL. This could be done by extending `owl:imports` with a nested tag `owl:patterninstance`, which in turn would have `from_uri` and `to_uri` attributes for the rebasing. Also, nested `owl:mapping` elements with attributes `from_name` and `to_name` could be used for the element/value assignations. The meaning of these attributes, their domain and range, would be according to what was established in the Pattern Instance definition ontology, in section 4.

Regarding the expressiveness of the pattern definition language, it currently allows roles, concepts and individuals to be treated as parameters. This may be sufficient in most cases, but it might be interesting to have other parametrisable parts of an ontology. For instance, numbers in cardinality restrictions could also be parameters. Note that the introduction of parametricity at this level would likely make the pattern definition no longer be an ontology.

Even though some content ODPs can easily be represented with our definition of pattern, it is not clear that all ODPs can. Future research should also focus on trying to find Ontology Design patterns that cannot be represented with our definition, in order to identify other elements that can also be parameterised.

References

1. Franz Baader and Werner Nutt. Basic description logics. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *Description Logic Handbook*, pages 43–95. Cambridge University Press, 2003.
2. Jie Bao, Doina Caragea, and Vasant Honavar. On the semantics of linking and importing in modular ontologies. In *International Semantic Web Conference*, pages 72–86, 2006.
3. Jean H. Gallier. *Logic for computer science: foundations of automatic theorem proving*. Harper & Row Publishers, Inc., New York, NY, USA, 1985.
4. Aldo Gangemi. Collection design pattern. http://ontologydesignpatterns.org/wiki/Submissions:Collection.
5. Luigi Iannone, Alan Rector, and Robert Stevens. Embedding knowledge patterns into OWL. In *6th Annual European Semantic Web Conference (ESWC2009)*, pages 218–232, June 2009.
6. Iván Pérez. Ontology engineering protege plugins, 2009. IMDEA Software. http://sharesource.org/project/ontoengineeringprotegepluglins/.
7. Iván Pérez. Pattern definition ontology, 2009. IMDEA Software. http://babel.ls.fi.upm.es/~iperez/pattern-ontologies/patterndefinition.
8. Iván Pérez. Pattern instance ontology, 2009. IMDEA Software. http://babel.ls.fi.upm.es/~iperez/pattern-ontologies/patterninstance.
9. Iván Pérez. Pattern Instance definition ontology, 2009. IMDEA Software. http://babel.ls.fi.upm.es/~iperez/pattern-ontologies/patterninstancedefinition.
10. Valentina Presutti. Classification design pattern. http://ontologydesignpatterns.org/wiki/Submissions:Classification.
11. Valentina Presutti. Collection Entity design pattern. http://ontologydesignpatterns.org/wiki/Submissions:CollectionEntity.
12. Valentina Presutti. Componency design pattern. http://ontologydesignpatterns.org/wiki/Submissions:Componency.

Preliminary Results of Logical Ontology Pattern Detection Using SPARQL and Lexical Heuristics

Ondřej Šváb-Zamazal[1,2], François Scharffe[2], and Vojtěch Svátek[1]

[1]University of Economics, Prague, {ondrej.zamazal,svatek}@vse.cz
[2] INRIA & LIG, Montbonnot, France, {francois.scharffe}@inria.fr

Abstract. Ontology design patterns were proposed in order to assist the ontology engineering task, providing models of specific construction representing a particular form of knowledge. Various kinds of patterns have since been introduced and classes of patterns identified. Detecting these patterns in existing ontologies is needed in various scenarios, for example the detection of the the two parts of an alignment pattern in an ontology matching scenario, or the detection of an anti-pattern in an optimization scenario.
In this paper we present a novel method for the detection of logical patterns in ontologies. This method is based on both SPARQL, as the underlying language for retrieving patterns, and a lexical heuristic constraining the query. It extends our previous works on ontology patterns modeling and detection. We describe an algorithm computing a token-based similarity measure used as the lexical heuristic. We conduct an experiment on a large number of Web ontologies, obtaining interesting measures on the usage frequency of three selected patterns.

1 Introduction

Ontology Patterns turn out to be an important instrument in many diverse applications on the Semantic Web. This is reflected by many different *Ontology Design Pattern* types such as *Logical Patterns*, *Content Patterns*, *Refactoring Patterns*, *Transformation Patterns* or *Alignment Patterns*[1]. Many applications of ontology patterns need to detect patterns at first. In this paper we present preliminary experiments with logical ontology pattern detection using SPARQL and additional lexical heuristics. This work extends the work presented in [7, 8] in terms of detection experiments over real ontologies. Our particular motivation for ontology pattern detection is *ontology transformation* where detection is the first step. In [8] we argue that ontology transformation is useful for many different Semantic Web use-cases such as ontology matching and ontology re-engineering. The remainder of this paper is organized as follows: in the next section we describe three ontology patterns and the way to detect them. Then in Section 3 we present preliminary results of a large scale detection experiment along with illustrative examples of these ontology patterns. We wrap-up the paper with conclusions and future work.

[1] `http://ontologydesignpatterns.org/wiki/OPTypes`

2 Patterns and their Detection

All of the logical ontology patterns introduced in this section have been presented before [8]. We introduce here preliminary results of their detection over a collection of real ontologies. The detection of these patterns has two aspects: structural and naming ones. Our method first detect the structural aspect using the SPARQL language[2]. We currently use the SPARQL query engine from the Jena framework[3]. SPARQL queries corresponding to each detected pattern are detailed in sections below. Then, the method applies the lexical heuristic computed by Algorithm 1.

Algorithm 1 *calculateAverageTokenBasedSimilarityMeasure*

```
$MainEntity \Leftarrow$ lemmatized tokens of main entity
$Entities \Leftarrow$ lemmatized tokens of entities
$c \Leftarrow 0$
$i \Leftarrow 0$
for all $u \in Entities$ do
  if $|u \cap MainEntity| \neq \emptyset$ then
    $i \Leftarrow i + 1$
  end if
end for
$c \Leftarrow i/|Entities|$
return $c$
```

Algorihtm 1 computes an average token-based similarity measure c. The particular instantiation of $MainEntity$ and $Entities$ depends on the ontology pattern, see below. This algorithm works on names of entities (a fragment of the entity URI) which are tokenised (See [6]) and lemmatized.[4] Lemmatization can potentially increase the recall of the detection process. The lexical heuristics constraint is fulfilled when c exceeds a certain threshold which is dependent on particular ontology pattern. The motivation of this computation is based on an assumption that entities involved in patterns share tokens. More entities share the same token, the higher probability of occurrence of a pattern. We detail below three patterns that were detected in the experiment described in Section 3.

2.1 Attribute Value Restriction

The AVR pattern has been originally introduced in [4] as a constituent part of an *alignment pattern*, a pattern of correspondence between entities in two ontologies. Basically, it is a class the instances of which are restricted with some

[2] `http://www.w3.org/TR/rdf-sparql-query/`
[3] http://jena.sourceforge.net/
[4] We use the Stanford POS tagger `http://nlp.stanford.edu/software/tagger.shtml`.

attribute value. The SPARQL query for detection of this ontology pattern is the following:

```
SELECT ?c1 ?c2 ?c3
WHERE {
?c1 rdfs:subClassOf _:b.
_:b owl:onProperty ?c2.
_:b owl:hasValue ?c3.
?c2 rdf:type owl:ObjectProperty.
FILTER (!isBlank(?c1)) }
```

In this query we express a value restriction applied on a named class. Furthermore restricting properties must be of the type 'ObjectProperty' in order to have individuals (eg. 'Sweet') as values and not data types (eg. String). Currently we do not consider the naming aspect for this pattern.

2.2 Specified Values

We first considered the SV pattern in [7], but it had been originally presented in a document from the SWBPD group[5]. This ontology pattern deals with 'value partitions' representing specified collection of values expressing 'qualities', 'attributes', or 'features'. An example is given in the next section 3.

There are mainly two ways for capturing this pattern which are reflected by two different SPARQL queries. Either individuals where qualities are instances can be used for the detection:

```
SELECT distinct ?p ?a1 ?a2
WHERE {
?a1 rdf:type ?p.
?a2 rdf:type ?p.
?a1 owl:differentFrom ?a2 }
```

Or subclasses where qualities are classes partitioning a 'feature' can be used:

```
SELECT distinct ?p ?c1 ?c2
WHERE {
?c1 rdfs:subClassOf ?p.
?c2 rdfs:subClassOf ?p.
?c1 owl:disjointWith ?c2
FILTER (
!isBlank(?c1) && !isBlank(?c2) && !isBlank(?p))}
```

We are interested in mutually disjoint named classes (siblings) and we use non-transitive semantics (ie. direct) of 'subClassOf' relation here. Otherwise we would get 'specified value' as many times as there are different superclasses for those siblings. Regarding the initialisation of variables from the Algorithm 1, the *MainEntity* is either a *?p* instance (for the first query) or class (for the second query). *Entities* are all other entities from the *SELECT* construct. The experimental setting for the threshold is 0.5.

[5] http://www.w3.org/TR/swbp-specified-values/

2.3 Reified N-ary Relations

We have already considered the N-ary pattern in [7]. It has also been an important topic of the SWBPD group [2], because there is no direct way how to express N-ary relations in OWL[6]. Basically, a N-ary relation is a relation connecting an individual to many individuals or values. For this pattern we adhere to a solution introduced in [2]: introducing a new class for a relation which is therefore reified. For examples in the next section 3 we will use the following syntax (property(domain,range)):

$relationX(X, Y); relationY1(Y, A); relationY2(Y, B)$

The structural aspect of this pattern is captured using the following SPARQL query:

```
SELECT ?relationX ?Y ?relationY1 ?relationY2 ?A ?B
WHERE {
?relationX rdfs:domain ?X.
?relationX rdfs:range ?Y.
?relationY1 rdfs:domain ?Y.
?relationY1 rdfs:range ?A.
?relationY2 rdfs:domain ?Y.
?relationY2 rdfs:range ?B
FILTER (?relationY1!=?relationY2)}
```

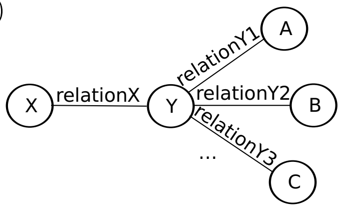


Fig. 1. N-ary relation

The intended structure of this reified N-ary relation pattern is depicted in the Figure on the right. In order to increase the precision of the detection we also apply lexical heuristics introduced above in Algorithm 1: variable $MainEntity$ is initialised with the value $?relationX$. $Entities$ are all other entities from the $SELECT$ construct. The experimental setting for the threshold is 0.4.

3 Experiment

In order to acquire a high number of ontologies, we applied the Watson tool[7] via its API. We searched ontologies imposing conjuction of the following constraints: OWL as the representation language, at least 10 classes, and at least 5 properties. Alltogether we collected 490 ontologies. However, many ontologies have not been accessible at the time of querying or there were some parser problems. Futhermore we only include ontologies having less than 300 entities. All in all our collection has 273 ontologies.

Table 1 presents overall numbers of ontologies where certain amount of ontology patterns were detected.

We can see that patterns were only detected in a small portion of ontologies from the collection. In four ontologies, the AVR pattern was detected more than 10 times. It reflects the fact that some designers tend to extensively use this

[6] It also holds for OWL 2. The notion of N-ary datatype was not introduced there, except for syntactic constructs allowing further extensions, see `http://www.w3.org/TR/2009/WD-owl2-new-features-20090611/#F11:_N-ary_Datatypes`

[7] `http://watson.kmi.open.ac.uk/WS_and_API.html`

	$\geq 10\times$	$(9-4)\times$	$3\times$	$2\times$	$1\times$	all
AVR pattern	$4\times$	–	$2\times$	$1\times$	$1\times$	$8\times$
SV pattern	–	$4\times$	–	$2\times$	$9\times$	$15\times$
N-ary pattern	–	$5\times$	$4\times$	$16\times$	$25\times$	$50\times$

Table 1. Frequency table of ontologies wrt. number of ontology patterns detected.

pattern. Other two ontology patterns were not so frequent in one ontology (the SV pattern was detected maximally 8 times and the N-ary pattern was detected maximally six times). On the other hand the most frequent pattern regarding a number of ontologies was the N-ary pattern. This goes against an intution that this pattern is quite rare. It can be explained with a low precision detection of this pattern, see below.

In order to obtain raw preliminary precision estimation for the ontology pattern detection we analysed one randomly chosen detected pattern instance from 10 ontologies (in the case of the AVR pattern from 8 ontologies). Although we tried to apply ontology transformation perspective for manual evaluation, we could not fully avoid coarse-grained and subjective evaluation due to soft boundaries between ontology patterns and sometimes unexpected conceptualisations in some Web ontologies.

The overall precision for the AVR pattern is 0.6, for the SV pattern 0.7, and for the N-ary pattern 0.3. For better insight we will look at two examples (one positive and one negative) for each of these ontology patterns.

AVR pattern This ontology pattern was found many times in a wine ontology[8] with high precision. One positive example is the following:

$\ni hasColor.\{White\} \sqsupseteq Chardonnay$

Chardonnay wine is restricted on these instances having value 'White' for the property hasColor. On the other hand, one negative example is the following[9]:

$\ni \_2.\{coordinate\_0\} \sqsupseteq North$

In this ontology each point of the compass (eg. North) is described using three different relations (_2 is one of them) having coordinates. This cannot be interpreted as an attribute value restriction pattern.

SV pattern The following[10] is one example which we evaluated as positive (a shared token is 'Molecule', $c = 1.0$):

$Molecule \sqsupseteq AnorganicMolecule; Molecule \sqsupseteq OrganicMolecule$

This can be interpreted as a collection of different kinds of molecules which is a complete partitioning. Furthermore disjointness is ensured by a query. On the other hand in another ontology[11] a negative example was detected:

[8] `http://www.w3.org/TR/2003/CR-owl-guide-20030818/wine`
[9] `http://sweet.jpl.nasa.gov/ontology/space.owl`
[10] `http://www.meteck.org/PilotPollution1.owl`
[11] `http://www.cip.ifi.lmu.de/~oezden/MastersThesisWeb/STCONCEPTS.owl`

$TimePeriods \sqsupseteq SocioculturalTimePeriods; TimePeriods \sqsupseteq CalendarDatePeriods$

In this case it can hardly be a complete partitioning, however this is always dependent on domain of discourse. It points out that ontology pattern detection should generally be a semi-automatic process.

N-ary pattern This pattern has the lowest precision. Due to the usage of a relaxed structural condition there are a lot of negative cases. Even if the lexical heuristics constraint improves this low precision, there is still ample space for improvement.

In the *PML* ontology[12] the following positive example was detected:

$hasPrettyNameMapping(InferenceStep, PrettyNameMapping)$
$hasPrettyName(PrettyNameMapping, string)$

$hasReplacee(PrettyNameMapping, string)$

This is the example of N-ary relation where the reified property 'PrettyNameMapping' ('?Y') captures additional attributes ('hasReplacee') describing the relation ('hasPrettyNameMapping'). $c = 0.5$ where shared tokens were 'Pretty' resp. 'has'.

On the other hand in the *earthrealm*[13] a negative example was detected:

$hasUpperBoundary(EarthRealm, LayerBoundary)$
$isUpperBoundaryOf(LayerBoundary, EarthRealm)$

$isLowerBoundaryOf(LayerBoundary, EarthRealm)$

In this case 'LayerBoundary' was detected as a reified N-ary relation ('?relationX') connecting different aspects ('?relationY1', '?A' and '?relationY2','?B') of the same relation; $c = 0.75$ where a shared token was 'Boundary'. But if we look at the classes bound with '?X', '?A' and '?B', we can see that there is an implicit 'inverseOf' relation between 'hasUpperBoundary' and 'isUpperBoundaryOf' resp. with 'isLowerBoundaryOf'. This 'inverseOf' relation cannot be directly found in this ontology. However we could assume this and therefore we could increase the precision considering this specific case in the future work.

Another recurrent negative example is the following[14]:

$concludedby(perdurant, perdurant)$
$startedby(perdurant, perdurant)$

$concludedby(perdurant, perdurant)$

This is a chain of properties connected with the same class in domain/range; $c = 0.66$ where a shared token is 'by'. Thus, at the first sight a detection could be improved with considering this negative example ('?X'='?Y'='?A'='?B'). On the other hand we can also find a counter-example considering true semantics of domain and range in OWL, ie. restrictions are superclasses of possible individuals. As usual, applying this condition we could filter out some current negative examples (getting higher precision), on the other side we could miss other positive examples (getting lower recall). This choice is application-dependant.

4 Related Work

In [3] the authors generally consider using SPARQL expressions for extracting Content Ontology Design Patterns from an existing reference ontology. It is followed by a manual

[12] `http://inferenceweb.stanford.edu/2004/07/iw.owl`
[13] `http://sweet.jpl.nasa.gov/sweet/earthrealm.owl`
[14] `http://neuroscientific.net/bio-zen.owl`

selection of particular useful axioms towards creating new Content Ontology Design Pattern. The Ontology Pre-Processing Language (OPPL) is specialized on pattern-based manipulation with ontologies. It can be used for ontology pattern detection, however there is no such a lexical support which our detection needs. In [1] the authors envision employing OPPL for detecting recurring patterns in ontologies and materialize them as new patterns. This is also one of our long-term effort.

By now, we use the SPARQL language for detecting structural aspect of ontology patterns. But SPARQL is a query language for RDF. Considering ontology patterns as DL-like conceptualisations, it leads to a necessity of expressing DL-like concepts in RDF representations which is rarely 1:1. This can be overcome by using some OWL-DL aware query language, eg. SPARQL-DL [5]. However for now this language does not support some specific DL constructs e.g. *restriction* and it is not fully implemented yet.

