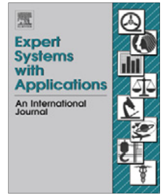
Expert Systems with Applications 38 (2011) 7633–7646

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa



SPARSE: A symptom-based antipattern retrieval knowledge-based system using Semantic Web technologies

Dimitrios L. Settas *, Georgios Meditskos, Ioannis G. Stamelos, Nick Bassiliades

Department of Informatics, Aristotle University, 54124 Thessaloniki, Greece

ARTICLE INFO

Keywords:
Antipatterns
Symptom-based retrieval
OWL ontology
Production rules
Objects

ABSTRACT

Antipatterns provide information on commonly occurring solutions to problems that generate negative consequences. The number of software project management antipatterns that appears in the literature and the Web increases to the extent that makes using antipatterns problematic. Furthermore, antipatterns are usually inter-related and rarely appear in isolation. As a result, detecting which antipatterns exist in a software project is a challenging task which requires expert knowledge. This paper proposes SPARSE, an OWL ontology based knowledge-based system that aims to assist software project managers in the antipattern detection process. The antipattern ontology documents antipatterns and how they are related with other antipatterns through their causes, symptoms and consequences. The semantic relationships that derive from the antipattern definitions are determined using the Pellet DL reasoner and they are transformed into the COOL language of the CLIPS production rule engine. The purpose of this transformation is to create a compact representation of the antipattern knowledge, enabling a set of object-oriented CLIPS production rules to run and retrieve antipatterns relevant to some initial symptoms. SPARSE is exemplified through 31 OWL ontology antipattern instances of software development antipatterns that appear on the Web.

© 2010 Elsevier Ltd. All rights reserved.

1. Introduction

A successful way for solving newly encountered problems is by subconsciously applying a solution that has been previously applied successfully to a similar problem (Laplante & Neil, 2006). This approach requires experience through successes and failures and expertise in order to identify such problem and solution pairs and then document them as patterns (Alexander, 1979). In software engineering, design patterns describe a recurring problem and its solution. Design patterns can reduce development time by providing proven development paradigms (Gamma, Helm, Johnson, & Vlissides, 1994). By documenting design patterns, software developers and architects can reuse design patterns in order to prevent subtle issues that can cause major problems. The usefulness of studying the successful ways of solving problems has been well proved in software engineering by the valuable concept of design patterns. However, design patterns focus on coding and architectural issues and can not be used in order to document and share software project management knowledge. In software project management, it is much easier to identify a defective situation than to implement a solution to a given problem.

The primary cause of software project failure is the lack of appropriate software project management. In project management, commonly occurring repeated bad practices are referred to as antipatterns. Antipatterns are the latest generation of design pattern research. Software project success or failure can be attributed to the incorrect handling of one or more project variables: people, technology and process (Brown, McCormick, & Thomas, 2000). Antipatterns have been proposed as mechanisms for managing these project variables. These mechanisms describe how to arrive at a good (refactored) solution from a fallacious solution that has negative consequences (Brown et al., 2000). As a result, managers are able to avoid the specious solution(s) that have resulted in finding themselves in an unhealthy situation for the organization and the individual (Laplante & Neil, 2006). Using antipatterns, a software project can be managed more effectively by bringing insight into the causes, symptoms, consequences, and by providing successful repeatable solutions (Brown, Malveau, McCormick, & Mowbray, 1998).

The number of antipatterns that appears documented in the literature (Brown et al., 1998, 2000; Laplante & Neil, 2006) and the Web (Pattern Community Antipattern Catalogue, 2010;

* Corresponding author. Tel.: +30 2310 991927; fax: +30 2310 998419.
E-mail addresses: dsettas@csd.auth.gr (D.L. Settas), gmeditsk@csd.auth.gr (G. Meditskos), stamelos@csd.auth.gr (I.G. Stamelos), nbassili@csd.auth.gr (N. Bassiliades).

0957-4174/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2010.12.097

Software Project Antipattern-Pardon, 2010; Software Project Management Antipattern Blog, 2010; Wikipedia Antipatterns, 2010) is increasing. Furthermore, there are significant challenges for software project managers in using antipatterns during a software project, because antipatterns are usually related to other antipatterns and rarely appear in isolation (Brown et al., 2000; McCormick, 1999). Documenting antipatterns is also a difficult task. Managers need to decide which template to use from a variety of antipattern templates (Brown et al., 2000; Laplante & Neil, 2006) available. Furthermore, the antipatterns that appear on the Web are documented in a more unofficial manner that usually do not conform to templates. This introduces difficulty in understanding antipatterns and identifying how such antipatterns are inter-related with other antipatterns, which are properly documented.

Based on formalisms (Settas, Bibi, Sfetsos, Stamelos, & Gerogiannis, 2006; Settas & Stamelos, 2007a) and models (Settas, Sowe, & Stamelos, 2009; Settas & Stamelos, 2008) tools that can assist managers in carrying out project management related tasks using antipatterns can be developed. In spite of numerous software project management antipattern formalisms and methodologies proposed, the computer-mediated dissemination of knowledge encoded in software project management antipatterns still remains an open issue. As a result, managing software projects using antipatterns is not popular in the practitioners' community. For antipatterns to become a widespread practice, a knowledge-based system that assists managers in detecting antipatterns is required.