5 Conclusions and Future Work

In this paper we presented preliminary results of logical ontology pattern detection for which we use SPARQL and lexical heuristics. We conducted an experiment on a large number of Web ontologies. We manually evaluated 28 ontology pattern instances so as to roughly estimate precision. The performance of this detection must be further improved. Besides future work depicted in Section 3, we have to further work on more sophisticated lexical heuristics constraint. We could also employ head noun detection [6]. Further, we should also try to perform our queries using SPARQL-DL as OWL-DL aware language. We work on a specific FILTER extension (in connection with head noun detection) for the SPARQL language which could include the naming aspect already at the level of the query language.

Acknowledgement The work has been partially supported by the IGA VSE grant no. 20/08 “Evaluation and matching ontologies via patterns”.

References

1. A. R. Luigi Iannone and R. Stevens. Embedding Knowledge Patterns into OWL. In *Proceedings of the 6th European Semantic Web Conference*, 2009.
2. N. Noy and A. Rector. Defining n-ary relations on the semantic web, Apr. 2006.
3. V. Presutti and A. Gangemi. Content ontology design patterns as practical building blocks for web ontologies. In *Proceedings of ER2008*. Barcelona, Spain, 2008.
4. F. Scharffe. *Correspondence Patterns Representation.* PhD thesis, University of Innsbruck, 2009.
5. E. Sirin and B. Parsia. SPARQL-DL: SPARQL Query for OWL-DL. In *OWLED2007*, 2007.
6. O. Šváb Zamazal and V. Svátek. Analysing Ontological Structures through Name Pattern Tracking. In *Proceedings of the 16th International Conference on Knowledge Engineering and Knowledge Management*, 2008.
7. O. Šváb-Zamazal and V. Svátek. Towards Ontology Matching via Pattern-Based Detection of Semantic Structures in OWL Ontologies. In *Proceedings of the Znalosti Czecho-Slovak Knowledge Technology conference*, 2009.

8. O. Šváb-Zamazal, V. Svátek, and F. Scharffe. Pattern-based Ontology Transformation Service. In *Proceedings of the 1st International Conference on Knowledge Engineering and Ontology Development*, 2009.

Ontology Construction for Web Services

Aviv Segev[1] and Quan Z. Sheng[2]

[1] Department of Knowledge Service Engineering, KAIST, Daejeon 305-701, Korea
aviv@kaist.edu
[2] School of Computer Science, The University of Adelaide, SA 5005, Australia
qsheng@cs.adelaide.edu.au

Abstract. Ontologies have become the de-facto modeling tool of choice, employed in a variety of applications and prominently in the Semantic Web. Nevertheless, ontology construction remains a daunting task. Ontological bootstrapping, which aims at automatically generating concepts and their relations in a given domain, is a promising technique for ontology construction. Bootstrapping an ontology based on a set of predefined textual sources, such as Web services, must address the problem of multiple concepts that are largely unrelated. This paper exploits the advantage that Web services usually consist of both WSDL and free text descriptors. The WSDL descriptor is evaluated using two methods, namely Term Frequency/Inverse Document Frequency (TF/IDF) and Web context generation. We propose an ontology bootstrapping process that integrates the results of both methods and validates the concepts using the free text descriptors, thereby offering a more accurate definition of ontologies.

1 Introduction

Ontologies are used in an increasing range of applications, notably the Semantic Web, and essentially have become the preferred modeling tool. However, the design and maintenance of ontologies is a formidable process [1]. Ontology bootstrapping, which has recently emerged as an important technology for ontology construction, involves automatic identification of concepts relevant to a domain and relations between the concepts [2]. Previous work on ontology bootstrapping focused on either a limited domain or expanding an existing ontology [3]. In the field of Web services, registries such as the Universal Description, Discovery and Integration (UDDI) have been created to encourage interoperability and adoption of Web services. Unfortunately, UDDI registries have some major flaws [4]. In particular, UDDI registries either are publicly available and contain many obsolete entries or require registration which limits access. In either case, a registry only stores a limited description of the available services. Ontologies created for classifying and utilizing Web services can serve as an alternative solution. However, the increasing number of available Web services makes it difficult to classify Web services using a single domain ontology or a set of existing ontologies created for other purposes. Furthermore, the constant increase in the number of Web services requires continuous manual effort to evolve an ontology.

The Web service ontology bootstrapping process proposed in this paper is based on the advantage that a Web service can be separated into two types of descriptions: i) the Web Service Description Language (WSDL) describing "how" the service should be used and ii) a free text description of the Web service describing "what" the service does. This advantage allows bootstrapping the ontology based on WSDL and verifying the process based on the Web service free text descriptor.

The ontology bootstrapping process is based on analyzing a Web service using three different methods, where each method represents a different perspective of viewing the Web service. In particular, the first method analyzes the Web service from an internal point of view, i.e., what concept in the text best describes the document content. The second method describes the document from an external point of view, i.e., what most common concept represents the answers to the Web search queries based on the WSDL content. Finally, the third method is used to resolve inconsistencies with the current ontology. An ontology evolution is performed when all three analysis methods agree on the identification of a new concept or a relation change between the ontology concepts. The relation between two concepts is defined using the descriptors related to both concepts. Our approach facilitates automatic building of an ontology that could assist in expanding, classifying, and retrieving relevant services, without the prior training required by previously developed approaches.

2 Related Work

The field of automatic annotation of syntactic Web services contains several works relevant to our research. [5] presents a combined approach toward automatic semantic annotation of Web services. The approach relies on several matchers (e.g., string matcher, structural matcher, and synonym finder), which are combined using a simple aggregation function. Machine learning is used in a tool called Assam [6], which uses existing annotation of semantic Web services to improve new annotations. [7] suggests a context-based semantic approach to the problem of matching and ranking Web services for possible service composition. Unfortunately, all these approaches require clear and formal semantic annotations to ontologies.

Ontology evolution has been researched on domain specific Web sites [8]. Noy and Klein [1] defined a set of ontology-change operations and their effects on instance data used during the ontology evolution process. Unlike prior work which was heavily based on existing ontology or domain specific, our work evolves an ontology for Web services "from scratch". A survey on the state of the art Web service repositories [9] suggests that analyzing the Web service textual description in addition to the WSDL description can be more useful than analyzing each descriptor separately. The survey mentions the limitation of existing ontology evolution techniques which yield low recall. Our solution overcomes the low recall using Web context recognition.

3 The Bootstrapping Ontology Model

The bootstrapping ontology model proposed in this paper is based on the continuous analysis of WSDL documents and employs an ontology model based on concepts and relationships [10]. The innovation of the proposed bootstrapping model is the combination of the use of two different extraction methods, TF/IDF and Web based, and the verification of the results using a third method analyzing the external service descriptor. We used these three methods to demonstrate the feasibility of our model. Other more complex methods, from the field of Machine Learning (ML) and Information Retrieval (IR), can also be used to implement the model. However, the straightforward use of the methods emphasizes that many methods can be "plugged in" and that the results are attributed to the model's process of combination and verification.

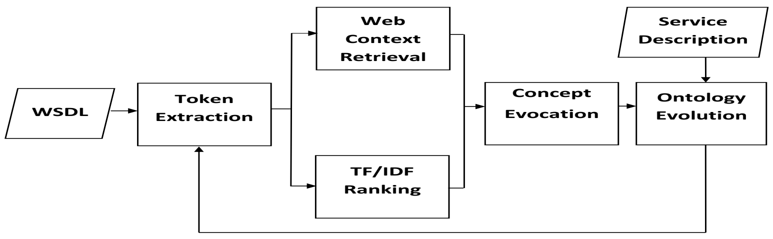


Fig. 1. Web Service Ontology Bootstrapping Process

The overall bootstrapping ontology process is described in Figure 1. There are four main steps in the process. The *token extraction* step extracts tokens representing relevant information from a WSDL document. The second step analyzes in parallel the extracted WSDL tokens using two methods. In particular, *TF/IDF* analyzes the most common terms appearing in each Web service document and appearing less frequently in other documents. *Web context* extraction uses the sets of tokens as a query to a search engine, clusters the results according to descriptors, and classifies which set of descriptors identifies the context of the Web service. The *concept evocation* step identifies the descriptors appearing in both the TF/IDF method and the Web context method. These descriptors identify possible concept names which could be utilized by the ontology evolution. The context descriptors also assist in the convergence process of the relations between concepts. Finally, the *ontology evolution* step expands the ontology as required according to the newly identified concepts and modifies the relations between them. The external Web service textual descriptor serves as a moderator if there is a conflict between the current ontology and a new concept. The relations are defined as an ongoing process according to the most common context descriptors between the concepts. After the ontology evolution, the process continues with the next WSDL. It should be noted that the processing order of WSDL documents is arbitrary.

3.1 Token Extraction

The analysis starts with token extraction, representing each service, S, using a set of tokens called *descriptors*. Each token is a textual term, extracted by simply parsing the underlying documentation of the service. The descriptor represents the WSDL document, formally put as $\mathcal{D}^{S}_{wsdl} = \{t_1, t_2, \ldots, t_n\}$, where t_i is a token. WSDL tokens require special handling, since meaningful tokens (such as names of parameters and operations) are usually composed of a sequence of words, with the first word lowercase, followed by first letter of other words capitalized (*e.g.*, `getInstitutionNameFromDomain`). Therefore, the descriptors are divided into separate tokens. Figure 2 depicts a WSDL document with the tokens bolded.

The extracted token list serves as a *baseline*. These tokens are extracted from the WSDL document of a Web service that determines whether an email address or domain name belongs to an academic institution. The service is used to illustrate the initial step in building the ontology. All elements classified as *name* are extracted, including tokens that might be less relevant.

3.2 TF/IDF Analysis

TF/IDF is a common mechanism in IR to generate a robust set of representative keywords from a corpus of documents. The method is applied here to the WSDL de-

```
<definitions name="AcademicVerifier"
targetNamespace="http://www.capeclear.com/AcademicVerifier.wsdl" ...>
<message name="isAcademicEmailAddress"><part name="emailAddress">
  <message name="getInstitutionNameFromDomain">
  <message name="getInstitutionNameFromDomainResponse">
  <message name="getInstitutionNameFromEmailAddress">
```

Fig. 2. Initial Processing Example of the Academic Verifier

scriptors. By building an independent corpus for each document, irrelevant terms are more distinct and can be thrown away with a higher confidence. To formally define TF/IDF, we start by defining $freq(t_i, \mathcal{D}_i)$ as the number of occurrences of the token t_i within the document descriptor $\mathcal{D}_i$. We define the term frequency of each token t_i as: $tf(t_i) = \frac{freq(t_i, \mathcal{D}_i)}{|\mathcal{D}_i|}$. We define $\mathcal{D}_{wsdl}$ to be the corpus of WSDL descriptors. The inverse document frequency is calculated as the ratio between the total number of documents and the number of documents which contain the term: $idf(t_i) = \log \frac{|\mathcal{D}|}{|\{\mathcal{D}_i : t_i \in \mathcal{D}_i\}|}$. Here, $\mathcal{D}$ is defined generically, and its actual instantiation is chosen according to the origin of the descriptor. The TF/IDF weight of a token, annotated as $w(t_i)$, is calculated as: $w(t_i) = tf(t_i) \times idf^2(t_i)$.

The token weight is used to induce ranking over the descriptor's tokens. We define the ranking using a precedence relation $\preceq_{tf/idf}$, which is a partial order over $\mathcal{D}$, such that $t_l \preceq_{tf/idf} t_k$ if $w(t_l) < w(t_k)$. The ranking is used to filter the tokens according to a threshold which filters out words with a frequency count higher than the second standard deviation from the average frequency. Figure 3 on the left circle of every concept presents the list of tokens which received a higher weight than the threshold. Several tokens which appeared in the baseline list (see Figure 2) were removed due to the filtering process. For instance, words such as "response" and "get" received below-the-threshold TF/IDF weight, due to their high frequency.

3.3 Context Extraction

We define a context descriptor c_i from domain $\mathcal{DOM}$ as an index term used to identify a record of information, which in our case is a Web service. A weight $w_i \in \Re$ identifies the importance of descriptor c_i in relation to the Web service. For example, we can have a descriptor c_1 = *Academic* and w_1 = 42. A *descriptor set* $\{\langle c_i, w_i\rangle\}_i$ is defined by a set of pairs, descriptors and weights. Each descriptor can define a different point of view of the concept. The descriptor set defines all the different perspectives and their relevant weights, which identify the importance of each perspective.

By collecting all the different view points delineated by the different descriptors we obtain the *context*. A *context* $\mathcal{C} = \left\{\{\langle c_{ij}, w_{ij}\rangle\}_i\right\}_j$ is a set of finite sets of descriptors, where i represents each context descriptor and j represents the index of each set. For example, a context $\mathcal{C}$ may be a set of words (hence $\mathcal{DOM}$ is a set of all possible character combinations) defining a Web service and the weights can represent the relevance of a descriptor to the Web service. In classic IR, $\langle c_{ij}, w_{ij}\rangle$ may represent the fact that the word c_{ij} is repeated w_{ij} times in the Web service descriptor document.

The context recognition algorithm was adapted from [11], which can be formally defined as: Let $\mathcal{D} = \{\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_m\}$ be a set of textual propositions representing a Web service, where for all $\mathcal{P}_i$ there exists a collection of descriptor sets forming the context $\mathcal{C}_i = \{\langle c_{i1}, w_{i1}\rangle, ..., \langle c_{in}, w_{in}\rangle\}$ so that $ist(\mathcal{C}_i, \mathcal{P}_i)$ is satisfied. McCarthy [12] defines a relation $ist(\mathcal{C}, \mathcal{P})$, asserting that a proposition $\mathcal{P}$ is true in a context $\mathcal{C}$. In our case, the adapted algorithm uses the corpus of WSDL descriptors, $\mathcal{D}^{wsdl}$, as propositions $\mathcal{P}_i$ and

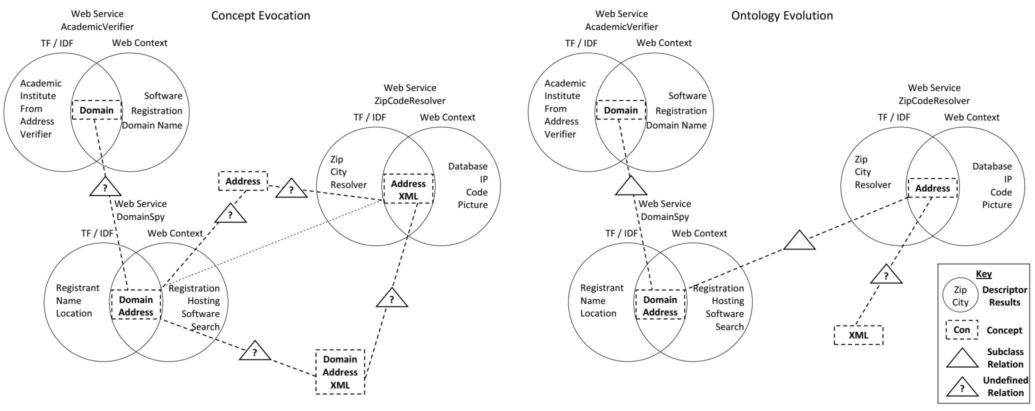


Fig. 3. Example of Web Service Ontology Bootstrapping

the contexts describing the WSDL as descriptors c_{ij} with their associated weight w_{ij}. The context recognition algorithm identifies the outer context $ist(\mathcal{C}, \bigcap_{i=1}^{m} ist(\mathcal{C}_i, \mathcal{P}_i))$.

The context recognition algorithm consists of three main phases: 1) selecting contexts for each text, 2) ranking the contexts, and 3) declaring the current contexts. The result of the token extraction is a list of keywords obtained from the text. The selection of the current context is based on searching the Web for relevant documents according to these keywords and on clustering the results into possible contexts. The output of the ranking stage is the current context or a set of highest ranking contexts. The set of preliminary contexts that has the top number of references, both in number of Web pages and in number of appearances in all the texts, is declared to be the current context and the weight is defined by integrating the value of references and appearances. The input to the algorithm is a stream of information in text format. Figure 3 shows the result of the Web context extraction in the right circle of each concept. The figure shows the context that includes only the highest ranking descriptors which pass the cutoff to be included in the context. For example, *Domain*, *Software*, *Registration*, and *Domain Name* are the context descriptors selected to describe the `AcademicVerifier` service.

3.4 Concept Evocation

Concept evocation identifies a possible concept definition which will be refined in the ontology evolution. The concept evocation is based on context intersection. An ontology concept is defined by the descriptors which appear in the intersection of both the Web context results and the TF/IDF results. We defined one descriptor set from the TF/IDF results, tf/idf_{result}, based on extracted tokens from the WSDL text. The *context*, $\mathcal{C}$, is initially defined as a descriptor set extracted from the Web representing the same document. As a result, the ontology concept is represented by a set of descriptors, c_i, which belong to both sets: $Concept = \{c_1, ..., c_n | c_i \in tf/idf_{result} \cap c_i \in \mathcal{C}\}$.

Figure 3 shows an example of the concept identified by the intersection. For the `AcademicVerifier` Web service, the concept is based on the intersection of both descriptor sets is identified as *Domain*. The concept can consist of more than one descriptor (e.g., `DomainSpy` Web service is identified by the descriptors *Domain* and *Address*). Concepts can be evoked as a result of partial overlapping concepts. This example can be seen by *Address* and the set of *Domain*, *Address*, and *XML*.