In this paper, we describe SPARSE (symptom-based antipattern retrieval system using Semantic Web technologies), our approach to assist the antipattern detection process according to the symptoms that are visible during a software project, by introducing Semantic Web technologies and tools. SPARSE is an ontology supported knowledge-based system that is based on the DL knowledge representation formalism, providing an "upper-level" OWL ontology for describing antipattern resources and their relationships, in terms of symptoms, causes and consequences. Based on the inferencing capabilities of the Pellet DL reasoner (Sirin, Parsia, Grau, Kalyanpur, & Katz, 2007), the ontological knowledge is imported as an object-oriented (OO) model in the COOL language of the CLIPS production rule engine (Riley, 1991), where OO production rules derive conclusions and assist project managers to determine antipatterns. The antipattern OWL ontology has been populated with data on 31 antipatterns that exist on the Web (Pattern Community Antipattern Catalogue, 2010; Software Project Management Antipattern Blog, 2010; Software Project Antipattern-Pardon, 2010; Wikipedia Antipatterns, 2010).

The main contribution of SPARSE is that it offers an extensible framework under which antipatterns may be semantically defined, managed and discovered, using already established Semantic Web standards and tools, such as OWL ontologies and DL reasoners. In that way, SPARSE can propose directly related but also semantically retrieved antipatterns, according to a list of visible symptoms that may exist in a software project.

This paper is divided in six sections, which are organized as follows: Section 2 describes the background, the related work and the literature review used in our research. Section 3 describes the antipattern ontology, data collection and SWRL rules used in SPARSE. Section 4 describes the rule program of SPARSE. This includes a description of the COOL model, validation rules, the object-oriented mapping procedure and how SPARSE searches for antipatterns. In Section 5, the process of creating new instances of the antipattern ontology and using SPARSE to propose directly related but also semantically retrieved antipatterns, is described. Finally, in Section 6, the findings are summarized, future work is proposed and conclusions are drawn.

2. Background and related work

2.1. Software antipatterns

Antipatterns provide real-world experience in recognizing recurring problems in software development and provide the tools to enable developers, architects and managers to identify these problems and determine their underlying causes (Brown et al., 1998). Another benefit of using antipatterns in software development is that these mechanisms provide a common vocabulary of expressing and identifying problems and discussing solutions. There exist different kinds of antipatterns (Brown et al., 1998): Software Development, Software Architecture and Software Project Management antipatterns. Laplante and Neil (2006) have also documented environmental antipatterns which are the result of misguided corporate strategy or uncontrolled socio-political forces.

Software projects not only fail because of their inherent complexity, but also because of poor project management. Ineffective software project management is a major factor that contributes to software project failure. Software project management antipatterns are caused by the lack of understanding of how to manage software development projects (Brown et al., 2000). These mechanisms define a bad project management practice through a description of its causes, symptoms and consequences. The refactored solution is what makes antipattern beneficial by describing how to avoid the antipattern as well as how to resolve the problems caused by the antipattern.

Various templates can be used in order to document antipatterns. Brown et al. (1998) has proposed several templates that can be used to document antipatterns according to the level of detail that the antipattern author wishes to use. The full antipattern template contains a number of optional and required sections. Laplante's informal presentation style is a more compact template that focuses on identifying the dysfunction and remedies for all those involved (Laplante & Neil, 2006) (see Table 1).

Antipatterns can appear isolated but can also be related with other antipatterns. The later type is referred to as interacting antipatterns (Brown et al., 2000) and is evident when a project management antipattern causes a software development antipattern or an architecture antipattern. SPARSE focuses on software project management antipatterns but takes into account the fact that these antipatterns can be related with other types of antipatterns. Antipatterns can be related through the attributes of Table 2.

Table 1
Antipattern structure.

Name	A short name that conveys the antipattern's meaning
Central concept	The short synopsis of the antipattern
Dysfunction	The problems and symptoms of the current practice
Explanation	The expanded explanation including causes and consequences
Band-aid	A short term coping strategy for those who cannot refactor it
Self-repair	The first step for someone perpetuating the antipattern
Refactoring	What changes should be done to remedy the situation
Identification	A list of questions for diagnosis of the antipattern

Table 2
Attributes relating software project management antipatterns.

Causes	A list which identifies the causes of this antipattern
Symptoms	A list which contains the visible symptoms of this antipattern
Consequences	A list which contains the consequences that result from this antipattern

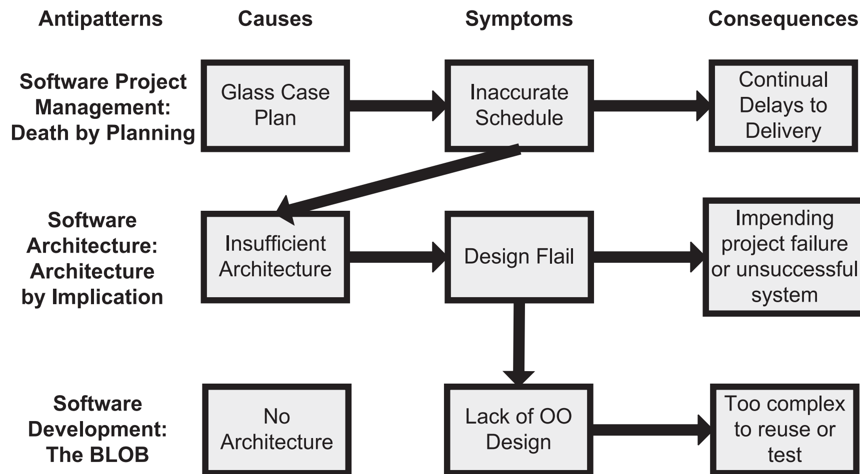


Fig. 1. Example dataset of how antipatterns can be related through their attributes.

Fig. 1 illustrates an example relationship between three different antipatterns, through their causes, symptoms and consequences.

In this example, the "Death by Planning" project management antipattern interacts with the other two antipatterns through the symptom "Inaccurate schedule". SPARSE operates with a symptom based approach. If a project manager has selected the "Inaccurate schedule" symptom, SPARSE can report the directly matching "Death by Planning" antipattern but also detects the other two related antipatterns of the example and other semantically related antipatterns. The following sections describe the underlying technology of SPARSE and the architecture of the proposed knowledge-based system.

2.2. OWL ontologies and description logic reasoning

The Web Ontology Language (OWL)[1] is the W3C recommendation for creating and sharing ontologies in the Web and its theoretical background is based on the Description Logic (DL) (Baader, 2003) knowledge representation formalism, a subset of predicate logic. For example, OWL DL and OWL Lite are based on $\mathcal{SHOIQ}(\mathcal{D})$ and $\mathcal{SHIN}(\mathcal{D})$, respectively. An OWL ontology is actually a finite set of DL axioms, such as axioms about concepts, concept inclusions ($C \sqsubseteq D$), role definitions, role inclusions ($R \sqsubseteq S$), concept assertions ($C(a)$) and role assertions ($R(a, b)$), where C, D are concepts, R, S are roles and a, b are instances. These axioms can be divided into two categories, namely the TBox and the ABox of the ontology. The TBox consists of the concept and role definitions/inclusions, and the ABox of concept and role assertions. Intuitively, the TBox refers to the schema of the ontology, whereas the ABox to the instances.

The formal semantics of the OWL language enable the application of reasoning techniques in order to make logical derivations. These derivations are performed by the reasoners, which are systems able to handle and apply the semantics of the ontology language. There are several OWL DL reasoners (Haarslev & Möller, 2003; Sirin et al., 2007; Tsarkov & Horrocks, 2006) that implement DL algorithms, such as the tableaux-based algorithms (Baader & Sattler, 2001). Assuming that $\mathcal{KB} = (\mathcal{T}, \mathcal{A})$ is a DL knowledge base with the TBox $\mathcal{T}$ and the ABox $\mathcal{A}$, basic DL reasoning problems include:

- *Concept equivalence.* Two concepts C and D are equivalent in $\mathcal{T}$ if and only if $\mathcal{T} \models C \sqsubseteq D$ and $\mathcal{T} \models D \sqsubseteq C$.
- *Concept subsumption.* A concept C is subsumed by D in $\mathcal{T}$ if and only if $C \sqcap \neg D$ is not satisfiable in $\mathcal{T}$.
- *Satisfiability.* A concept C is satisfiable in $\mathcal{T}$ if and only if C is not subsumed by $\bot$ (`owl:Nothing`) or $(\mathcal{T}, \{i : C\})$ is consistent.
- *Realization.* is an instance of C according to the $\mathcal{KB}$ if and only if $(\mathcal{T}, \mathcal{A} \cup \{i : \neg C\})$ is not consistent.

2.3. Semantic Web and rule-based systems

Rule-based systems have been extensively used in several applications and domains, such as e-commerce, personalization, games, businesses and academia. The ability of manipulating and using semantically annotated information into rule systems is vital for both businesses and the successful proliferation of Semantic Web technologies, since it enables the already existing and well-known infrastructure of rule engines to gain access in the new evolution of Semantic Web.

There are practically two approaches for the development of rule-based applications in the Semantic Web, following a *tight* or a *loose* combination of rules and ontologies. In the first approach, the rule engine is used both for reasoning on the ontological knowledge and for running the rule program. Such approaches, known also as homogeneous (Antoniou et al., 2005), treat rule and ontology predicates homogeneously, as a new single logic language. The general idea is that rules can use unary and binary predicates from the ontology (i.e., classes and properties), as well as predicates that occur only in rules (rule predicates).

In the second approach, an external DL reasoner is interfaced to the rule engine. In such architectures, also known as *hybrid* (Antoniou et al., 2005), a modular architecture of two subsystems is followed, each of which deals with a distinct portion of the knowledge base. More specifically, they combine the reasoning capabilities of the DL reasoning paradigm and the rule execution capabilities of a rule engine, separating rule and ontology predicates.

2.4. Motivation

There are three underlying technologies involved in SPARSE: (a) ontologies, through the use of the OWL ontology language, (b) DL reasoners, through the use of the Pellet DL reasoner and (c) production rule engines, through the use of the CLIPS production rule engine. In the following we justify the decision of using these technologies and tools.

[1] http://www.w3.org/TR/owl-features/.

2.4.1. Antipatterns and ontologies

Ontologies have been created by the AI community in order to facilitate knowledge sharing and reuse by interweaving human understanding with machine processability (Van Harmelen, 2003). In the domain of antipatterns, ontologies can be used as a vocabulary of terms for describing the task/domain knowledge of antipatterns. The antipattern ontology contains information of antipatterns and antipattern attributes and describes how these attributes are interconnected in an antipattern and between several related antipatterns. By defining a top level ontology for describing antipatterns, we have the following advantages.

1. Collaborative definition of antipatterns, allowing software project managers to create antipatterns using new ontology instances of causes, symptoms and consequences or using any such attributes that have already been documented. The standard-based and open-world nature of OWL ontologies allow their extension, reuse and merge, creating an antipattern knowledge base that encapsulates different perspectives, according to the project they appear in.
2. Semantic processing of antipatterns, based on DL algorithms (DL reasoners). DL reasoners ensure the semantic consistency of the ontology-based antipatterns, as well as the derivation of any implicit (hidden) knowledge that derives from the antipattern definitions. In SPARSE we have used the Pellet DL reasoner (Sirin et al., 2007) as the underlying reasoning infrastructure.
3. Exploitation of the research that has been done on the combination of rules and ontologies. In that way, we are able to express richer semantic relationships among antipatterns, in a more declarative way. SPARSE is able to incorporate logical consequences expressed as SWRL rules (Horrocks et al., 2004) that are handled by the underlying Pellet DL reasoner.