A context can consist of multiple descriptor sets and can be viewed as a meta-representation of the Web service. The added value of having such a meta-representation is that each descriptor set can belong to several ontology concepts simultaneously. For example, a descriptor set $\{\langle\ Registration, 23\rangle\}$ can be shared by multiple ontology concepts (Figure 3) that have interest in domain registration. The different concepts can be related by verifying whether a specific domain exists, domain spying, etc., although the descriptor may have differing relevance to the concept and hence different weights are assigned to it. Such overlap of contexts in ontology concepts affects the task of Web service ontology bootstrapping. The appropriate interpretation of a Web service context that is part of several ontology concepts is that the service is relevant to all such concepts. This leads to the possibility of the same service belonging to multiple concepts based on different perspectives of the service use.

The concept relations can be deduced based on convergence of the context descriptors. The ontology concept is described by a set of contexts, each of which includes descriptors. Each new Web service that relates to the concept adds new context descriptor sets. As a result, the most common context descriptors which relate to more than one concept can change after every iteration. The sets of descriptors of each concept are defined by the union of the descriptors of both the Web context and the TF/IDF results. The *context* is expanded to include the descriptors identified by the Web context, the TF/IDF, and the concept descriptors: $Context_{expanded} = \{c_1, ..., c_n | c_i \in tf/idf_{result} \cup c_i \in \mathcal{C}\}$. For example, in Figure 3, the context of service `AcademicVerifier` includes the descriptors: *Software*, *Registration*, *Domain Name*, *Domain*, *Academic*, *Institute*, *From*, *Address*, and *Verifier*.

The relation between two concepts, Con_i and Con_j, can be defined as the context descriptors common to both concepts, for which weight w_k is greater than a cut off value of a: $Re(Con_i, Con_j) = \{c_k | c_k \in Con_i \cap Con_j, w_k > a\}$. However, since multiple context descriptors can belong to two concepts, the value of a for the relevant descriptors needs to be predetermined. A possible cutoff can be defined by TF/IDF, Web Context, or both. Alternatively, the cutoff can be defined by a minimum number or percent of Web services belonging to both concepts based on shared context descriptors. The relation between the two concepts *Domain* and *Domain Address* in Figure 3 can be based on *Domain* or *Registration*. The example takes a minimum number of appearances in a document as the cutoff of both the TF/IDF and Web Context methods.

3.5 Ontology Evolution

The ontology evolution consists of four steps including: 1) building new concepts, 2) determining the concept relations, 3) identifying relations types, and 4) re-setting the process for the next WSDL document. Building a new concept is based on refining the possible identified concepts. The evocation of a concept in the previous step does not guarantee that it should be integrated with the current ontology. Instead, the new possible concept should be analyzed in relation to the current ontology.

The descriptor is further validated using the textual service descriptor. The analysis is based on the advantage that a Web service can be separated into two descriptions: the WSDL description and a description of the Web service in free text. The WSDL descriptor is analyzed to extract the context descriptors and possible concepts as described previously. The second descriptor, $\mathcal{D}^{S}_{desc} = \{t_1, t_2, \ldots, t_n\}$, represents the text

```
1:  For each Web service
2:      Extract tokens from WSDL
3:      $TF/IDF_{result}$ = Apply TF/IDF algorithm to $\mathcal{D}_{wsdl}$
4:      $WebContext_{result}$ = Apply Web Context algorithm to $\mathcal{D}_{wsdl}$
5:      $PossibleCon_i = TF/IDF_{result} \cap WebContext_{result}$
6:      If( $PossibleCon_i \subseteq \mathcal{D}_{desc}$)
7:          $Con_i = TF/IDF_{result} \cap WebContext_{result}$
8:      $PossibleRel_i = TF/IDF_{result} \cup WebContext_{result}$
9:  For each concept pair $Con_i, Con_j$
10:     If( $Con_i \subseteq Con_j$ )
11:         $Con_i$ subclass $Con_j$
12:     Else
13:         $Re(Con_i, Con_j) = PossibleRel_i \cap PossibleRel_j$
```

Fig. 4. Ontology Bootstrapping Algorithm

description of the service supplied by the service developer in free text. These descriptions are relatively short and include a sentence or two to describe the Web service. The verification process includes matching the concept descriptors in simple string matching against all the descriptors of the textual service descriptor. We use a simple string-matching function, $match_{str}$, which returns 1 if two strings match and 0 otherwise.

Continuing the example in Figure 3, analysis of the `AcademicVerifier` service yields only one descriptor as a possible concept. The descriptor *Domain* was identified by both the TF/IDF and the Web Context results and matched with a textual descriptor. It is similar for the *Domain* and *Address* appearing in the `DomainSpy` service. However, for the `ZipCodeResolver` service both *Address* and *XML* are possible concepts but only *Address* passes the verification with the textual descriptor. As a result, the concept is split into two separate concepts and the `ZipCodeResolver` service descriptors are associated with both of them.

To evaluate the relation between concepts, we analyze the overlapping context descriptors between different concepts. In this case, we use descriptors which were included in the union of the descriptors extracted by both the TF/IDF and Web context methods. Precedence is given to descriptors which appear in both concept definitions over descriptors which appear in the context descriptors. In our example, the descriptors related to both *Domain* and *Domain Address* are: *Software*, *Registration*, *Domain*, *Name*, and *Address*. However, only the *Domain* descriptor belongs to both concepts and receives the priority to serve as the relation. The result is the relation which can be identified as a subclass, where *Domain Address* is a subclass of *Domain*.

The process of analyzing the relation between concepts is performed after the concepts are identified. The identification of a concept prior to the relation allows in the case of *Domain Address* and *Address* to again apply the subclass relation based on the similar concept descriptor. However, the relation of *Address* and *XML* concepts remains undefined at the current iteration of the process since it would include all the descriptors that relate to `ZipCodeResolver` service. The relation described in the example is based on descriptors which are the intersection of the concepts. Basing the relations on a minimum number of Web services belonging to both concepts will result in a less rigid classification of relations. The process is performed iteratively for each additional service which is related to the ontology. The iterations stop once all the services are analyzed. Alternatively, an ontology administrator can decide to suspend the ontology evolution at any given time.

To summarize, we give the ontology bootstrapping algorithm in Figure 4. The first step is extracting the tokens from the WSDL for each Web service (line 2). The next step is applying the TF/IDF and Web Context to extract the result of each algorithm (lines 3-4). The possible concept, $PossibleCon_i$, is based on the intersection of tokens of the results of both algorithms (line 5). If $PossibleCon_i$ tokens appear in the document descriptor, $\mathcal{D}_{desc}$, $PossibleCon_i$ is defined as concept, Con_i. The union of all token results is $PossibleRel_i$ for concept relation evaluation (lines 6-8). Each pair of concepts, Con_i and Con_j, is analyzed for whether the token descriptors are contained in one another. If yes, a subclass relation is defined. Otherwise the concept relation can be defined by the intersection of the possible relation descriptors, $PossibleRel_i$ and $PossibleRel_j$ (lines 9-13).

4 Conclusion

This paper proposes an approach for bootstrapping an ontology based on Web service descriptions. The approach analyzes Web services from multiple perspectives and integrates the results. Web services usually consist of both WSDL and free text descriptors. This allows bootstrapping the ontology based on WSDL and verifying the process based on the Web service free text descriptor. The approach enables the automatic construction of an ontology without the prior training required by previously developed methods. As a result, ontology construction and maintenance efforts can be substantially reduced. Our ongoing work includes further performance study of the proposed ontology bootstrapping approach. We plan to apply the approach in other domains in order to examine the automatic verification of the results.

References

1. Noy, N.F., Klein, M.: Ontology Evolution: Not the Same as Schema Evolution. Knowledge and Information Systems **6**(4) (2004) 428–440
2. Ehrig, M., Staab, S., Sure, Y.: Bootstrapping Ontology Alignment Methods with APFEL. In: Proc. of 4th Intl. Semantic Web Conference (ISWC'05), Galway, Ireland (2005)
3. Zhang, G., Troy, A., Bourgoin, K.: Bootstrapping Ontology Learning for Information Retrieval Using Formal Concept Analysis and Information Anchors. In: Proc. of 14th Intl. Conference on Conceptual Structures (ICCS'06), Aalborg University, Denmark (2006)
4. Platzer, C., Dustdar, S.: A Vector Space Search Engine for Web Services. In: Proc. of the 3rd European Conference on Web Services (ECOWS'05), Växjö, Sweden (2005)
5. Patil, A., Oundhakar, S., Sheth, A., Verma, K.: METEOR-S Web Service Annotation Framework. In: Proc. of the 13th Intl. Conference on World Wide Web. (2004)
6. Heß, A., Johnston, E., Kushmerick, N.: ASSAM: A Tool for Semi-automatically Annotating Semantic Web Services. In: Proc. of Intl. Semantic Web Conference (ISWC'04). (2004)
7. Segev, A., Toch, E.: Context-Based Matching and Ranking of Web Services for Composition. IEEE Transactions on Services Computing **2**(3) (2009) 210–222
8. Davulcu, H., Vadrevu, S., Nagarajan, S., Ramakrishnan, I.: OntoMiner: Bootstrapping and Populating Ontologies From Domain Specific Web Sites. IEEE Intelligent Systems **18**(5) (2003) 24–33
9. Sabou, M., Pan, J.: Towards Semantically Enhanced Web Service Repositories. Web Semantics **5**(2) (2007) 142–150
10. Gruber, T.R.: A Translation Approach to Portable Ontologies. Knowledge Acquisition **5**(2) (1993) 199–220
11. Segev, A., Leshno, M., Zviran, M.: Context Recognition Using Internet as a Knowledge Base. Journal of Intelligent Information Systems **29**(3) (2007) 305–327
12. McCarthy, J.: Notes on Formalizing Context. In: Proc. of the 13th Intl. Joint Conference on Artificial Intelligence (IJCAI'93), Chambéry, France (1993)

Ontology Analysis Based on Ontology Design Patterns

María Poveda, Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez
Ontology Engineering Group. Departamento de Inteligencia Artificial.
Facultad de Informática, Universidad Politécnica de Madrid.
Campus de Montegancedo s/n.
28660 Boadilla del Monte. Madrid. Spain
mpoveda@delicias.fi.upm.es, {mcsuarez, asun}@fi.upm.es

Abstract. The so-called Ontology Design Patterns (ODPs), which have been defined as solutions to ontological design problems, are of great help to developers when modelling ontologies since these patterns provide a development guide and improve the quality of the resulting ontologies. However, it has been demonstrated that, in many cases, developers encounter difficulties when they have to reuse ontology design patterns and include errors in the modelling. Thus, to avoid errors in ontology modelling, this paper proposes classifying errors into two types: (1) errors related to existing ODPs, called *anti-patterns*, and (2) errors not related to existing ODPs, called *worst practices.* This classification is the result of analysing a set of ontologies which come from an academic experiment. In addition, the paper presents a general classification of the worst practices found and a set of worst practice examples. Finally, the paper shows an example of how the aforementioned worst practices could be related among them.

Keywords: patterns, anti-patterns, worst practices, ontology.

1 Introduction

Modelling ontologies has become one of the main topics of research within ontological engineering because of the difficulties it involves. In recent years, the emergence of Ontology Design Patterns (ODPs), which are defined as solutions to design problems [3], has supposed a great help to ontology developers.

Some experiments [2] carried out in ontology engineering have demonstrated that design patterns are perceived as an aid to modelling ontologies, a development guide, and a way to improve the quality of the resulting ontologies. However, it is well known that [2, 1], in some cases, ontology developers experience difficulties when reusing the patterns during modelling, and include errors in the modelling. Therefore, in order to understand and use correctly ODPs, we need a better support that prevents the emergence of modelling errors.

Thus, to avoid the appearance of errors in ontology development, we are working on the creation of a new set of methodological guides. These guides, based on the identification and classification of modelling errors, classify errors into two types: (1) errors related to ODPs, called *anti-patterns*; and (2) errors not related to ODPs, called *worst practices*. This classification of errors arises from performing an analysis, based on ODPs, of a set of ontologies. In this paper, such a classification is presented. In addition, we include a classification of the worst practices identified and a set of examples of such worst practices.

The remainder of the article is structured as follows: Section 2 presents the state of the art of patterns and anti-patterns in ontological engineering. Section 3 describes the experiment carried out in order to obtain the set of ontologies analyzed in this paper. Section 4 describes the analysis carried out in the aforementioned ontologies, which shows the presence of ODPs and describes anti-patterns and worst practices. Section 5 includes the classification of the worst practices found, a set of examples of worst practices, and an example of how the worst practices could be related among them. Finally, Section 6 includes the conclusions drawn and future lines of work.

2 State of the Art: Patterns and Anti-Patterns

In ontology engineering, the ontology design patterns can be considered as modelling solutions to problems widely known in the area. These solutions are based on good practices and solve modelling problems.

In the ODPs field, we can distinguish between logical patterns and conceptual patterns. With regard to logical patterns, the W3C work team, known as "*Semantic Web Best Practices and Deployment* (SWBPD)[1], has established that in order to provide support to developers and users of the Semantic Web, a set of good practices is required. To that purpose, this group proposes patterns that solve design problems in the OWL language, independently of the particular specification, which solve logical problems. Regarding conceptual patterns [3], the author proposes patterns (in OWL or any other logical language) that solve design problems for specific domains, which solve content problems.

In addition to the distinction mentioned above, in [5, 4] the authors classify the ODPs into content, structural, lexico-syntactic, reasoning, presentation and correspondence, and their subtypes.

The work described in [5, 4] is focused on content patterns and provides guidelines on how to apply content ODPs using import, specialization, composition and expansion functions. In the same work, the content ontology anti-pattern concept is defined as a design that is different from a content pattern in that the former codes the solution to a problem in a wrong way.

However, we have observed that none of the papers analyzed have carried out a thorough study on the use of any type of ODPs and their corresponding anti-patterns. We have also observed that there is no previous work focused on identifying and preventing to model errors not related to any existing ODP.

ODPs can be found in on-line libraries that include both the description and the OWL code associated to the patterns as, for example, "the *Ontology Design Pattern* Portal"[2], or they can be obtained from the work team "*Semantic Web Best Practices and Deployment".* Some other libraries [5, 6] do not provide the pattern code, but

[1] http://www.w3.org/2001/sw/BestPractices/

[2] http://ontologydesignpatterns.org/

they store descriptions of a great number of ODPs. These libraries, which follow a software engineering approach, use a template for describing the patterns.

3 Experiment Description

With the aim of identifying the types of errors normally made when developing ontologies, we have analyzed 11 ontologies that tackle different aspects (art, architecture, geography, tastes and likings, and community services) of the domain of Saint James's Way. These ontologies were developed by Master students as a practical assignment on the "Ontologies and the Semantic Web" subject at the *Universidad Politécnica de Madrid*, during 2007-2008.

Before performing the practical assignment, theoretical issues related to ontology development were explained to the students. Such theoretical lessons were combined with hands-on activities that allowed students to get the basis for building ontologies.

To develop the ontologies in the practical assignment, students worked in pairs bearing in mind the particular aspect within the Saint James's Way domain that they choose. Student groups carried out the following steps to develop their ontologies:

1. To find knowledge resources suitable for reuse in the ontology development.
2. To identify the life cycle model and the life cycle for the ontology to be developed, and to schedule the ontology development project.
3. To identify the requirements that the ontology to be developed should fulfil.
4. To model the knowledge to be represented in the ontology, integrating (if needed) the knowledge from the resources found in step 1.
5. To implement the ontology in OWL.

Students were free to choose as the ontology language among Spanish, English, or both, although most of them developed the ontologies in Spanish. Finally, we obtained 2 ontologies about art, both in Spanish; 2 ontologies about architecture, 1 in Spanish and 1 in English; 2 ontologies about geography, 1 in Spanish and 1 in English; 1 ontology about tastes and likings in Spanish; and 4 ontologies about community services, 3 in Spanish and 1 in English.

It is worth mentioning that ODPs issues were not explained to students during the master course. Thus, students did not have any notion about ODPs to be applied during the ontology development; therefore, it did not make sense to focus our study on the correctly or incorrectly reuse of ODPs. Instead, we focused our study on examining whether ODPs could be detected in the resulting ontologies. The approach adopted in the analysis was based on the manual search of ODPs in ontologies and on the manual identification of two types of modelling errors: (1) those related to ODPs, called *anti-patterns;* and (2) those not related to ODPs, called *worst practices*.

4 Analysis based on Ontology Design Patterns

As mentioned in Section 3, we focused our study on analysing the 11 ontologies in the Saint James' Way with the aim of searching whether ODPs could be detected in the resulting ontologies. The approach adopted in the analysis was based on the manual search of three possible scenarios in the 11 ontologies: (1) ODPs appearing in ontologies; (2) modelling errors related to ODPs; and (3) modelling errors not related to ODPs.

As a result of the analysis carried out, we have identified the possibilities shown in the tree of Fig. 1. In the first place, we can distinguish two main branches in the tree: the left one that represents whether a design equivalent to an ODP has been observed; and the right one that represents the case when no design equivalent to an ODP has been observed. Second, and finally, the tree classifies each of the scenarios found in some of the four leaf nodes and determines if such scenario matches a pattern identification, an anti-pattern identification, or a worst practice identification in the following way:

1. *Identification of an ontology design pattern.* (1st leaf from the left) We have identified some cases that have a correct design for the modelling problem; such a design matches an ODP of those in the available libraries.
2. *Identification of an anti-pattern*. (2nd and 3rd leaf) In some cases we have identified a design that matches an ODP but this design is not a suitable solution to the modelling problem. In some other cases we have not identified a design that could match an ODP in a suitable solution to the modelling problem; however, there is a suitable pattern that could have been applied if we had known the ODPs.
3. *Identification of a worst practice (WP)*. (4th leaf) In some other cases we have observed that there is an unsuitable solution to the modelling problem whereas there is not an ODP suitable for the modeling problem. Most of these cases have been treated as WPs; however, a few set of cases could have been solved by means of adapting or combining existing ODPs.