2.4.2. Antipatterns and rule-based systems

Antipatterns can benefit software managers by catering for the collaborative exchange of software project management knowledge using software tools. Knowledge-based systems are a branch of applied artificial intelligence (AI), and can transfer knowledge from a human to a computer (Liao, 2005). The computer can then make inference and arrive at conclusions in order to provide specific advise to its users (Turban & Aronson, 2001). Different methodologies can be used to develop a knowledge-based system such as rule-based systems, neural networks and fuzzy expert systems (Liao, 2005). SPARSE follows the rule-based approach, using the CLIPS production rule engine and the COOL language in order to build a knowledge based expert system over the OO antipattern model that derives from DL reasoning. Therefore, SPARSE follows the hybrid paradigm.

In SPARSE we make a distinction between the ontology inference rules and the domain rules that are used for deriving conclusions over the ontological knowledge (CLIPS production rules). The former are used at the ontology reasoning level and they are embedded into the DL reasoning procedure in order to infer the appropriate semantic relationships among antipatterns. The latter are used for defining the rule-based applications over the OO model of the extensional ontological knowledge, without altering the ontology itself. In that way we give the opportunity to use an efficient and well-known production rule engine in order to develop the rule-based application of SPARSE over a shared ontology in a hybrid manner.

Object-oriented model. OWL is built upon RDF and RDFS and has the same syntax, the XML-based RDF syntax (Beckett, 2004). A more machine processable syntax is the N-Triples format (Grant & Beckett, 2004), that is a textual format for RDF graphs which stems directly from the RDF/XML syntax. More specifically, N-Triples is a line-oriented format where each triple must be written on a separate line, and consists of a subject, a predicate and an object.

A common way of representing the asserted and inferred OWL ontology axioms in a rule engine is to store the ontology triples in the form of facts. The limitation of the triple-based representation is that it is not able to exploit any form of semantics that could potentially exist in the environment where the mapping will take place. In that way, all the triples should be explicitly stated, following a brute force approach with increased space requirements. We argue that the semantics of an OO environment can be effectively used in order to represent the instance-related DL reasoner axioms in an OO model that embeds the notion of class subsumption transitivity. In SPARSE, instead of defining the CLIPS rule-based application directly over the ontology triples that derive from DL reasoning, we perform a transformation of the derived ontological knowledge of Pellet into the COOL OO language of CLIPS. The purpose of this transformation is twofold:

1. The generated OO knowledge base is more compact than the corresponding triple-based model. This happens since the OO model exploits the environmental class and property semantics, such as class and property inheritance, in contrast to the triple-based model that needs to state explicitly all the relevant relationships. In that way, we are able to group the related information about an antipattern instance in a single resource represented as an object and we can have direct access to all of its properties and values by exploiting the message passing mechanism of CLIPS.
2. The rule-based application of SPARSE consists of OO production rules that match objects and not triple-based facts. We consider the utilization of domain rules as a more intuitive rule programming paradigm than of the triple-based rules, using OO notations and semantics that are well established.

2.5. Related work

The majority of antipattern literature is concerned with documenting new antipatterns. This includes books (Brown et al., 1998, 2000; Laplante & Neil, 2006), conference proceedings and journal papers (Kuranuki & Hiranabe, 2004; Laplante, Hoffman, & Klein, 2007; Krai & Zemlicka, 2007), but also Web sites (Pattern Community Antipattern Catalogue, 2010; Wikipedia Antipatterns, 2010) and blogs (Software Project Management Antipattern Blog, 2010). This provides researchers, field practitioners and academics with a wide variety of antipatterns on all software antipattern types including software project management.

Previous work has proposed a variety of formalisms that can be used in order to represent software project management antipatterns such as Bayesian networks (Settas et al., 2006) and ontologies (Settas & Stamelos, 2007a, 2007b). Methodologies such as the Dependency structure matrix (Settas & Stamelos, 2008) and semantic social networks (Settas et al., 2009) have addressed the issue of ontology similarity and resolved the complexity between inter-related antipatterns. Given two antipattern ontologies, the same entity can be described using different terminology. Therefore, the detection of similar antipattern ontologies is a difficult task. The three layered antipattern semantic social network (Settas et al., 2009) involves the social network, the antipattern ontology network and the concept network. Social Network Analysis (SNA) techniques can be used to assist software project managers in finding similar antipattern ontologies. For this purpose, SNA measures can be extracted from one layer of the semantic social network to another and use this knowledge to infer new links between antipattern ontologies.

The architecture of PROMAISE (Settas & Stamelos, 2007b) provided the framework for the implementation of the expert system using Semantic Web techniques. The PROMAISE intelligent system (Settas & Stamelos, 2007b) has been proposed as a framework to allow an antipattern knowledge base to be created and updated dynamically using the antipattern OWL ontology. The antipattern ontology specified the conceptual structure of the antipattern knowledge base, encoded tacit software project management knowledge into computer understandable form and allowed the sharing and reuse of this knowledge by software tools. Furthermore, the issue of capturing and quantifying uncertainty in the antipattern ontology has been addressed by including the concepts of antipattern BN models and their corresponding OWL ontology in the design of the generic antipattern ontology.

Design patterns have been used successfully in recent years in the software engineering community in order to share knowledge. The issue of providing expert advice to developers regarding the selection of design patterns has been discussed in the literature (de Souza & Ferreira, 2002; Dietrich & Elgar, 2005; Henninger, 2006; Moynihan, Suki, & Fonseca, 2006). Moynihan et al. (2006) proposed the development of a prototype expert system for the selection of design patterns that are used in object-oriented software. The prototype system represents an initial step towards providing an automated solution regarding the design pattern application problem, i.e. leading a designer to a suitable design pattern which is applicable to the problem at hand. Design patterns have also been used as a different approach to help designers promote reuse in rule-based knowledge systems (de Souza & Ferreira, 2002). In this work, design patterns described a reusable architecture for rule-based systems. The aim of these patterns was to constitute a design catalog that can be used by designers to understand and create new rule-based systems, thus promoting reuse in these systems.

Semantic Web technologies have also been applied in the context of design patterns in order to create a distributed repository of patterns that relate problems to solutions through typed relationships that manifest a systems design method (Henninger, 2006). Defining the criteria for pattern languages in a formal medium, such as Description Logic (DL) in OWL, facilitated a degree of utility not afforded in informal representations. In addition to logical inference, the author in Henninger (2006) identified that rules can be applied to the pattern descriptions and relationships to further enrich pattern languages. The Web ontology language (OWL) has been used to formally define design patterns and some related concepts such as pattern participant, pattern refinement, and pattern instance (Dietrich & Elgar, 2005). The resulting prototype Java client accesses the pattern definitions, detects patterns in Java software, and analyzes some scan results.

Despite of the extensive body of literature of design pattern applications on knowledge based systems and Semantic Web technologies, as far as we are aware, the research summarized in this paper represents the first implementation of a knowledge based system that supports software project managers in the antipattern selection process.

3. The antipattern OWL ontology

SPARSE provides an initial top-level antipattern OWL ontology as a starting point for defining antipatterns. The ontology consists of 7 concepts, 21 roles (19 object and 2 datatype roles), 192 individuals and 7 SWRL rules. The current DL expressivity is $\mathcal{ALCHN}(\mathcal{D})$, that is, $\mathcal{ALC}$ with role hierarchies, cardinality restrictions and datatype properties. The ontology can be extended, both with OWL constructs and SWRL rules, according to user requirements. In this section we describe the default semantic capabilities that are provided through the ontology.

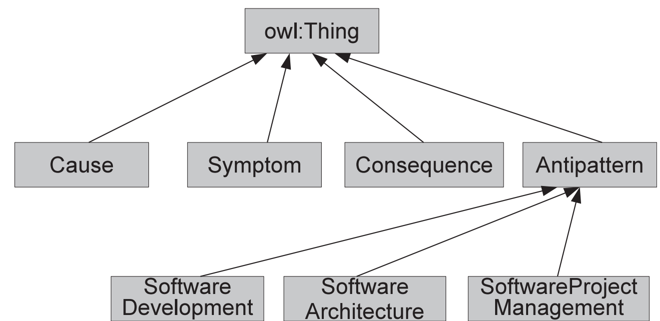


Fig. 2. The concept hierarchy of the antipattern ontology.

3.1. Concept definitions

The antipattern ontology consists of seven concepts whose hierarchical relationships are depicted in Fig. 2. In addition to the intuitive `Antipattern` concept, we have included three more concepts as subclasses of the `Antipattern` concept in order to demonstrate the way the ontology hierarchy can be extended with custom concepts according to managers' needs. In the rest of this section we analyze the concept definitions and semantics using the DL syntax.

The `Cause` concept is used in order to define the causes of the ontology. It has been defined as the subclass of the intersection of three universal role restrictions (see Section 3.2 for more details about the roles of the ontology):

$$\begin{aligned}\mathtt{Cause} \sqsubseteq\ & \forall \mathtt{causeToCause.Cause} \sqcap \\ & \forall \mathtt{causeToSymptom.Symptom} \sqcap \\ & \forall \mathtt{causeToConsequence.Consequence}\end{aligned}$$

In that way, for a `Cause` instance, all of its values in the `causeToCause` role belong to the `Cause` concept, all of its values in the `causeToSymptom` role belong to the `Symptom` concept and all of its values in the `causeToConsequence` role belong to the `Consequence` concept.

The `Symptom` concept is used in order to define the symptoms of the ontology. It has been defined as the subclass of the intersection of four universal role restrictions:

$$\begin{aligned}Symptom \sqsubseteq=\ & \forall \mathtt{symptomToCause.Cause} \sqcap \\ & \forall \mathtt{symptomToSymptom.Symptom} \sqcap \\ & \forall \mathtt{symptomToConsequence.Consequence} \sqcap \\ & \geqslant 1 \mathtt{symptomToConsequence}.\top\end{aligned}$$

The definition is similar to the `Cause` concept, apart from an additional restriction on the `symptomToConsequence` role that defines the existence of at least one value in the role.

The `Consequence` concept is used in order to define the consequences of the ontology and it has been defined as the subclass of a single universal restriction:

$$\mathtt{Consequence} \sqsubseteq= \forall \mathtt{consequenceToConsequence.Consequence}$$

The `Antipattern` concept is the root concept of the antipattern hierarchy and is defined in terms of at least one cause, symptom and consequence instance values in the corresponding roles:

$$
\begin{aligned}
\mathtt{Consequence} \sqsubseteq= & \ \forall \mathtt{hasCause.Cause} \sqcap \\
& \forall \mathtt{hasSymptom.Symptom} \sqcap \\
& \forall \mathtt{hasConsequence.Consequence} \sqcap \\
& \geqslant 1 \ \mathtt{hasCause.}\top \sqcap \\
& \geqslant 1 \ \mathtt{hasSymptom.}\top \sqcap \\
& \geqslant 1 \ \mathtt{hasConsequence.}\top
\end{aligned}
$$

The other three antipattern-related concepts have been defined directly as subclasses of the `Antipattern` concept, that is,

$$
\begin{aligned}
& \mathtt{SoftwareDevelopment} \sqsubseteq \mathtt{Antipattern} \\
& \mathtt{SoftwareArchitecture} \sqsubseteq \mathtt{Antipattern} \\
& \mathtt{SoftwareProjectManagement} \sqsubseteq \mathtt{Antipattern}
\end{aligned}
$$