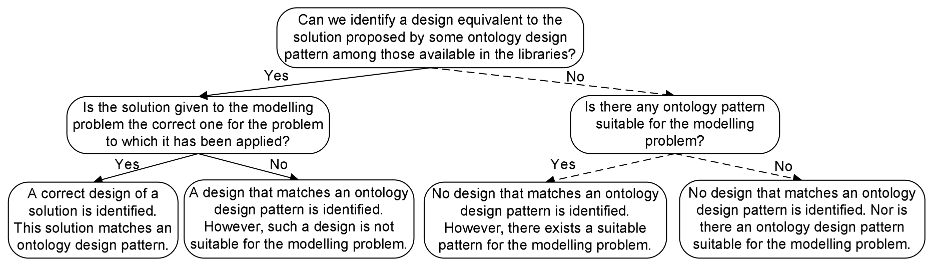

Can we identify a design equivalent to the solution proposed by some ontology design pattern among those available in the libraries?
Yes
No
Is the solution given to the modelling problem the correct one for the problem to which it has been applied?
Is there any ontology pattern suitable for the modelling problem?
Yes
No
Yes
No
A correct design of a solution is identified. This solution matches an ontology design pattern.
A design that matches an ontology design pattern is identified. However, such a design is not suitable for the modelling problem.
No design that matches an ontology design pattern is identified. However, there exists a suitable pattern for the modelling problem.
No design that matches an ontology design pattern is identified. Nor is there an ontology design pattern suitable for the modelling problem.

Fig. 1. Decision tree for classifying patterns, anti-patterns and worst practices.

During the analysis performed with the aforementioned 11 ontologies, we have found 208 cases in which we have identified a correct design of a solution that matches an ODP. We have also found 117 cases of anti-patterns, of which 83 correspond to situations in which the solution matches the modelling proposed in some ODP. However, such a solution is not suitable for the design problem intended to solve. Besides, we have found 34 cases in which we were unable to identify a design solution matching an ODP though a suitable ODP exists. Finally, we have come across a set of 231 WPs, that is, solutions unsuitable for the design problem in the domain of the ontologies studied for such design problem; such WPs do not have an ODP associated. These WPs are described in Section 5.

5 Classification and Examples of Worst Practices

As can be seen in Section 4, we have identified designs unsuitable for solving a modelling problem for which there is no an available ODP. In this paper, this type of design solutions is named worst practices (WPs).

We have classified the WPs identified in the analyzed ontologies bearing in mind the types of ODPs proposed in [5, 4]. In Fig. 2, each WP is associated to the type(s) of ODP to which a pattern created to avoid such a WP could belong. In Table 1 we provide a brief description of each WP classified.

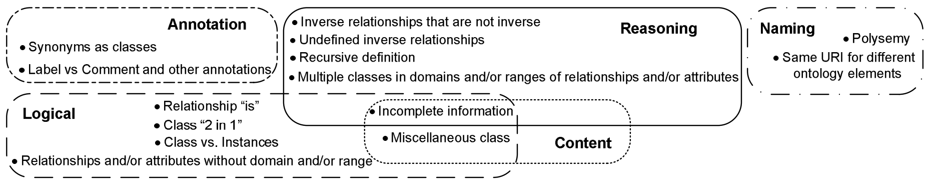

Annotation
• Synonyms as classes
• Label vs Comment and other annotations
• Inverse relationships that are not inverse
• Undefined inverse relationships
• Recursive definition
• Multiple classes in domains and/or ranges of relationships and/or attributes
Reasoning
Naming
• Polysemy
• Same URI for different ontology elements
Logical
• Relationship "is"
• Class "2 in 1"
• Class vs. Instances
• Relationships and/or attributes without domain and/or range
• Incomplete information
• Miscellaneous class
Content

Fig. 2. Classification of the worst practices identified.

As can be observed in the WPs classification appearing in Fig. 2, the types of WPs identified are the following: 1) **Annotation WP** that refers to the ontology usability from the user's point of view by including additional information in the form of annotations in the ontology; 2) **Reasoning WP** that refers to the implicit knowledge derived from the ontology when reasoning procedures are applied to such an ontology; 3) **Naming WP** that refers to the ontology usability from the user's point of view and, specifically, to the naming of the ontology elements; 4) **Logical WP** that refers to the solution to design problems in which the primitives of the representation language used do not provide support; and 5) **Content WP** that refers to the solution to design problems related to the ontology domain.

Table 1. WPs descriptions.

WP Name	WP Description
Synonyms as classes	This WP consists both in creating several classes whose identifiers are synonyms and in defining them as equivalent.
Label vs. Comment and other annotations	This WP consists in interchanging the contents of the annotations of the types "label" and "comment" and in not including any annotation of the type "label" and "comment".
Inverse relationships that are not inverse	This WP consists in defining two relationships as inverse when, in fact, they are not.
Undefined inverse relationships	This WP consists in having inverse relationships in the ontology, but they are not defined as such.
Recursive definition	This WP entails using an ontology element in its own definition.
Multiple classes in domains and/or ranges of relationships and/or attributes	This WP consists in defining the domains and/or ranges of the relationships and/or attributes by intersecting several classes in cases in which they should be the union of such classes.
Polisemy	This WP entails using an ontology element to represent concepts different from the domain under consideration.
Same URI for different ontology elements	This WP entails assigning the same URI to two different ontology elements.

WP Name	WP Description
Relationship *"is"*	This WP entails confusing the subclass relationship (*subclassOf*), the membership to a class (*instanceOf*), or the equality between instances *(sameIndividual*) with an ad hoc relation called "is".
Class 2 in 1	This WP entails creating a class whose name is "Class1AndClass2".
Classes vs. Instances	This WP consists in deepening into a hierarchy so that the more specific classes do not have instances since such classes become class instances of the upper level of the hierarchy.
Relationship and/or attributes without domain and/or range	This WP consists in not specifying the domain or range in the relationships/attributes.
Incomplete information	This WP entails not representing all the knowledge that could be included in the ontology.
Miscellaneous class	This WP consists in creating an artificial miscellaneous class to classify in a certai n level the instances not belonging to any of the sibling classes of this level.

In Section 5.1 we present a set of WPs that have appeared very frequently in the ontologies analyzed.These WPs are related among them in Section 5.2.

5.1 Worst practices in relationship modelling

This section presents three examples of WPs, identified in the ontologies we have analyzed, that would affect the relationship modelling. The first one is related to logical patterns, and the other two, to reasoning patterns.

1. **Relationships and/or attributes without domain and/or range.** This WP entails not specifying the domain and/or range in the relationships/attributes. In the ontology analyzed we found that the relationship "itIsIn" has been defined without specifying nor its domain nor range. That relationship should be defined including the domain and the range involved. For example, the student should define the relationship "itIsIn" that have "Construction" as domain and "Place" as range.
2. **Undefined inverse relationships.** The WP consists in having inverse relationships in the ontology but not defining them as such. This implies that when reasoning we obtain less information than that we could infer. For example, within an ontology on architecture, we have observed how the relationships "itIsIn" and "isBuilt" have been created without defining them as inverse. With this modelling, if we have that `"building1 itIsIn site3"`, then, when we activate a reasoner, we will not obtain that in `"site3 isBuilt building1"`. If relationships were defined as inverses then the reasoner could infer the expected knowledge.
3. **Inverse relationships that are not inverse.** This WP consists in defining two relationships as inverse when, in fact, they are not. As a consequence of this WP, undesired knowledge is probably obtained when a reasoner is applied to the ontology. For example, within an ontology on art, we have seen that the relationships "isSoldIn" and "isBoughtIn" are defined as inverse. This fact could cause an error at the semantic level, since if we have that `"object1 isSoldIn place3"`, then the reasoner will infer that `"place3 isBoughtIn object1"`. An appropriate solution would be to replace the relationship "isBoughtIn" by "isPointOfSale". In this way, the semantic error above

commented disappears, since if "`object1 isSoldIn place3`", then the reasoner will infer that "`place3 isPointOfSale object1`".

5.2 Relationships among worst practices

During the analysis of the ontologies we have observed that the WPs can be related through different types of relationships. For instance, some WPs can be specific cases of a more general WP, or a WP can occur as a consequence of other WP.

With regard to the WPs presented in Section 5.1, it should be noted that the "Inverse relationships that are not inverse" and "Not defined inverse relationships" WPs are inverse. That is, in the first case, not valid knowledge is represented, whereas in the second case knowledge is omitted although is valid. In addition, these two WPs can be a consequence of the "Relationships and/or attributes without domain and/or range" WP, since in the case of "Not defined inverse relationships" WP, two inverse relationships may not be defined as such because their ranges and domains have not been defined. In other words, if there are two relationships in which the domain of one is the range of the other and vice versa, we may hint that both are inverse; however, since both domains and ranges are not defined, the hint is not at all clear. We would try to clarify these notions through the picture shown in Fig. 3[3].

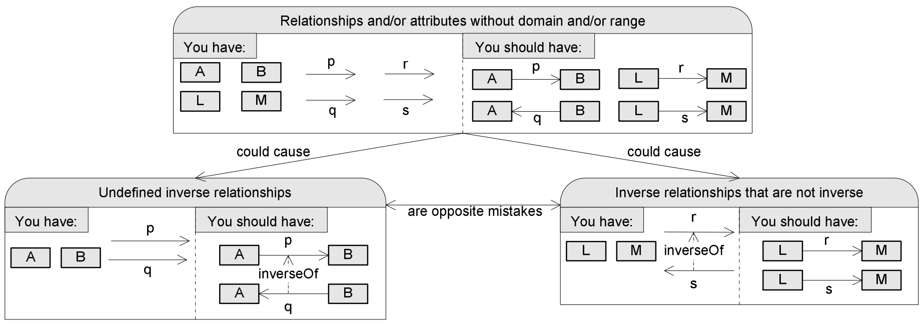

Relationships and/or attributes without domain and/or range
You have:
You should have:
could cause
could cause
Undefined inverse relationships
are opposite mistakes
Inverse relationships that are not inverse
inverseOf

Fig. 3. Some relationships among worst practices

We think that the relationships between the WPs can be very useful during the development of the methodological guides to avoid WPs. Therefore, we are currently analyzing the existence and types of relationships.

6 Conclusions and Future Lines of work

This paper presents the analysis carried out on 11 ontologies, which come from an academic experiment, with the aim of identifying a series of patterns and anti-patterns. During the analysis we have also found a set of WPs that may appear during

[3] The rectangles, the arrows with continuous line and discontinuous line represent the classes, the relationships, and the *inverseOf* properties within an ontology respectively.

the ontology development. The WPs identified have been classified according to a subset of the types of ODPs found in the literature with which they could be related. In addition, a set of examples of these WPs is provided in this paper.

However, even though it is possible to find relationships among the different WPs, as already shown, we have not studied all the possible relationships. Therefore, as a future line of work we propose the identification of the relationships between the different WPs. It would be interesting to identify the groups of WPs that usually appear simultaneously so that, once the concurrence of a WP is identified, the other WPs that could appear could also be identified.

We also propose to analyze the existence of WPs and their relationships in domains other than those represented in the ontologies studied in this paper.

Another interesting work could be to carry out the analysis described in this paper but modifying the ontology development process followed by the students. In this new process we propose to include notions about ODPs in order to compare the resulting ontologies with and without notions about OPDs and analyze to what extent the WPs do not appear.

Finally, as we have already mentioned, the WPs identified are not related to any ODP present in the current libraries; therefore, we are investigating the creation of new design patterns with the objective of avoiding the use of such WPs.

Acknowledgments. This work has been partially supported by the European Commission project NeOn (FP6-027595) and the Spanish project GeoBuddies (TSI2007-65677-C02).

References

1. Aguado De Cea, G., Gómez-Pérez, A., Montiel-Ponsoda, E., and Suárez-Figueroa, M.C. *Natural Language-Based Approach for Helping in the Reuse of Ontology Design Patterns*. In *Proceedings of the 16th International Conference on Knowledge Engineering (EKAW)*, pp. 32-47. (2008)
2. Blomqvist, E.; Gangemi, A.; Presutti, V. *Experiments on Pattern-based Ontology Design*. In *Proceedings of the Fifth International Conference on Knowledge Capture (K-CAP)*, pp. 41-48. (2009)
3. Gangemi, A. *Ontology Design Patterns for Semantic Web Content*. Musen *et al.* (eds.): Proceedings of the Fourth International Semantic Web Conference (ISWC 2005), Galway, Ireland. LNCS, vol. 3729, pp. 262–276. Springer, Heidelberg (2005)
4. Gangemi, A.; Presutti, V. *Ontology Design Patterns*. Handbook on Ontologies (Second Edition). S. Staab and R. Studer Editors. Springer. International Handbooks on Information Systems. (2009)
5. Presutti, V.; Gangemi, A.; David S.; Aguado de Cea, G.; Suárez-Figueroa, M.C.; Montiel-Ponsoda, E.; Poveda, M. *NeOn D2.5.1: A Library of Ontology Design Patterns: reusable solutions for collaborative design of networked ontologies. NeOn project.* http://www.neon-project.org. (2008)
6. Suárez-Figueroa, M.C.; Brockmans, S.; Gangemi, A.; Gómez-Pérez, A.; Lehmann, J.; Lewen, H.; Presutti, V.; Sabou, M. *NeOn D5.1.1: NeOn Modelling Components*. *NeOn project.* http://www.neon-project.org. (2007)

View Inheritance as an Extension of the Normalization Ontology Design Pattern

Bene Rodriguez-Castro, Hugh Glaser, and Ian Millard

Intelligence, Agents and Multimedia Group,
School of Electronics and Computer Science,
University of Southampton,
Southampton SO17 1BJ, UK,
{b.rodriguez, hg, icm}@ecs.soton.ac.uk

Abstract. There are ontology domain concepts that are difficult to represent due to the complexities in their definition and the presence of multiple alternative criteria to classify their abstractions. To assist ontologists in overcoming these challenges, an analysis of available design patterns in ontology and object-oriented modeling has been carried out. As a result, the View Inheritance Ontology Design Pattern (ODP) is introduced. The pattern extends the Normalization ODP (a.k.a. Untangling or Modularization) and revelas the notion of Inter- and Intra-criterion Multiple Inheritance. Our contribution is illustrated with a concrete example of a use case scenario that benefits from the outcome of this study.

1 Introduction

Ontologies have emerged as one of the key components needed for the realization of the Semantic Web vision. Ontologies bring with them a broad range of development activities that can be grouped into what is called ontology engineering. Ontology engineering practices present many similarities to those in the software engineering field and there have been different adaptations of software engineering principles to the ontology engineering domain [1]. Within ontology engineering, this research primarily focuses on ontology modeling, more specifically on Ontology Design Patterns (ODPs) [2] and on how they can help representing complex domain concepts. ODPs have evolved from the general notion of design pattern, defined in [2] as "archetypal solutions to design problems in a certain context" and they are justifiably receiving a significant amount of attention by ontologists due to the preceding success achieved by software design patterns [3].

In this study, we introduce the View Inheritance ODP. The View Inheritance pattern intends to represent ontology domain concepts that presents multiple alternative criteria to classify their abstractions. The motivation that led us to the development of the View Inheritance pattern originates in the ReSIST (Resilience for Survivability in Information Society Technologies) project [4]. One of the objectives of ReSIST is to create a knowledge base application in the field of resilient and dependable computing. The ReSIST Knowledge Base

(RKB)[1] features an ontology in the domain of resilient systems. This ontology was built using the definitions and taxonomies presented in [5].

Among all the ReSIST concepts, one that particularly stands out from a representational point of view is the concept of "Fault". The concept of "Fault" referred hereto is extensively detailed in [5]. This concept involves certain complexities that makes it difficult to represent in the ReSIST ontology, such as a) the dual role that it supports in the ontology, b) the number of relationships that it participates in with other ontology domain concepts, c) support for classifying occurrences of actual faults in real world systems and d) providing a keyword index for subjects of publications and research interests/areas of projects, institutions or people. The characteristics of role and reusability of domain concepts that we identified in [6] laid out some guidelines to handle the first two aspects. To handle the rest, it is crucial that all types of faults identified in [5] can be represented in the ReSIST ontology.

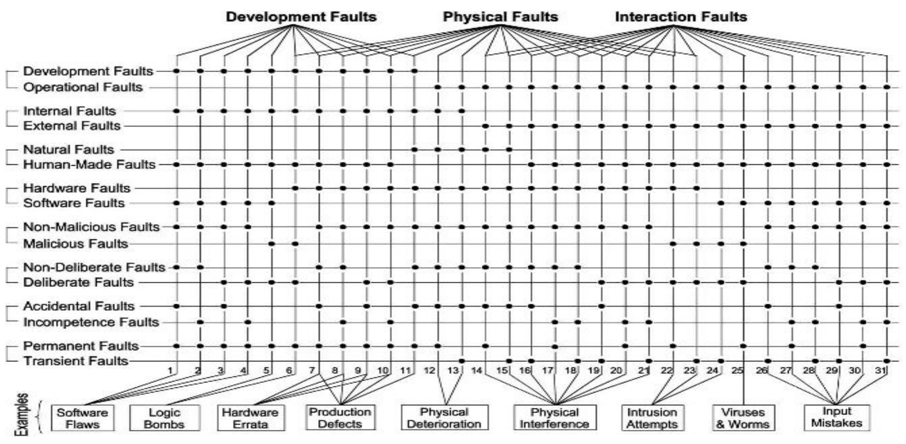


Fig. 1. Matrix representation of "Fault" in Avizienis et al. [5] used in the ReSIST KB ontology

Figure 1 in particular, shows a matrix representation of all types of faults which may affect a system during its life (see also Figure 5b in [5] for a tree representation of these faults). Implicitly, the figure reveals several alternative criteria for the classification of faults:

- A first criterion can be derived from the left column of the matrix. This column represents the values of the eight basic viewpoints (see Figure 4 in [5]) which lead to the elementary fault classes : "Development/Operational Faults", "Internal/External Faults" and so on.