It is important at this point to mention that the concept and instance classification, as well as the consistency checking, are performed by the Pellet DL reasoner. Therefore, the open-world semantics are followed, that is, unstated information does not necessarily mean negated information. For that reason, the role restrictions we have described for the concept definitions do not act as constraints, in terms of database constraints. For example, it is valid to define an instance of the concept `Symptom` without specifying a `symptomToConsequence` value or an antipattern instance without a `hasSymptom` value, since the reasoner assumes that such a value exists but has not been stated yet (open world). However, all the values in the `symptomToConsequence` role of a symptom instance will be classified in the `Consequence` concept (necessary condition). In order to determine ontology inconsistencies in terms of value constraints (e.g. integrity constraints), we apply a set of CLIPS production rules that we describe in Section 4.2.

3.2. Role definitions

The ontology roles allow the definition of basic knowledge related to antipattern causes, symptoms and consequences, as well as to their correlations. Table 3 depicts a summary of the defined roles. In the rest of this section we describe in detail the definition of each role using the OWL RDF/XML syntax.

3.2.1. Documentation roles

The ontology defines two datatype roles for providing human-readable textual descriptions for ontology instances. The `title` role can be used in order to define a short title for an instance and the `description` role can be used in order to provide a detailed documentation. Intuitively, the `title` role can be used as the human-readable description of an ontology instance `rdf:ID`. Both roles are necessary for the successful interaction of the project manager with the interface of SPARSE. For that reason, SPARSE informs the user about the absence of values for any of the two roles for an instance (see Section 4.2).

```
<owl:DatatypeProperty rdf:ID="title">
  <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="description">
  <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
```

3.2.2. Cause, symptom and consequence object roles

The antipattern ontology allows the definition of correlations among causes, symptoms and consequences. In that way, the ontology reasoning procedures, that is, the Pellet DL reasoner and the SWRL rules, are able to infer correlations that are not explicitly stated in the ontology, exploiting the antipattern knowledge that has been created by different managers. In this section, for simplicity, we describe only the roles that correlate a cause with a cause, a symptom and a consequence.

The `causeToCause` object role allows the correlation of a cause with another cause. In that way, there is no need to state explicitly all the causes of a specific antipattern. The ontology reasoning procedure through the SWRL rules that we describe in Section 3.4, is able to infer all the relevant (implicit) causes for a specific antipattern following the `causeToCause` relations.

```
<owl:ObjectProperty rdf:about="#causeToCause">
  <rdfs:domain rdf:resource="#Cause"/>
  <rdfs:range rdf:resource="#Cause"/>
</owl:ObjectProperty>
```

The `causeToSymptom` object role allows the correlation of a cause with a symptom. The ontology reasoning procedure on this role allows the inference of additional symptoms for a specific antipattern, following the `causeToSymptom` relationships of asserted or inferred causes.

```
<owl:ObjectProperty rdf:about="#causeToSymptom">
  <rdfs:domain rdf:resource="#Cause"/>
  <rdfs:range rdf:resource="#Symptom"/>
</owl:ObjectProperty>
```

The `causeToConsequence` object role allows the correlation of a cause with a consequence. Similar to the previous role, the ontology reasoning procedure on this role allows the inference of additional consequences for a specific antipattern, based on asserted or inferred causes.

```
<owl:ObjectProperty
rdf:about="#causeToConsequence">
  <rdfs:domain rdf:resource="#Cause"/>
  <rdfs:range rdf:resource="#Consequence"/>
</owl:ObjectProperty>
```

Similar object roles to the above are also used for describing symptoms. In this paper, we assume that for consequences, the antipattern ontology defines only the `consequenceToConsequence` role, since SPARSE uses a symptom based approach and also the relationships between consequences and symptoms are better described through the symptoms of an antipattern.

Table 3
The roles of the antipattern ontology.

Role	Description
`title`	The title of a resource
`description`	The description of a resource
`causeToCause`	Correlates two causes
`causeToConsequence`	Correlates a cause with a consequence
`causeToSymptom`	Correlates a cause with a symptom
`symptomToSymptom`	Correlates two symptoms
`symptomToCause`	Correlates a symptom with a cause
`symptomToConsequence`	Correlates a symptom with a consequence
`consequenceToConsequence`	Correlates two consequences
`hasCause`	Defines a cause for an antipattern
`hasPrimaryCause`	Defines a primary cause
`hasSecondaryCause`	Defines a secondary cause
`hasImplicitCause`	Defines an implicit cause
`hasSymptom`	Defines a symptom for an antipattern
`hasPrimarySymptom`	Defines a primary symptom
`hasSecondarySymptom`	Defines a secondary symptom
`hasImplicitSymptom`	Defines an implicit symptom
`hasConsequence`	Defines a consequence for an antipattern
`hasPrimaryConsequence`	Defines a primary consequence
`hasSecondaryConsequence`	Defines a secondary consequence
`hasImplicitConsequence`	Defines an implicit consequence

3.2.3. Antipattern roles

The `hasCause`, `hasSymptom` and `hasConsequence` roles are used to define the causes, symptoms and consequences of an antipattern, respectively. Note that due to the ontology reasoning procedure, an antipattern might result with more causes, symptoms or consequences than it has been initially defined with, as we have explained in the previous section.

```
<owl:ObjectProperty rdf:about="#hasCause">
  <rdfs:domain rdf:resource="#Antipattern"/>
  <rdfs:range rdf:resource="#Cause"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasSymptom">
  <rdfs:domain rdf:resource="#Antipattern"/>
  <rdfs:range rdf:resource="#Symptom"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasConsequence">
  <rdfs:domain rdf:resource="#Antipattern"/>
  <rdfs:range rdf:resource="#Consequence"/>
</owl:ObjectProperty>
```

For each one of the above roles, the ontology defines three subproperties of the form `hasPrimaryX`, `hasSecondaryX` and `hasImplicitX`, with `X` being `Cause`, `Symptom` or `Consequence`. The values of the properties of the first two types are explicitly stated in the ontology, whereas the values of the properties of the third type derive after ontology reasoning. For example, the subproperties of the `hasSymptom` role are defined as:

```
<owl:ObjectProperty rdf:ID="hasPrimarySymptom">
  <rdfs:subPropertyOf rdf:resource="#hasSymptom"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasSecondarySymptom">
  <rdfs:subPropertyOf rdf:resource="#hasSymptom"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasImplicitSymptom">
  <rdfs:subPropertyOf rdf:resource="#hasSymptom"/>
</owl:ObjectProperty>
```

The purpose of the subproperty relationships is to allow an additional level of granularity during the definition of the antipattern's causes, symptoms and consequences that is used by the underlying antipattern retrieval mechanism to sort the results (see Section 4.4). For example, the antipatterns that match a specific symptom in their `hasPrimarySymptom` role are considered more relevant than the antipatterns that match the same symptom in their `hasSecondarySymptom` role. The implicit subproperties are used by the SWRL rules in order to insert the values they infer.

3.3. OWL ontology data collection

For the purposes of this paper, the antipattern ontology has been populated with data on 31 antipatterns that exist on the Web. Table 4 contains the sources of antipatterns used in the OWL ontology. The most popular antipattern repositories that exist on the Web at the moment are the Wikipedia Antipatterns Web page (Wikipedia Antipatterns, 2010), the Pattern Community Antipattern Catalogue (Pattern Community Antipattern Catalogue, 2010) and software project management blogs (Software Project Management Antipattern Blog, 2010; Software Project Antipattern-Pardon, 2010).

Using SPARSE, an antipattern can be categorized in any or all three different types of antipatterns: development, architecture or management. In the dataset used in this paper there exist a total of 30 software project management antipatterns, 10 development antipatterns and five architectural antipatterns. This implies that some antipatterns belong to two or more categories.

Each antipattern is associated with at least one symptom and the total number of symptoms that exist in the ontology is 63. Also there exist 65 causes and 51 consequences.

3.4. SWRL rules

SPARSE comes with a set of seven SWRL rules that are used by the Pellet DL reasoner in order to infer new antipattern causes, symptoms and consequences based on their correlations. Fig. 3 presents the set of the SWRL rules. Note that all inferred values are inserted into the corresponding implicit role. We follow such an approach since the explanation mechanism of SPARSE (see Section 4.4.3) needs to know whether a value has been derived after ontology reasoning or it has been explicitly stated in the ontology, in order to generate the appropriate explanation message.

```
1) hasSymptom(?anti, ?s) ∧ symptomToConsequence(?s, ?con)
     → hasImplicitConsequence(?anti, ?con)
2) hasSymptom(?anti, ?s) ∧ symptomToCause(?s, ?c)
     → hasImplicitCause(?anti, ?c)
3) hasCause(?anti, ?c1) ∧ causeToCause(?c1, ?c2)
     → hasImplicitCause(?anti, ?c2)
4) hasConsequence(?anti, ?c1) ∧
                        consequenceToConsequence(?c1, ?c2)
     → hasImplicitConsequence(?anti, ?c2)
5) hasCause(?anti, ?c) ∧ causeToSymptom(?c, ?s)
     → hasImplicitSymptom(?anti, ?s)
6) hasCause(?anti, ?c) ∧ causeToConsequence(?c, ?con)
     → hasImplicitConsequence(?anti, ?con)
7) hasSymptom(?anti, ?s1) ∧ symptomToSymptom(?s1, ?s2)
     → hasImplicitSymptom(?anti, ?s2)
```

Fig. 3. The SWRL rules of SPARSE.

Table 4
Sources of antipatterns used in the antipattern OWL ontology.

Source	Antipattern	Quantity
Wikipedia antipatterns	Death march, Groupthink, Smoke and mirrors, Software bloat	4
Pattern community, Wiki antipattern, catalogue	Analysis Paralysis, Architectre by Implication, An Athena, Appointed Team, Architects Dont Code, Blowhard Jamboree, Corn Cob, Carbon Copy His Manager, Death By Planning, Decision By Arithmetic, Dry Waterhole, Egalitarian Compensation, Email Is Dangerous, Emperors New Clothes, Fear Of Success, Glass Wall, Leading Request, Shoot The Messenger, Smoke and Mirrors, The BLOB, The Customers are Idiots, Thrown Over The Wall, Train The Trainer, Untested But Finished, Yet Another Meeting Will Solve It	25
Software Project Management Blogs	Pardon my dust, Project Managers who write specs	2

To exemplify, consider an ontology that defines an antipattern antip to have the symptom sym1 (hasSymptom(antip, sym1)) and that symptom sym1 is correlated with symptom sym2 (symptomToSymptom(sym1, sym2)). By incorporating the SWRL rules in the ontology reasoning procedure, we are able to infer also that antipattern antip has the (implicit) symptom sym2 (rule 7 in Fig. 3). Such type of reasoning is very useful since it enables the collaborative definition of antipattern relationships, drawing conclusions using relationships that have been modeled by different project managers, without requiring from them to be aware of the complete domain knowledge.