[1] http://rkbexplorer.com/

– A second criterion can be abstracted from the bottom row (listing numbers 1 to 31). This row represents the 31 likely combinations of fault classes out of the 256 possible.
– A third criterion is implicit at the top row, representing the three major partially overlapping groupings of faults: “Development”, “Physical” and “Interaction”.
– A fourth criterion can be seen at the bottom row, labeled “Examples”, containing nine illustrative examples of fault classes.

The representation of multiple alternative criteria (views) to classify the abstractions of a certain domain concept motivated the development of the View Inheritance ODP, which is explained throughout the paper as follows: Section 2 presents a brief overview of the main work related to the modeling of ODPs. Section 3 introduces the View Inheritance ODP in detail and finally, Section 4 covers the conclusions gathered from this endeavor and open lines for further investigation.

2 Related Research

An initial set of ODPs is introduced in [2]. The notion of Conceptual (or Content) Ontology Design Patterns (CODePs) is defined and several examples are included. Future work in the direction of CODePs has been carried out in the context of the NeOn project[2] [7][8]. They present a more detailed overview and refinement of different types of ODPs at different levels of abstraction (see Figure 2.2 in [8]) and a thorough catalogue of patterns is also documented and made available online[3].

Experiences in the development of large ontologies in the Biology domain led to the development of a separate repository of ODPs[4] [9] although with a significant degree of overlap with the one previously mentioned.

From all the patterns surveyed, only the Normalization ODP [9][10] analyzes the implications of having a high number of multiple inheritance relations in the ontology and it refers to the notion of modelling different *semantic axes* as the cause that can lead to polihierarchical structures or a *tangled* ontology. It then outlines a very effective step by step procedure that would *untangle* the ontology becoming a collection of independent modules easy to maintain.

To achieve this however, it relies on modelling constructs available only at the OWL-DL expressivity level of the OWL language (such as *owl:disjointWith* and to some extent *owl:intersectionOf*). This also implies the need of an OWL-DL capable reasoner in order to fully benefit from the use of this pattern. Unfortunately, in the context of ReSIST support for OWL-DL reasoning is beyond the scope of the project.

[2] http://www.neon-project.org
[3] http://ontologydesignpatterns.org/
[4] http://odps.sourceforge.net/

Another aspect where the Normalization ODP and the representation of "Fault" in ReSIST diverge is connected to the normalization criteria of the pattern. These criteria requires the primitive skeleton of the domain ontology to consist only of disjoint homogeneous trees [10]. Intuitively, this appears as a very limiting constrain considering the amount of overlap that Figure 1 reveals among the types of faults identified.

Praising the virtues of the Normalization pattern, we attempted to find complementary material to characterize the modelling scenario that the many faces of "Fault" presents. In that sense and again in the domain of O-O software, the rationale of View Inheritance given in [11] as part of his detailed taxonomy of *types of inheritance* and discussion of *multiple inheritance* seem to correlate very well with the representation needs of "Fault".

In summary, the related work referred hereto, provided an essential framework that served as the basis for the proposed View Inheritance ODP to assist with the development of the ontology component of ReSIST.

3 View Inheritance Ontology Design Pattern

It might be too early to consider View Inheritance a full right member of the ODP family given that certain aspects of it are still being refined and a more extensive collection of archetypal use cases is still needed. The intent of the pattern is to provide a characterization to the ontological modelling problem of representing a certain domain concept whose abstractions can be classified according to multiple alternative criteria. It represents all of these relevant classification criteria so that multiple possible combinations of the concepts that refine them are allowed.

The applicability requirements below stems from the use of View Inheritance in the O-O design [11]. Nonetheless, the level of abstraction is high enough to deem them suitable for ontology development as well. In ontology design terms, these applicability requirements can be summarized as:

– The various classification criteria of the ontology domain concept being represented are equally important, so any choice of a primary one would be arbitrary.
– Many possible combinations of the concepts that refine the various classification criteria are needed.
– Reusability. The domain concepts under consideration are so important as to justify spending significant time to get the best possible inheritance structure.

Structure (Diagram). Figure 2 provides a generic graphical representation of the View Inheritance pattern while Figure 3 shows a subset of the classes involved in the overall representation of "Fault" and how it aligns to this generic structure.

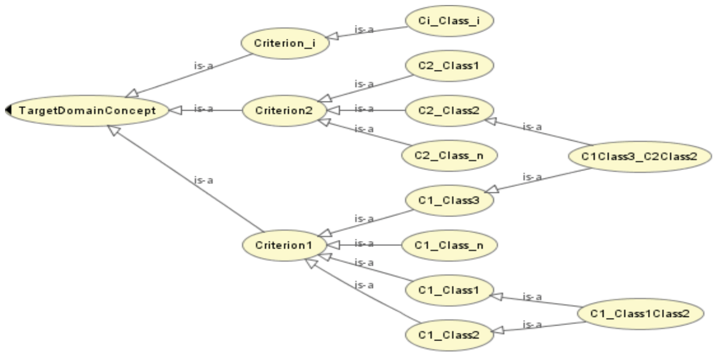


Fig. 2. Structure of a generic use case of the View Inheritance ODP.

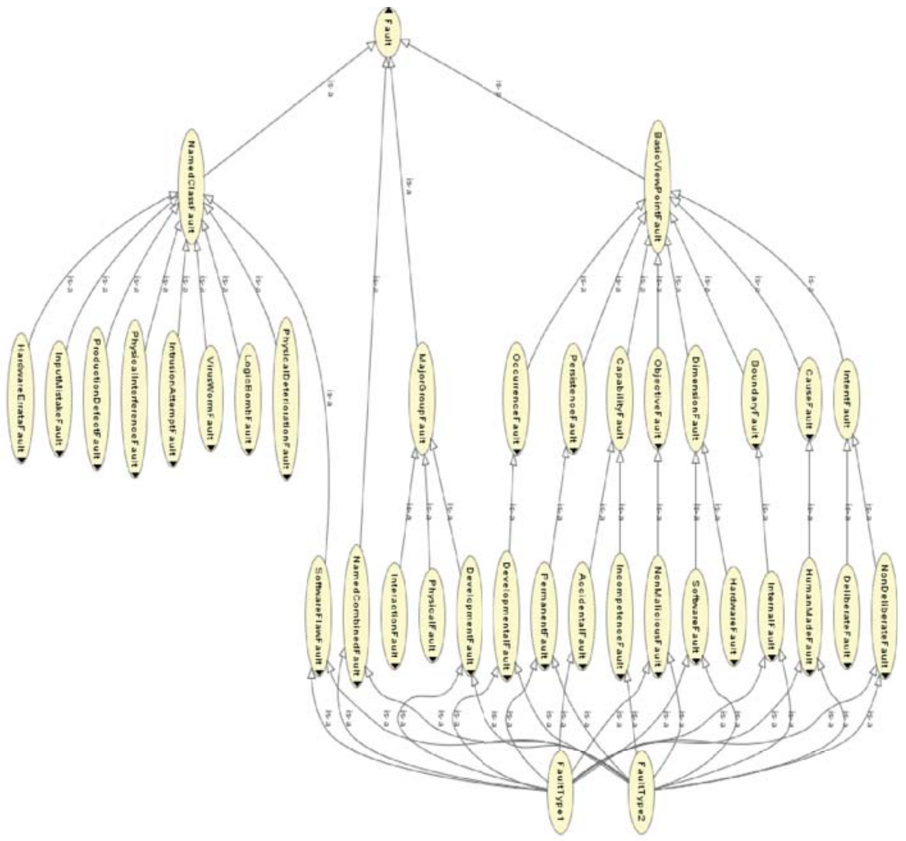


Fig. 3. Structure of the View Inheritance ODP for the representation of "Fault". For simplicity, only 2 types of faults are shown out of the 31 types defined.

Elements and Relationships. The classes participating in the pattern together with their responsibility are listed below. The generic names for classes corresponds to classes located in Figure 2 and the names of classes from the representation of the "Fault" example are located in Figure 3.

- **TargetDomainConcept** (Fault)
  - This class represents the ontology domain concept being defined for which multiple alternative abstraction criteria exist.
- **Criterion_i** (BasicViewPointFault, MajorGroupFault, NamedClassFault, NamedCombinedFault)
  - These classes represent each one of the alternative abstraction criteria of the TargetDomainConcept (Criterion1, Criterion2, Criterion_i in Figure 2). The list of classes may not be exhaustive or pairwise disjoint.
- **Ci_Class_x**[5] (All subclasses of BasicViewPointFault, MajorGroupFault, NamedClassFault, NamedCombinedFault)
  - These classes refine each abstraction criteria class (C1_Class1, ..., C2_Class1, ..., Ci_Class_i in Figure 2). The list of classes may not be exhaustive or pairwise disjoint.
- **CiClass_x_CjClass_y**[6]**, Ci_Class_xClass_y**[7] (FaultType1, FaultType2, ..., FaultType32)
  - These classes participate in multiple inheritance relationships combining different refinements from the alternative abstraction criteria classes (C1Class3_C2Class2 and C1_Class1Class2 in Figure 2).

Inter- and Intra-criterion Multiple Inheritance. There is an interesting feature regarding the types of multiple inheritance relations that can take place in the context of a View Inheritance pattern, that to the best of our knowledge has not been made explicit so far in ontology modeling. These types of multiple inheritance relationships can be characterized as:

- **Inter-criterion**, when the parent classes involved in the multiple inheritance relation are subclasses of different abstraction criteria. The class C1Class3_C2Class2 in Figure 2 is an example of this type of inheritance because one of its parent classes, C1_Class3, is a refining concept of Criterion1 and the other parent class, C2_Class2, is a refining concept of Criterion2.
- **Intra-criterion**, when the parent classes involved in the multiple inheritance relation are subclasses of the same abstraction criterion. The class C1_Class1Class2 is an example of this type of inheritance because all of its parents classes, C1_Class1 and C1_Class2, are refining concepts of the same criterion, Criterion1.
- **Intra- and inter-criterion**, when there are at least two parents involved in the relation that are subclasses of the same abstraction criterion and there is at least one more different parent that is a subclass of a different abstraction criterion. An example of this type of inheritance is trivial to extrapolate from the composition of the previous two.

[5] Short for: Class_x from Criterion_i
[6] Short for: Class_x from Criterion_i and Class_y from Criterion_j
[7] Short for: Class_x and Class_y from Criterion_i

Nested View Inheritance. It is worth noting that View Inheritance patterns could occur in a nested fashion. That is, a View Inheritance pattern with a concept Ci as root, could enclose another case of a View Inheritance pattern with a different concept Cj as root where Ci subsumes Cj.

As an example, consider the View Inheritance scenario with "Fault" as root in Figure 3. Let us focus in the subtree that originates in one of its abstraction criterion, in this case "BasicViewPointFault" and *zoom in* on it. From the point of view of "BasicViewPointFault" and with this concept as root, its direct subclasses ("ObjectiveFault", "BoundaryFault" and so on) could be regarded as alternative abstraction criteria of "BasicViewPointFault". The classes "FaultType1" and "FaultType2" could be regarded as cases of inter-criterion multiple inheritance given that each one of their parents is a class subsumed by a different abstraction criterion of "BasicViewPointFault".

ODP Classification. According to the classification in [8], the View Inheritance ODP could be considered an Architectural OP given that it provides an ontological characterization of a particular case of polihierarchy in the overall ontology structure. By the same classification, the definition of Inter- and Intercriterion Multiple Inheritance could be regarded as Logical OPs given that they refine the Logical OP that describes the case of generic multiple inheritance in [7][8]. This classification situates View Inheritance at a higher level of abstraction than that of Inter- and Intra-criterion Multiple Inheritance.

Based on the classification of ODPs in [9], View Inheritance can be regarded as an *extension* to the Normalization ODP, which is considered part of the group of Good Practices ODPs. In this sense, the former concentrates on certain ontological aspects of the modelling problem that the latter addresses. View Inheritance focuses on the nature of a specific type of polyhierarchy structure and provides a characterization of it.

4 Conclusions and Future Work

There are a number of key points that would be worth summarizing. There can be certain ontology domain concepts difficult to represent due to the existing alternative abstraction criteria that define them. That is the case of the "Fault" concept in the context of the ReSIST project.

A survey of the current ontology building techniques was carried out. The Normalization ODP seemed a viable option, yet the pattern did not fully address the definition of "Fault" and the application requirements of the ReSIST project.

To bridge this gap, the View Inheritance ODP is put forward as an extension to the Normalization ODP, combining the latter with the notion of View Inheritance originated in the O-O software design. View Inheritance revealed two basic types of likely relations that could take place in the structure of the pattern: Inter- or Intra-criterion Multiple Inheritance.

These contributions, while not solving all the modelling challenges of the ontology module for ReSIST, do provide additional awareness to be considered in the development process.

Outstanding issues open for future work include the identification of additional examples of real world use cases of View Inheritance. In that sense, we intend to selectively explore the ontologies most frequently used in the data repositories that are part of the Linked Data project[8] for possible occurrences of View Inheritance patterns. The ReSIST project is one of the contributors to the Linked Data set of repositories.

References

1. Gomez-Perez, A., Corcho, O., Fernandez-Lopez, M.: Ontological Engineering : with examples from the areas of Knowledge Management, e-Commerce and the Semantic Web. First Edition (Advanced Information and Knowledge Processing). Springer (July 2004)
2. Gangemi, A.: Ontology design patterns for semantic web content. In Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A., eds.: International Semantic Web Conference. Volume 3729 of Lecture Notes in Computer Science., Springer (2005) 262–276
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional (1995)
4. ReSIST: Resilience and Survivability in IST. Network of Excellence. Contract: IST 4 026764 NOE, EU and Sixth Framework Programme (FP6) and Information Society Technology (IST) (2005–2008) http://www.resist-noe.eu/.
5. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing **01**(1) (2004) 11–33
6. Rodriguez-Castro, B., Glaser, H.: Whose "fault" is this? untangling domain concepts in ontology design patterns. In: 1st International Workshop on Knowledge Reuse and Reengineering over the Semantic Web at the 5th European Semantic Web Conference. (June 2008)
7. Suarez-Figueroa, M.C., Brockmans, S., Gangemi, A., Gomez-Perez, A., Lehmann, J., Lewen, H., Presutti, V., Sabou, M.: Neon modelling components. NeOn deliverable D5.1.1, Universidad Politecnica de Madrid (2007)
8. Presutti, V., Gangemi, A., David, S., de Cea, G.A., Surez-Figueroa, M.C., Montiel-Ponsoda, E., Poveda, M.: A library of ontology design patterns: reusable solutions for collaborative design of networked ontologies. NeOn deliverable D2.5.1, Institute of Cognitive Sciences and Technologies (CNR) (2008)
9. Egana-Aranguren, M.: Ontology Design Patterns for the Formalisation of Biological Ontologies. MPhil Dissertation, Bio-Health Informatics Group, School of Computer Science, University of Manchester (2005)
10. Rector, A.L.: Modularisation of domain ontologies implemented in description logics and related formalisms including owl. In: K-CAP '03: Proceedings of the 2nd international conference on Knowledge capture, New York, NY, USA, ACM (2003) 121–128
11. Meyer, B.: Object-Oriented Software Construction (Book/CD-ROM) (2nd Edition). Prentice Hall PTR (March 2000)

[8] http://linkeddata.org/

Ontology Naming Pattern Sauce for (Human and Computer) Gourmets

Vojtěch Svátek[1], Ondřej Šváb-Zamazal[1], and Valentina Presutti[2]

[1] Department of Information and Knowledge Engineering,
University of Economics, W. Churchill Sq.4, 130 67 Prague 3, Czech Republic
{svatek|ondrej.zamazal}@vse.cz
[2] ISTC-CNR, Via Nomentana 56, 00161 Rome, Italy
presutti@cnr.it

Abstract. Various explicit and implicit naming conventions for entities have emerged in ontological engineering realms during the decade/s of its existence. In the paper we argue that the naming principles are neither trivial nor completely haphazard in practice, present a preliminary categorisation of ontology naming patterns, and discuss the impact of entity naming on both human and computer perception of ontologies.

1 Introduction

By the *OntologyDesignPatterns.org* (ODP) portal categorisation, ontology naming patterns (Naming OPs) are "good practices that boost ontology readability and understanding by humans, by supporting homogeneity in naming procedures". The present work is one of first attempts to systematically populate this category of design patterns; there has recently been similar effort carried out in the narrower context of bioinformatics [6], and references to entity name content have been made in general literature on ontological modelling such as [3, 8].

Meaningful names are helpful for both *people* and *machines*. Undoubtedly, in particular from the point of view of machine 'consumers', the logical structure of an ontology is mandatory and unambiguous, while entity naming is dependent on subjective choices of designers, and is even optional in the sense that random strings can be used instead of names. Metaphorically, we could thus view the logic as 'meat' and naming as 'sauce'. Even if sauce is not a necessary part of every meal, it often helps *digest* the meat, and could in some cases be more *caloric* (to read: bear more real-world semantics) than the meat. 'Digesting' the logic is easy when entities are presented to the user in small chunks, such as in window-based interfaces of ontology editing environments. However, in this mode, only a small part of the whole knowledge structure can be viewed. On the other hand, various linear and diagrammatic notations allow to display larger clusters of entities but require the user to employ his/her intuition on the role of each entity in the structure. Then natural-language-like naming gains on importance. Note that some user-facing initiatives in ontological engineering, such as the introduction of Manchester syntax for OWL [1], use natural-language-like features to improve the readability at the level of *meta-model constructions*. Naming patterns could play an analogous role at the level of *model entities*. Naming can also increase the 'nourishing factor' (i.e. information value) of knowledge structures, just because of the same feature that precludes their unambiguous processing: while the inventory of logical constructs (and even logical design patterns) in a language such as OWL is necessarily restricted by the language norm, naming conventions and patterns can exploit any kind of structure that can be expressed within alphanumeric strings. While 'digesting' is only an issue for humans, 'additional calories' can be quite beneficial for software tools that analyse and process ontologies, such as ontology matchers over complex correspondences [5].

Let us rapidly demonstrate the 'digestive' and 'nourishing' potential of adequate naming on an OWL restriction in Manchester syntax:

```
StateOwned Director only (nomination some ministry)
```

With more careful naming the same axiom could look like this:

```
StateOwnedCompany hasDirector only (nominatedBy some Ministry)
```

Presumably, this version much more clearly conveys the message that "all directors of state-owned companies are nominated by some ministry". We will refer to elements of this example later. The rest of the paper is structured as follows. Section 2 outlines our principles of categorising naming patterns. Section 3 then characterises different categories of patterns, including examples from existing ontologies.[3] We first discuss generic naming conventions, then focus on patterns specific for a particular entity type (classes, instances or properties), and finally on patterns spanning over multiple entities. Finally, Section 4 wraps up the paper and outlines directions for future research.

2 Naming Pattern Categorisation Criteria

In this first approximation we suggest to categorise naming patterns along four interdependent dimensions: (1) by structural complexity and underlying (modelling) language construct; (2) by lexical specificity and linguistic depth; (3) by domain specificity; (4) by descriptiveness/prescriptiveness.

In this paper we use the *structural complexity* of the pattern and underlying *language construct* (from the meta-model) as the primary categorisation criterion, as it is rather crisp. In this respect, we distinguish between generic naming conventions, single-entity patterns related to different entity types (classes, object properties, data properties and instances) and cross-entity patterns related to constructs such as class-subclass pairs or pairs of mutually inverse properties. For the moment, we do not systematically cover patterns defined on the top of more than two directly connected entities. We also assume the underlying language to be OWL, although naming patterns are obviously, compared to logical patterns, less sensitive to shifting to a different language (say, with different formal semantics but similar outlook, as is the case with frame languages).

[3] A more thorough description is in the long version of the paper, see `http://nb.vse.cz/~svatek/wop09long.pdf`.

Patterns can differ in their *lexical specificity.* Some refer to concrete lexemes, which can be both 'stop words' (such as 'is' or 'of') and 'semantic' words (such as 'part'); on the other hand, some patterns only refer to parts of speech. The *linguistic depth* of patterns may span from surface attributes of strings such as capitalisation or presence of numerals to patterns referring to deeper linguistic notions such as active/passive mode of verbs.

Some naming patterns can certainly be characteristic for problem *domains*, say, engineering or genomics. We do not consider this aspect here.

Finally, we include both patterns that have been tentatively verified as 'frequent' in *existing ontologies*, i.e. 'descriptive' patterns, and patterns that we see as useful as guidance for developing *new ontologies* (or reengineering old ones) even if they are not widely used nowadays, i.e. 'prescriptive' patterns. We believe that naming patterns should on the one hand try to accomodate what is intuitive for most modellers (and thus widely used) and on the other hand promote clarity and readability even at the cost of going against the mainstream.

3 Detailed Descriptions of Naming Pattern Categories

3.1 Generic Principles and Conventions

Naming Vocabulary In view of comprehensibility to humans as well as NLP tools, the terms from which an entity name is constructed should be built from *human language vocabulary*; as mentioned in the introduction, the designer should not forget that the ontology will probably be used not only by purely formal reasoners but also by people and even NLP procedures that could leverage on meaningful naming. Furthermore, *abbreviations* (as also suggested in [3]) and *colloquialisms* should be avoided. *Acronyms* are often inevitable; however, the practice of using acronyms as prefixes of whole taxonomic trees, as artificial codes indicating the membership of the entity to this tree, is questionable, as it alienates the naming from the natural language.

The requirement of using human language naturally does not stipulate that only terms from common, generic vocabularies can appear in entity names. Specific domains may have their own terminology that is only familiar to a few dozens of experts and still could (or even should) be included in ontologies. Some of the terms may not exhibit typical features of words in human language; for example, names of genes in a gene ontology would consist of mixed alphabetic/numeric strings. Moreover, terms having a different generic meaning could be used in a specific domain ontology; for example the term 'Mouse' (as one of numerous metaphoric terms that are no longer viewed as colloquialisms) can be used in a domain ontology of computer equipment without the need for (unnatural) specifier such as 'ComputerMouse'. Care should however be taken when using terms so generic that they could interfere with entities in the *same* ontology (e.g. qualifier terms such as 'high' or 'light'); this problem is discussed in Section 3.3.

Case and Delimiter Conventions Such conventions exist even in programming environments. For OWL ontologies, a minimal requirement on capitalisation and delimiters seems to be to keep the same convention for all occurrences of one entity type; in addition, we would recommend, consistently with [3] (and following conventions used in description logics), to capitalise class names and decapitalise property names. As we saw in Example 1, this improves the readability of complex OWL restrictions, which often consist of sequences of alternating class and property names (aside modelling language keywords). For delimiters, OWL best practices do not encourage blanks in names, so underscore (This_Class), hyphen (This-Class) and 'camel case' (ThisClass) are all frequently used. In our opinion, however, underscore and 'camel case' are better alternatives, as the use of hyphen may interfere with compound words (in which the token before the hyphen often has a different role than if the same term were used in appositive), especially if the ontology is analysed by an automated NLP procedure that tries to properly tokenise each entity name.

3.2 Class Naming Patterns

The central issue in naming classes is whether the name of a class should imperatively be a noun phrase and whether it should be in singular or plural. We would strongly encourage *singular* for OWL ontologies: first, it is nowadays predominant in existing ontologies; second, some linear RDF notations such as N3[4] expect it in their syntax by using the 'a' (indefinite article) token for instance-class relationship, such as "John a Person" (John is an instance of class Person). On the other hand, there are situations where merely syntactical *plural* is fully justified for a class name. Let us consider 'Bananas' as subclass of 'FruitMeal' in a catering ontology: here, multiple physical entities (bananas) play the role of a single object (meal) and do not matter individually.

There does not seem to be any logical reason for using another part of speech than noun for class name. Modellers sometimes omit the noun if it is present at a higher level of the hierarchy, and only use the specifying *adjective*, such as 'StateOwned' as subclass of 'Company' in our initial example. We however discourage from such shorthanding. First, for elementary comprehensibility reasons illustrated on Example 2. Second, even if frame-based ontology engineering is tolerant in this respect ([3] for example only discourages from incomplete shorthanding, such as having both 'RedWine' and 'White' as subclasses of 'Wine'), note that in OWL ontologies, due to the underlying description logics, the explicit taxonomy is only secondary to axiomatisation as such. Making a concept anyhow dependent (even in a 'harmless' manner, such as in terms of naming) on its parent concept is thus rather awkward.

On the other hand, entity names consisting of *too many* tokens are also undesirable. It may be the case that they could be transformed to *anonymous classes* as part of axioms, see for example 'FictionalBookbyLatinAmericanAuthor' mentioned by Welty [8] or linguistic disjunctions mentioned below.

[4] `http://www.w3.org/2000/10/swap/Primer`

3.3 Instance Naming Patterns

If individuals are present in an OWL ontology, their names typically correspond to *standard vocabulary noun phrases*; examples are chemical elements or political countries. In very specific ontologies (or ontologies that are melted with a specific knowledge base) instance names could also be *non-linguistic strings* such as names of genes or product codes. Individuals are sometimes also used for *specified values*, as depicted in the corresponding in logical pattern [4]. Then the enumerated individuals (usually declared as different from each other) define a class; e.g. the set of individuals 'poor_health', 'medium_health' and 'good_health' defines the class 'Health_value'. A subtle issue is whether the name of such an individual can be other than noun phrase. It seems that if an individual is to denote a mere 'value' or 'status' rather than a real-world entity, the part of speech of its name does not matter in principle. However, using plain adjectives such as 'good' or 'high' is tricky. Note that there is a risk of confusing such individuals with the general notions of 'goodness' or 'highness'; this is emphasised by the status of individuals as first-class citizens in OWL. It may then easily happen that an individual originally defining a specified value with respect to a certain class would be *improperly reused with respect to another class*. For example, in a wine ontology[5] the individual Light is part of enumeration of class WineBody; then someone might reuse the same individual as part of enumeration of WineGrape, or even of WineBottle or anything that can be light or heavy. Clearly, the lightness values of wine body are ontologically different from the lightness values of a wine grape; and even the physical lightness values of a wine grape are ontologically different from the lightness values of a wine bottle, as each of them is associated with a different scope of weight (as measurable quantity). For this reason we recommend to refer to the name of class in the name of the individuals representing specified values. On the other hand, there is a risk of confusing the 'value' or 'status' individuals with *real-world entities*; for example the individual representing the *status* of 'excellent student' should probably not be an instance of class Student. A safe option for naming such individuals thus would be to include both the class name and a term such as 'status' or 'value' in their name, e.g. 'poor_health_value', 'excellent_student_status' or 'light_wineBody_value'.

3.4 Property Naming Patterns

Although object properties and data properties have similar status in OWL, their naming seems to be linked to different patterns.

Comprehensibility concerns suggest that the name of an object property should not normally be a plain noun phrase, for clear discernability from class names as well as from the name of the inverse property. Indeed, a majority of object properties either have a *verb* as their head term or end with an attributive *preposition* (such as 'of', 'for'), which indicates that the name should be read as if it started with 'is': for example '(is) friend_of', '(is) component_for'. A *plain*

[5] `http://www.ninebynine.org/Software/HaskellRDF/RDF/Harp/test/wine.rdf`

preposition is occasionally used for spatio-temporal relationships. Furthermore, linguistic processing of ontologies would possibly benefit from the usage of *content verbs* rather than auxiliary ones where appropriate, as content verbs bring additional lexemes into the game. In this sense, property names like 'manufactures' or 'writtenBy' bring 'extra calories' compared to property names like 'hasProduct' or 'hasAuthor' (assuming that the range of the properties is 'Product' and 'Author', respectively). We elaborate further on object properties in the paragraph on naming patterns over restrictions in Section 3.7.

On the other hand, for *data property* names *nouns* seem appropriate, as they are analogous to database fields. Often the 'primitive data' nature of data properties can be underlined by using head nouns such as 'date', 'code', 'number', 'value', 'id' or the like.

3.5 Subclass and Instantiation Naming Patterns

It is quite common that a subclass has the *same head noun* as its parent class.[6] By an earlier study [7] we found out that this pattern typically represents between 50–80% of class-subclass pairs such that the subclass name is a *multi-token* one. This number further increases if we consider *thesaurus correspondence* (synonymy and hyperonymy) rather than literal string equality. Sometimes the head noun also disappears and reappears again along the taxonomic path, as a specific concept cannot be expressed by a dedicated term but only by circumlocution; for example in *Player - Flutist - PiccoloPlayer* (note that `Flutist` is a single-token name, i.e. not in conflict with our pattern), in a music ontology.[7]

Retrospectively, violation of head noun correspondence in many cases indicates a problem in the ontology. Common situations are:

– Inadequate use of class-subclass relationship, typically in the place of whole-part or class-instance relationship, i.e. a *conceptualisation error.*
– *Name shorthanding*, typically manifested by use of adjective, such as 'StateOwned' (subclass of 'Company'), as mentioned above.

While the former probably requires manual debugging of the ontology, the latter could possibly be healed by propagation of the parent name downto the child name. Note that such propagation may not be straightforward if the parent itself has a multi-word name. For example, 'MD_Georectified', which is a subclass of 'MD_GridSpatialRepresentation',[8] could be extended to 'MD_GeorectifiedRepresentation', 'MD_GeorectifiedSpatialRepresentation' or 'MD_GeorectifiedGridSpatialRepresentation', and only deep understanding of the domain would allow to choose the right alternative.

The *class-instance* relationship does not seem to follow generic naming patterns. An exception is the case of specified values discussed in Section 3.3.

[6] The head noun is typically the last token, but not always, in particular due to possible prepositional constructions, as e.g. in 'HeadOfDepartment'.
[7] `http://www.kanzaki.com/ns/music`
[8] Taken from `http://lists.w3.org/Archives/Public/public-webont-comments/2003Oct/att-0026/iso-metadata.owl`.

3.6 Subproperty and Inverse Property Naming Patterns

We are not aware of a conspicuous naming pattern for the *subproperty* relationship. A tentative suggestion for reengineering methods could perhaps be the following: if there are multiple (object or data) properties with same head noun (depending on a usual auxilliary verb), they could possibly be *generalized* to a superproperty. For example, the properties 'hasFirstName', and 'hasFamilyName' could yield 'hasName' as superproperty.

Inverse property naming patterns should help link an object property to its inverse and at the same time discern between the two. They are thus related to the logical design pattern of *bi-directional relations*: if there is no inverse property, there is less of problem at the level of naming but more at the logical level. As canonical inverse property naming patterns we can see the following:

- active and passive form of the same *verb*, such as 'wrote' and 'writtenBy'
- same *noun phrase* packed in auxilliary terms (verbs and/or prepositions), such as 'memberOf' and 'hasMember'.

If the nominal and verbal form are mixed, e.g. 'identifies' and 'hasIdentifier', the accessibility is fine for humans but worse for NLP procedures.

3.7 Naming Patterns over Restrictions

As we mentioned Section 3.4, one alternative for object property name is that including the name of the class in the range and/or domain of this property. This can be seen as a naming pattern over a *global property restriction.* Let us illustrate some options for such patterns on the notorious pizza domain. We suggest that the property from PizzaTopping to Pizza can be labelled as:[9] 'isToppingOfPizza'; 'isToppingOf'; 'toppingOf'; or maybe 'ofPizza'. Intuitively, we probably feel that 'hasPizza' does not sound well. On the other hand, for the inverse property we would rather suggest[10] 'hasTopping' or maybe 'PizzaTopping'.

As possible reasons for the different 'psychologically natural' choice of naming pattern for the mutually inverse properties we could see the nature of topping as 1) an entity *dependent* on a pizza entity (a topping cannot exist without a pizza), or 2) a *role* entity (as being a pizza topping is merely a role of some food). The first hypothesis would mean that the presence of the name of a class in the name of a property (for which this class is in the domain or range) indicates that entity of this class is dependent on the entity on the other side of the property. The second hypothesis would mean that the presence of the name of a class in the name of a property (for which this class is in the domain or range) indicates that this class is a role. Both hypotheses can also be adjusted according to presence of auxilliary verbs ('is', 'has') and suffixed propositions.

In principle, we could also identify naming patterns over *local property restrictions*, for example in the form of 'lexical tautologies' such as `MushroomPizza`

[9] Let us for simplicity ignore the naming options with alternative prepositions ('isToppingOn') or without domain/range tokens at all ('laidOn', 'on').

[10] Again ignoring essentially different options such as 'withTopping' or 'laidWith'.

`equivalentTo (Pizza and contains some Mushroom)`. This issue may deserve further study, although the frequency of such constructions is not very high.

4 Conclusions and Future Work

The intended contribution of the paper is a preliminary system of ontology naming patterns, which we illustrated on examples. Undoubtedly, consistent and comprehensible entity naming is an important aspect of re/usability of ontologies. The main reason why research on this topic has been quite scarce to date is probably the high risk of subjectivity and subtle, heuristic nature of any cues one could figure out. We are aware of this risk; the naming suggestions in this paper are meant to serve as starting point for discussion in the pattern community rather than a mature system of best practices.

Most imminent future work will consist in *large-scale evaluation* of existing ontologies in terms of naming as well as bare plain logical patterns.[11] Within the empirical analysis stream, we should also study the usage of *other textual labels* rather than URI fragments (such as rdf:label and rdf:description), and compare their content with that of the URIs. We would also like to set up a specific *metadata schema* for collecting this type of patterns in the *ODP portal*. Finally, in the context of this portal, we would like to apply the naming patterns to evaluate *other types of ontology design patterns*, especially the content ones.

This work has been partially supported by the IGA VSE grant no.20/08 "Evaluation and matching ontologies via patterns".

References

1. The Manchester OWL Syntax. Online `http://www.co-ode.org/resources/reference/manchester_syntax/`
2. Annotation System. OWL WG, Work-in-Progress document, `http://www.w3.org/2007/OWL/wiki/Annotation_System`.
3. Noy N. F., McGuinness D.L.: Ontology Development 101: A Guide to Creating Your First Ontology. Online `http://www-ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness.pdf`
4. Rector A. (ed.): Representing Specified Values in OWL: "value partitions" and "value sets". W3C Working Group Note, 17 May 2005, online at `http://www.w3.org/TR/swbp-specified-values/`.
5. Ritze D., Meilicke C., Šváb-Zamazal O., Stuckenschmidt H.: A pattern-based ontology matching approach for detecting complex correspondences. In: OM Workshop at ISWC'09.
6. Schober D. et al.: Survey-based naming conventions for use in OBO Foundry ontology development. *BMC Bioinformatics*, Vol.10, Issue 1, 2009.
7. Šváb-Zamazal O., Svátek V.: Analysing Ontological Structures through Name Pattern Tracking. In: EKAW-2008, Acitrezza, Italy, 2008.
8. Welty C.: Ontology Engineering with OntoClean. In: SWAP 2007, Bari.

[11] We plan to continuously update these results at `http://nb.vse.cz/~svabo/namingPatternsAnalysis/`.

Ontology Patterns and Beyond
Towards a Universal Pattern Language

Olaf Noppens and Thorsten Liebig

Inst. of Artificial Intelligence, Ulm University, Ulm, Germany
firstname.lastname@uni-ulm.de

Abstract. In this paper we argue for a broader view of ontology patterns and therefore present different use-cases where drawbacks of the current declarative pattern languages can be seen. We also discuss use-cases where a declarative pattern approach can replace procedural-coded ontology patterns. With previous work on an ontology pattern language in mind we argue for a general pattern language.

1 Motivation

Though ontology engineers are supported by a wide range of authoring tools the task of designing ontologies is still difficult even for experts. Several studies on analyzing common errors in formalizing knowledge may serve as a prime example (e. g. [1], [2]). They have shown that often modeling problems were concerned with the act of writing down conflicting axioms, or with finding solutions for non-obvious but common modeling challenges (e. g. n-ary relationships). Extending the original notion of knowledge patterns [3] ontology design patterns (ODP) provide a modeling solution to solve a recurring ontology design problem [4], [5]. There is a wide range of ODP types such as ontology content patterns (OCPs), logical patterns, architectural patterns etc. [6].

In this paper, we would like to broaden the term *ontology pattern* by looking at different ontology engineering approaches. Some of them rely on procedural knowledge. We argue that lifting them to a declarative level would enable a (semi-)automatic handling of patterns with the aim to support the ontology engineer during the entire ontology's life-cycle. The instantiation of an ontology pattern, e. g., a logical pattern, also depends on conditions that must be fulfilled. Violating these conditions at a later stage typically results in an incorrect instantiation. For that we argue that monitoring pattern instantiations is crucial. In our opinion, there is also a duality between finding pattern instances and instantiating patterns. For instance, finding axioms that correspond to a logical ontology pattern without having explicitly applied that pattern could be useful during (distributed) ontology engineering for better understanding or finding undesired modeling issues. Only little work has been spent in guiding the ontology engineer in finding patterns and monitoring correct instantiation of them.

Recently some work has been done with respect to collaborative and distributed ontology engineering [6] as well as in the context of Protégé 4 [7].

The reminder of this paper is organized as follows: first we discuss use-cases of the application of ontology patterns where some are hidden in procedural knowledge. Then we present a proposal for the obtained requirements and we will briefly discuss the integration in our language in an ontology authoring tool.

2 Use-Cases and Requirements

We will try to broaden the term *ontology pattern* by looking at different ontology engineering support methods to extract commonalities that should be fulfilled by a universal ontology pattern language in order to encode patterns in a declarative way. These use-cases are mainly driven by our experience in ontology authoring and therefore not intended to be exhaustive. As a starting point we take into account a recently proposed pattern language [7] based on OPPL. We refer to it as the *Manchester Ontology Pattern Language* (MPL). To the best of our knowledge, it is the only ontology pattern language.

Ontology Design Patterns Repositories such as `ontologydesignpatterns.org` provide access to pattern catalogues. Instead of encoding patterns in a declarative way they are described by their intent, examples and diagrams. Recently, the authors of MPL propose to describe patterns in a declarative way to support a more automatic handling of patterns. In the following we observe two shortcomings of their approach. Consider a very simple partition pattern: a *partition* is a structure that is divided into several disjoint parts. Using the Abstract Syntax of OWL 2 [8] it can be specified by the following axioms where P is the partition class, and C_1, ..., Cn are arbitrary classes or class expressions:

```
EquivalentClasses(P ObjectUnionOf(C1, C2, ..., Cn))
DisjointClasses (C1, C2, ..., Cn)
```

The first axiom can be expressed in MPL using `createUnion(?x.VALUES)` to bind an arbitrary number of classes to a given variable `?x`. However, MPL does not allow to express a set of classes as needed in the second axiom [1]. Moreover, OPPL/MPL allow variables of named entities only. This limits the usage of the language because some situations (as shown before) cannot be expressed.

To sum up a general pattern language has to deal also with an arbitrary number of classes in conjunction with all axioms in OWL which allow a set of classes (unions, intersections as in MPL but also disjointness, equivalences etc.). All types of OWL objects should be used as variable parts. Otherwise, the very simple partition example would only contain named classes.

[1] We refer here to the MPL grammar available at `http://www.cs.man.ac.uk/?iannonel/oppl/grammar.html`, last accessed on 20th of July, 2009, and [7].

Refactoring/Re-engineering There are different syntactical forms to model the same logical meaning in OWL. One reason is that syntactical sugar makes things easier to read and to express. For instance, property range restrictions can be expressed either as `ObjectPropertyRange(hasParent Person)` or using a GCI `SubClassOf(Thing ObjectAllValuesFrom(hasParent Person))`. The latter axiom, for instance, occurs in ontology transformations from KRSS to OWL. Negative property assertions can be either expressed as `NegativeObject-PropertyAssertion(property i1 i2)` or as `SubClassOf(ObjectOneOf(i1), ObjectComplementOf(ObjectSomeValuesFrom(property ObjectOneOf(i2))))` whereas the former one has been recently introduced in OWL 2. Obviously, in both cases, the first statement is more intuitive and convenient and even experts seem to encounter problems in understanding the latter one. It is, of course, an extreme case but it illustrates the problem of understanding the meaning when users look at ontologies created by others. It is not a rocket science to transfer these axioms into each other by writing some code. However, this is not an adequate solution: the transformation is hidden in procedural knowledge. Another simple re-factoring example is the replacement of cyclic subclass axioms with an equivalent-class axiom. `SubClassOf(C0 C1),..., SubClassOf(Cn C0)` is equivalent to `EquivalentClasses(C0,...,Cn)`. This is, for instance, implemented in a procedural manner in Protégé. This example shows that a pattern language needs also to support variability on axiom level: Here, the number of subclass axioms is unknown during pattern design-time.

Ontology Lint Tools In Software Engineering lint tools perform static analysis of source code in order to detect suspicious scopes of code wrt. language usage, condition violations, violation of type ranges etc. that might lead to unexpected behavior on run-time. Transferring this idea to ontologies, ontology lint tools give hints about potential modeling defects, i. e., they scan for "anti-patterns" that should be avoided. In our opinion, these anti-patterns do not differ from ontology patterns and therefore they should also be expressed in a declarative way – re-usable independent of any programming language. For instance, Pellint [9] tries to detect modeling patterns which potentially will slow down reasoning performance wrt. a tableaux-based reasoner, mainly optimized for the Pellet reasoner. There is also a tentative ontology lint plugin[2] for Protégé 4 providing several default lints similar to those of Pellint. However, in both tools lints are expressed by procedural knowledge, i. e., the analysis methods are encoded in the program. In contrast to general ontology patterns, additional constraints are typically needed: Pellint defines, for instance, a lint pattern that finds all classes with more than 5 explicit asserted subclass axioms for a given class. This cannot be expressed in MPL because neither a condition nor an arbitrary number of axioms (greater than 5) can be expressed. To sum up, a flexible way to use both an arbitrary number of axioms and conditions is needed. Ontology Lints represents the missing link between finding pattern instances, monitoring as well as instantiating them.

[2] `http://www.cs.man.ac.uk/~iannonel/lintRoll`

3 A Proposal towards a Universal Axiom Pattern Language

Inspired by the mentioned use-cases and the shortcomings of MPL we now present building blocks of a pattern language which fulfills the discussed requirements. We do not claim that those requirements are the only ones but they are a good starting point. For the rest of this paper we refer to OWL 2, OWL for short, and its *Abstract Syntax* for presentation purpose. Allowing arbitrary class expressions as possible bindings for variables results in loosing decidability because models of OWL ontologies, in general, are infinite. Therefore, we use the finite set of class (property) expressions as introduced in [10]: given an ontology $\mathcal{O}$, the finite set of class (property) expressions consists of all class (property) expressions that appear in the asserted axioms of $\mathcal{O}$.

Definition 1 (Variables). *Class expressions with variables are defined analogously to OWL class expressions but allowing variables at positions of class expressions. The range of a variable can either be a named class, class expression, or any subtype of it as produced by the corresponding rule in OWL. Properties, individuals, and constants are defined analogously. A* variable *is either a* single variable *`?x` that can be bound to a single expression (according to its range), or a* multi-variable *`??X` which consists of a set of single variables where each of them is referable via an index. By default, different index values denote different single variables within the set.*

Multi-variables allow to define an arbitrary number of expressions without loosing reference to the single variable if needed. Let `??X` be a multi-variable whose range are classes. Then `ObjectIntersectionOf(??X)` defines an intersection of an arbitrary number of classes. In contrast to the MPL statement `createIntersection(?x.VALUES)`, each `?X(i)` is referable in other condition which have an impact on the variable binding. For instance, one could state that `?X(i)` and `?X(j)` are used in `SubClassOf` axioms: `SubClassOf(?X(i) A)` and `SubClassOf(B ?X(j))` to define further constraints. By default, `?X(i)` and `?X(j)` have different bindings if there is no constraint specifying that both should be equal.

Definition 2 (Variablized Axioms). *Given an OWL ontology, axioms are defined analogously to OWL axioms according to the axiom rules, but also allow variables at position of class (resp. property) expressions (resp. indivduals). Axioms with variables are called* variablized axioms. *The semantic of using multi-variables in axioms is the following: for each bounded `?X(i)` a new re-incarnation of the axiom is produced.*

Given an ontology containing the following axioms `SubClassOf(A B)`, `SubClassOf(A C)`, `SubClassOf(A D)`, `SubClassOf(C B)`, `SubClassOf(B A)`. Let `?X` be a class variable occurring in the axiom `SubClassOf(?X B)`. Then `?X` can be bound to either `A` or `C` and we get the bounded axioms `SubClassOf(A B)` or `SubClassOf(C B)`. Let `??X` be a multi-variable (range bounded to classes). Given the axiom `SubClassOf(??X B)`, `??X` would be bound to the set $\{A, B\}$ and we get the axioms `SubClassOf(A B)` and `SubClassOf(C B)`.

Definition 3 (Constraints). *Every OWL axiom and variablized axiom according to the Definition 2 is a constraint. Let* `?x` *and* `?y` *be variables, and* `??Z` *a multi-variable. Then the following statements are also constraints:* `?x != ?y`, `?x = ?y` *(in- resp. equivalence of variable bindings),* `MinCount(??Z) = n`, `MaxCount(??Z) = n`, *resp.* `ExactlyCount(??Z) = n` *(specifying a minimum, maximum or exact number of bounded variables* `?Z(0)`*,...,* `?Z(n)` *in* `??Z`*).*

Definition 4 (Abstract OWL Pattern).

```
IRI iri
<ACTION>
LET variables
WHERE EXPLICIT constraints
WHERE IMPLICIT constraints
TYPE type
MESSAGE message
DOCUMENTATION documentation
```

is an abstract pattern where iri *is a unique identifier,* $\langle ACTION \rangle$ *is an action's placeholder,* variables *is a comma-separated set of variables containing all variables that are used in the pattern, and* constraints *is a comma-separated set of (conjunctively connected) constraints according to Definition 3. Axioms mentioned in the* WHERE *statement are either interpreted as explicit or implicit axioms.* type *denotes the type of the pattern (as explained in the following),* documentation *is a human-readable documentation of the pattern, and* message *is a message that can be displayed in tools.*

Following the type categorization of patterns given in [4] and `ontologydesignpatterns.org` we use the given types also in our OWL patterns. Therefore we extended the meta-ontology about types with a new one for ontology lint patterns. It depends on the pattern's type which $\langle ACTION \rangle$ is provided. By default

```
ADD axioms
REMOVE axioms
```

where axioms is a set of axioms according to the Definition 2 are defined. For instance, ODPs typically only use the ADD statement whereas re-factoring patterns also provide a remove section, and lint patterns might inform the user only (they do not provide any action).

Ontology patterns are often based on either entities or axioms defined in other patterns: combining patterns or re-using patterns is a key building block of pattern creation. We therefore introduce the following statement group

```
FROM iris
AS constraints
```

where *iris* is a set of comma-separated pattern identifiers (IRIs) and *constraints* is a set of comma-separated constraints referring to variables in the patterns identified by their IRI and variables of the surrounded pattern. To distinguished structural equivalent variables from different patterns it is always required that these variables are prefixed with the IRI. All axioms and variables contained in the patterns mentioned in *iris* are always part of the surrounding patterns. This allows, for instance, to combine two patterns and establish equivalence between variables with help of the `AS` statement.

In the following some very short examples of using our additional language constructs are given. An ontology lint pattern that finds redundant transitive property axioms because of the transitivity of the super-property can be formulated as follows:

```
IRI http://www.derivo.de/patterns/WOP2009/RedundantTransitiveAxiom
LET ?X - ObjectProperty, ?Y - ObjectProperty
REMOVE TransitiveObjectProperty(?X)
WHERE EXPLICIT SubObjectProperty(?X ?Y),
               TransitveObjectProperty(?X),
               TransitiveObjectProperty(?Y)
```

A *cyclic class definition* can easily be defined by using variablized class expressions as follows:

```
IRI http://www.derivo.de/patterns/WOP2009/CyclicExplicitClasses
ADD EquivalentClasses ??X
REMOVE SubClassOf (??X ??Y) - ?X(i) = ?Y(i+1), ?X(0) = ?Y(n)
LET ?X - ClassExpression
WHERE EXPLICIT SubClassOf(??X ??Y) - ?X(i) = ?Y(i+1), ?X(0) = ?Y(n)
DOCUMENTATION Cyclic class definitions are replaced
  by an EquivalentClassAxiom
```

An ontology lint patterns discovering at least 5 super-classes (borrowed from Pellint's lint collection) can be expressed as follows:

```
IRI http://www.derivo.de/patterns/WOP2009/AtLeastFiveSuperClasses
LET ?X - ClassExpression, ?Y - ClassExpression
WHERE EXPLICIT SubClassOf(?X ??Y), MinCount(??Y) = 5
```

4 Implementation

We have integrated the proposed language constructs in a pattern language as a supplement to the OWL API [11]. In order to apply constraints on variable bindings (esp. in the case of multi-variables) we implemented methods known from CSP. For a pro-actively user-support during ontology engineering we have integrated the language within the successor of our graphical-based ontology authoring framework OntoTrack [12]. Here, the user can instantiate patterns by selecting them from catalogues and filling variables either by selecting corresponding OWL object or by drag-'n-drop them to the bound variable. Constraints of a pattern instantiation will be immediately checked and monitored during ontology engineering in order to avoid undesired constraint violations. For instance, Figure 4 shows a screen capture of the graphical ontology view within our application presenting the hierarchy of classes and properties in a geographical domain. Here, a class labeled `Partition` has been introduced by

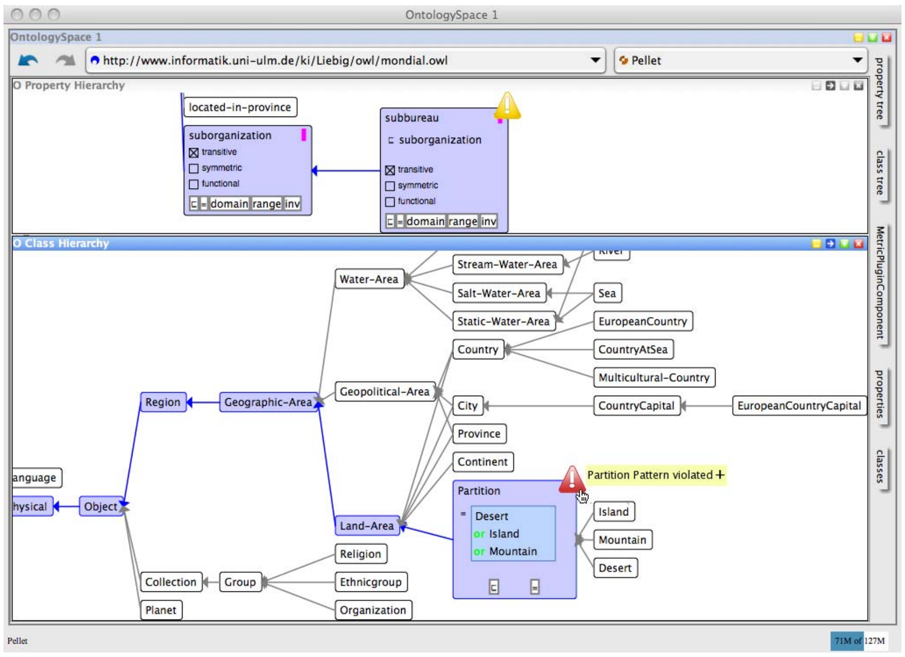


Fig. 1. Screen capture showing class hierarchy with pattern instantiation warnings in our OntoTrack-based ontology authoring framework.

applying the partition pattern. After removing the `DisjointClasses` axioms the user's intention of defining a partition is violated and therefore our framework immediately presents a hint to the user as shown as symbol in the right upper corner of the class box. Hovering over the class with the mouse pointer displays which pattern has been violated. Moreover, when clicking on the symbol one gets further information about the reason(s) for the violation.

Since ontology lint are handled in the same way as ODPs the same mechanism is used to monitor the ontology in order to find lints as a background task. As shown in the upper part of Figure 4, a redundant transitive axiom for a property that is a sub-property of a transitive property has been found.

5 Conclusion and Future Work

In this paper we presented some shortcomings of ontology pattern languages such as the OPPL-based ontology language and motivated to broaden the term ontology pattern by looking at ontology analysis methods which encode ontology patterns in program structures. A declarative pattern approach could benefit from automatic detection of pattern instantiations, monitoring of patterns violations as well as interoperability of patterns. To overcome the presented limitations we briefly introduced the building blocks of a universal pattern language and gave an overview of the implementation and integration in an ontology authoring environment. Future work will be concerned with further analysis of ontology patterns related areas such as ontology lints as well as ontology extractions to include language constructs to better support these patterns in order to get to our aim to build a universal pattern language.

References

1. Fiedland, N., Allen, P., Witbrock, M., Matthews, G., Salay, N., Miraglia, P., Angele, J., Stab, S., Israel, D., Chaudhri, V., Porter, B., Barker, K., Clark, P.: Towards a Quantitative, Plattform-Independent Analysis of Knowledge Systems. In: Proc. of the 9th KR Conference, Whistler, BC, Canada (2004) 507–514
2. Rector, A.L., Drummond, N., Horridge, M., Rogers, J., Knublauch, H., Stevens, R., Wang, H., Wroe, C.: OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns. In: Proc. of the 14th Int. Conf. on Engineering Knowledge (EKAW). (2004) 63–81
3. Clark, P., Thompson, J., Porter, B.W.: Knowledge Patterns. In: Handbook on Ontologies. (2004) 191–208
4. Gangemi, A.: Ontology Design Patterns for Semantic Web Content. In: Proc. of 4th Int. Semantic Web Conference (ISWC 2005). (2005) 262–276
5. Presutti, V., Gangemi, A.: Content Ontology Design Patterns as Practical Building Blocks for Web Ontologies. In: Proc. of the 27th Int. Conf. on Conceptual Modeling (ER). (2008) 128–141
6. Presutti, V., Gangemi, A., del Carmen Suarez-Figueroa, M.: Library of Design Patterns for Collaborative Development of Networked Ontologies. Deliverable D.2.5.1 NeOn project (2007)
7. Iannone, L., Rector, A.L., Stevens, R.: Embedding Knowledge Patterns into OWL. In: Proc. of the 6th European Semantic Web Conference (ESWC 2009). 218–232
8. Motik, B., Patel-Schneider, P.F., Parsia, B.: OWL 2 Structural Specification and Functional-Style Syntax. W3C Candidate Recommendation 11 June 2009
9. Lin, H., Sirin, E.: Pellint – A Performance Lint Tool for Pellet. In: Proc. of the 5th OWLED Workshop on OWL: Experiences and Directions. (2008)
10. Kubias, A., Schenk, S., Staab, S., Pan, J.Z.: OWL SAIQL – An OWL DL Query Language for Ontology Extraction. In: Proc. of the OWLED 2007 Workshop on OWL: Experiences and Directions (OWLED'07). (2007)
11. Horridge, M., Bechhofer, S., Noppens, O.: Igniting the OWL 1.1 Touch Paper: The OWL API. In: Proc. of the OWLED 2007 Workshop on OWL: Experiences and Directions (OWLED'07). (2007)
12. Liebig, T., Noppens, O.: OntoTrack: A Semantic Approach for Ontology Authoring. Journal of Web Semantics **3**(2) (2005)

The `newsEvents` Ontology[⋆]
An Ontology for Describing Business Events

Uta Lösch and Nadejda Nikitina

AIFB, Universität Karlsruhe, Germany
`{uhe,nani}@aifb.uni-karlsruhe.de`

Abstract. In the broader context of the development of an ontology-based new event detection system, we are developing the `newsEvents` ontology which allows modeling business events, the affected entities and relations between them. This paper presents requirements for and a first version of this ontology. A pattern-based approach to the design of the ontology was taken. Thereby a new useful pattern - the `EventRole` pattern - was identified and specified.

1 Introduction

The analysis of news and the estimation of their market impact is an important issue for traders in financial markets. As the amount of news that is made available is huge, at least a partial automation of the analysis process is desirable. The goal is to filter relevant news, to provide an aggreggated view on their content and to thus enable the users to only read those news that are really relevant for them. The relevance of a piece of news depends on their relevance for the traders' interests (i.e. is it about an entity that the trader is interested in) and its novelty. These two aspects are rather independent of each other and can thus be approached separately.

The idea of our work is to study which benefits ontologies may offer for the two described aspects. In order to study this problem, an ontology is needed which allows to describe relevant entities and events in the business news context. To the best of our knowledge no such ontology exists. We therefore decided to build one for the use in the kind of system we have described above. In this paper, the `newsEvents` ontology, which was developed for that purpose, will be presented.

In the task of relating news to a user's interest ontology-based annotations may provide a condensed representation of a text's content. A query on these annotations can be used to describe the user's information need. A similar system has been described by [4].

[⋆] This work has been supported by the Deutsche Forschungsgemeinschaft in the scope of the graduate school of Information Management and Market Engineering and by the EU through in the IST project NeOn (IST-2006-027595, http://www.neon-project.org/.

For the task of discovering new events, to which we refer as the new event detection problem, the annotation of texts with ontology entities allows for assessing their content and relating news to each other based on these annotations. The introduction of additional features that are based on annotations in the clustering task seems promising as it allows a more specific analysis of the content and as it enables a better distinction between similar events.

More precisely, annotations will be obtained from a state-of-the-art tool (OpenCalais[1] has been chosen for this task). To include the annotations in the clustering task a similarity measure that takes into account the similarity between annotations will be introduced. However, the provided annotations do not offer much structure, only a list of annotation types and some properties for each type are defined. A taxonomy on the annotation types is not defined. The `newsEvents` ontology will allow for the description of the current state of the domain and of events as reported in news. The ontology provides a formalisation of this kind of information that includes more background knowledge than OpenCalais' annotations.

The rest of the paper is structured as follows: Section 2 describes related work, section 3 describes our requirements for the ontology, section 4 describes the ontology we built, section 5 describes the newly defined EventRole pattern, before we conclude in section 6.

2 Related Work

While to the best of our knowledge no ontology exists which aims at describing companies, their relations among each other and the most important events, there is a number of models which address part of the scope of the intended ontology.

OpenCalais has defined a schema which is used for the annotation of news texts, in which events and entities are annotated. While the scope is quite similar to the scope of the `newsEvents` ontology, it only defines a list of annotation types and, for the complex annotations, the slots that have to be filled. The goal when defining the `newsEvents` ontology was to provide a model of the domain that provides more background knowledge on event and entity types and that is able to relate different annotations to each other.

The classification of news according to their content is very important for news providers. Therefore, various annotation languages have been defined for this purpose: while IIM[2] is today only used for annotating photos, its successors NITF[3] and NewsML[4] are still in use. However, their primary concern is to have a standardized format for providing and exchanging news.

[1] http://www.opencalais.com
[2] The Information Interchange Model, http://www.iptc.org/IIM/
[3] The News Industry Text Format, http://www.nitf.org
[4] The News Meta Language, http://www.newsml.org

The most important entity types (and also some events) are also defined in top-level ontologies such as Cyc[5] or Proton[6]. However, these ontologies are not detailed enough for our use case.

There is a number of ontologies in the finance domain. The LSDIS_Finance ontology[7] and the dip Ontology [1] both describe actors and products in the stock markets. However, these ontologies do not include any description of events.

3 Competency questions

As described above, the `newsEvents`[8] ontology shall be used for describing events which are relevant in a business context and for assessing similarity between different events. It will also be used for modeling the current information that is available about the economy and for determining whether a user might be interested in a new event that is reported. In the future, it may also be considered whether there are recurring patterns of series of events where the next events may be anticipated. It may also be useful for studying the impact of events on financial markets.

As the latter use cases may only be interesting in the future and are not the main motivation for developing the ontology, no special attention has been paid to them when developing the first version of the ontology. It will however be extended accordingly in the future.

When developing the ontology, the goal was to have as little modeling efforts as possible. Therefore, a pattern-based approach to its design was taken.

The first step in the development of the ontology was the definition of a set of competency questions. Competency questions are questions that the ontology should be able to answer. Ideally, the ontology should be able to answer all and only the competency questions - no superfluous additional information should be included in the model. For a more detailed discussion of competency questions see [3]. The following questions were defined:

- Related to the history of an event
  - *Is there any information on a specific event already available?* This question serves to determine whether a specific event has already been reported on. If this is not the case, the event is definitely new - information about it has to be integrated into the knowledge base. Additionally, its novelty will be high.
  - *In which order and in which timeframe was information on a specific event published?* The purpose of this question is to determine the development of the information that is made available on an event. This will help to study the development of the history of a single event. It may

[5] The Cyc knowledge base, http://www.cyc.com
[6] The proton ontology, http://proton.semanticweb.org
[7] http://lsdis.cs.uga.edu/projects/meteor-s/wsdl-s/ontologies/LSDIS_FInance.owl
[8] The ontology is available at `http://www.aifb.uni-karlsruhe.de/WBS/uhe/ontologies/newsEvents.owl`

however also help to determine patterns of how event histories look like and thus to anticipate new information.

- Related to the assessment of similarity
  - *How similar are two entities?* The similarity of two entities may be defined through their properties, like entity type, name, position, location, industry, etc. The assessment of entity similarity is needed for the assessment of event similarity, but it will also help to identify entities on which similar events may have similar impact, entities which may have a similar history, or entities that may be interesting for a user based on the interests he has stated explicitely.
  - *How similar are two events?* The similarity of two events is needed for deciding whether two events are in fact the same. Furthermore, if two events are very similar, they may be interesting for the same users and the history of one event may allow for anticipating future developments concerning the other event.
- Related to relations between entities
- *Which products does a company produce? Which industry does a company belong to? Where is a company located?* Although these questions may seem very heterogenuous at first sight, they all serve for finding entities and events that are related to a user's interests. For example, if he is interested in cars, he may be interested in all news that relate to car manufacturing companies.

4 The `newsEvents` ontology

Based on the requirements presented in the previous section, we have developed a first version of the `newsEvents` ontology. It describes various entity types that are relevant in business news as well as important events, in which described entities may be involved.

For the development of the ontology we tried to follow a pattern-based approach. Especially, we found the use of content design patterns helpful. These are small ontologies - typically consisting of two to ten classes and relations among them, which describe typical modeling problems arising in different domains. These patterns were proposed by Gangemi and Presutti [2] [6]. The goal is facilitate the design of the ontology by providing building blocks which can be composed, specialized or instantiated and thus be adapted to a specific domain.

As OpenCalais is used for obtaining annotations, the annotation types OpenCalais defines were taken as a starting point for the definition of the concepts that should be represented in the ontology.

After the most important concept types were chosen, the next step consisted in identifying the patterns that could be reused for defining the `newsEvents` ontology. Using the patterns described below, the description of most of the event and entity types turned out to be rather straight forward. As the whole ontology is quite big and complex, we only show parts of it to illustrate the use of design patterns:

- *TimeIndexedParticipation.* This pattern describes the involvement of objects in an event at a certain point in time. This pattern can be reused for the description of events like `Acquisition`, `Merger`, `IPO`, etc. The various event types could be described as specialisations of the concept `Event`, its participants could be defined through the use of specialisations of the `hasParticipant` property and of the `Participant` class, which are defined in the *Participation* pattern. The association of an event with the time at which it is happening is possible through the `TimeIndexedParticipation` concept, which relates events and their participants to a time component.
- *Situation.* This very general pattern can be used for the description of complex relations, like `CompanyTicker`. The latter describes a company which is traded at a specific stock exchange and has a specific ticker symbol there. This could be modeled by defining the concept `CompanyTicker` as subconcept of `Situation`.
- *Place.* This pattern defines how places and locations should be described in an ontology. The `newsEvents` ontology describes various concepts, especially event types, that happen at a certain location. The place pattern allows to model these locations and relations among them.
- *ObjectRole.* This pattern allows to describe the different roles an entity of a specific type may play. This pattern is useful for defining the different roles an entity may play in an event or in a relation among entities. For example, an `Acquisition` describes the event of one company acquiring another one. Obviously, there are two entities of the same type, i.e. two companies involved. One company is the acquiring company, the other one the acquired company - the companies thus play different roles in the event. To address the problem of roles in events we have defined the `EventRole` pattern, which will be described in section 5.

A taxonomy of event and entity types has been defined, which is needed to allow for the calculation of similarities between different event types.

To distinguish between the different type of events, we introduced additional classes for describing events that have a similar meaning. For this purpose classes like `CompanyCollaboration`, `LegalIssue`, or `StockEvent` were introduced. None of these classes defines concrete events. It is merely used as a grouping element. For example, the class `CompanyCollaboration` has the subclasses `Alliance`, `BusinessRelation`, `JointVenture`, and `Merger`. Each of these events describes a way in which two companies may choose to collaborate.

Entities were also grouped in a hierarchy. Here, grouping was chosen according to which roles an entity may play in events. Most importantly, we created a class `LegalEntity` which describes both legal and natural persons, i.e. every entity that may be an actor in an event.

5 The `EventRole` pattern

When building the ontology, design patterns could be used to address most of the modeling problems that were encountered. However, no obvious solution seemed to be available for the case where two entities of the same type are involved in the same event. This is however a recurring situation. Examples in our use case are acquired and acquiring company, provider and customer, plaintiff and suedEntity, etc. The occurrence of this kind of situation is not restricted to the business domain: an obvious example in an everyday context is a visit, where there is a visitor and a person that is visited.

The problem can be solved by composing the patterns *Participation* and *ObjectRole*. The idea is that the participants in an event are not described by their entity type, but by the role they are taking in the event. The resulting pattern is depicted in figure 1.

A new class `EventRole` has been introduced which serves as connector between the two original patterns. The `EventRole` class is used to describe the role that an entity plays in an event. Therefore, it is a specialization of the `Role` and the `Object` class. For each role an entity can play in an event, this class should be specialized. Additionally, the property `hasParticipant` should be specialized for each of the entity's roles and the thus defined properties should be declared disjoint. Thus, each object can only have one role in a given event.

To show how the pattern can be adapted to a specific use case, consider the definition of an acquisition event. An acquisition is the event of one company, the acquiring company, buying another one, the acquired company. Two companies are involved in this event, but it makes a huge difference for company A if it is acquired by company B or whether it acquires company B. Therefore, the event roles `AcquiringCompany` and `AcquiredCompany` have been defined. They both are roles of `Company`. The class `Acquisition` is then defined as a subclass of `Event`. Additionally, we defined the restriction that each acquisition has at least one acquiring company and at least one acquired company, but at most 2 participants. The properties `hasAcquiringCompany` and `hasAcquiredCompany` were specified as subproperties of the `hasParticipant` property. The two properties are defined as being disjoint.

6 Conclusion

The paper presented an ontology which can be used to describe events and entities in a business news context. Patterns have proven to be very useful for the design of the ontology as they directly solve many of the modeling issues that were encountered in the engineering process.

One of the recurring problems encountered while modeling the ontology was the description of the roles entities take in an event. The `EventRole` pattern has been proposed in order to solve this issue.

In the future, the proposed ontology will be refined and extended such that relations between events (especially causal relationships) may be included in the ontology.

Additionally, procedural knowledge (following our proposal in [5]) will be associated with the ontology, such that automatic updates of an entity's state after being affected by an event become possible. The ontology, which can then

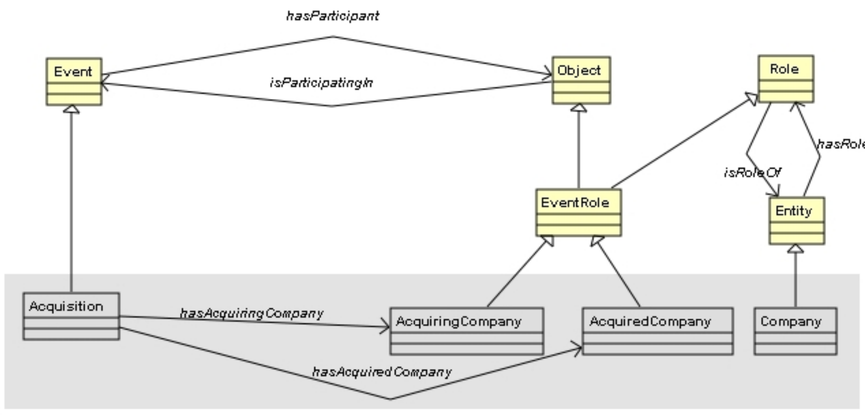


Fig. 1. The role event pattern and its adaptation for modeling acquisitions

automatically adapt to changes in the domain, will then be ready to be used in the new event detection process.

References

1. S. L. Alonso, J. L. Bas, S. Bellido, J. Contreras, R. Benjamins, and J. M. Gomez. Deliverable 10.7 - financial ontology. Technical report.
2. A. Gangemi. Ontology design patterns for semantic web content. *The Semantic Web ISWC 2005*, pages 262–276, 2005.
3. M. Gruninger and M. Fox. The role of competency questions in enterprise engineering, 1994.
4. C. Halaschek-Wiener and J. Hendler. Toward expressive syndication on the web. In *Proc. of the 16 th International World Wide Web Conference(WWW 2007)*, 2007.
5. U. Lösch, S. Rudolph, D. Vrandecic, and R. Studer. Tempus fugit - towards an ontology update language. In *6th European Semantic Web Conference (ESWC 09)*. Springer-Verlag, 2009.
6. V. Presutti and A. Gangemi. Content ontology design patterns as practical building blocks for web ontologies. In *ER '08: Proceedings of the 27th International Conference on Conceptual Modeling*, pages 128–141, Berlin, Heidelberg, 2008. Springer-Verlag.