It is worth mentioning the OWL two language (Grau et al., 2008), an extension to OWL that allows the SWRL rules of SPARSE to be expressed directly as *property chains*. For example, the rule 7 of Fig. 3 is actually a property chain of the form

hasSymptom o symptomToSymptom → hasImplicitSymptom

that can be expressed in OWL two as

```
<rdf:Description>
  <rdfs:subPropertyOf rdf:resource="#hasImplicit-
  Symptom"/>
  <owl:propertyChain rdf:parseType="Collection">
    <rdf:Description rdf:about="#hasSymptom"/>
    <rdf:Description rdf:about="#symptomToSymp-
    tom"/>
  </owl:propertyChain>
</rdf:Description>
```

We plan to move to the OWL two language, as soon as the underlying tools of SPARSE, i.e. the OWLAPI and the Pellet DL reasoner, provide stable and full support for OWL 2.

4. The rule program of SPARSE

SPARSE is based on the reasoning capabilities of the CLIPS production rule engine in order to apply the antipattern matching algorithm. In this section we make a short introduction to the CLIPS production rule engine, we refer briefly to the basic elements of the mapping procedure of the Pellet inferred knowledge base on the COOL language of CLIPS and we analyze the antipattern matching algorithms that are implemented over the generated COOL model.

4.1. The COOL model

CLIPS is a RETE-based production rule engine written in C that was developed in 1985 by NASA's Johnson Space Center and it has undergone continual refinement and improvement ever since. Today it is widely used throughout the government, industry and academia. Some of its main features are stability, speed, extensibility and low cost (public domain software).

One of the most interesting capabilities of CLIPS is that integrates the production rule paradigm with the model, which can be defined using the COOL (CLIPS object-oriented language) language of CLIPS. In that way, classes, attributes and objects can be matched in the production rule conditions (LHS), as well as to be altered on rule actions (RHS), presenting a fully dynamic behavior.

The semantics of CLIPS are the usual production rule semantics: rules whose condition is successfully matched against the current data are triggered and placed in the conflict set. The conflict resolution mechanism selects a single rule for firing its action, which may alter the data. Rule condition matching is performed incrementally, through the RETE algorithm.

```
(defclass <name>
      (is-a <superclass-name>+)
      (<multislot>* <constraint>*))

(make-instance [<name>] of <class>
      [(<multislot> <values>)]*)
```

(a) Defining classes, multislots and objects

```
(object <attribute-constraint>*))

<attribute-constraint> ::=
  (is-a <constraint>) |
  (name <constraint>) |
  (<multislot-name> <constraint>*)
```

(b) Matching objects in rule conditions

Fig. 4. Basic syntax of the COOL language of CLIPS.

Concerning the OO model, the semantics of CLIPS are the usual of an OO programming environment. CLIPS supports *abstraction*, *inheritance*, *encapsulation*, *polymorphism* and *dynamic binding*. Some of the main OO features of CLIPS include:

- *Single and multiple inheritance.* It is possible to define multiple direct superclasses for a CLIPS class. In that way, each object of a subclass belongs also to all superclasses.
- *Attribute inheritance.* Class attributes are inherited to subclasses as well.
- *Message-based functionality.* Each object is able to receive and send messages to other objects. Every action related to objects is performed via messages, such as the insertion of values into object attributes (put message) or for reading the attribute values of an object (get message).
- *Single and multifield attributes.* Each attribute can be defined as a single slot, taking only a single value, or as a multifield (multislot) attribute, able to take more than one values in a form of a list.

Fig. 4a depicts the basic COOL syntax for defining classes and objects. A class is defined by specifying the name, one or more superclasses, and zero, one or more multislot (attribute) definitions with type constraints. The type constraint is used to restrict the type of datatype attributes, and the allowed-classes constraint denotes the class type of the objects that a multislot can take, using the INSTANCE-NAME type constraint. The built-in USER class of COOL must be the root class of the class hierarchy. An object is defined by specifying the name inside brackets ([and]), which is actually the object ID, the class type and any multislot value. Fig. 4b depicts the object pattern syntax that is used in CLIPS production rules in order to match objects in the body of rules. An object is matched if it satisfies the is-a class constraint, the name constraint and the constraints about multislot values. A variable, which is denoted as ?x (a multislot variable is denoted as ?x), can be used at any place, except for the multislot name.[2]

4.2. Validation rules

The rule program of SPARSE contains a set of validation production rules that check the OO KB for simple inconsistencies, such as the absence of object titles or descriptions. Furthermore, there are rules that treat some OWL restrictions as database constraints, as we have described in Section 3.1. For example, the following

[2] http://clipsrules.sourceforge.net/.

production rule informs users about symptom objects that do not define any symptomToConsequence value.

```
(defrule validation-rule-check-symptoms
   (object
      (is-a Symptom)
      (title ?title)
      (symptom ToConsequence $?cons & :
         (eq (length$ $?cons) 0)))
   =>
   (printout t "The symptom" ?title "... " crlf))
```

4.3. The object-oriented mapping procedure

The goal of the mapping procedure is to preserve the extensional knowledge of the ontology, that is, instance class membership relationships and instance role values, in terms of object class type declarations and object attribute values, respectively. To achieve this goal, the mapping procedure consists of two phases: (a) the generation of an OO schema of classes with attributes that is not necessarily compliant with the TBox ontology semantics, but it is specially defined in order to preserve the ABox semantics, and (b) the definition of the objects and their attribute values with respect to the previously defined OO schema. The inadequacy of preserving the terminological semantics of OWL ontologies in the OO model does not affect the importance of our methodology, since in many rule-based application, such as SPARSE, we are usually interested in the extensional knowledge of a domain, which we are able to fully represent in our OO model.

Note that the mapping approach is not intended to be used as a framework for manipulating ontologies with rules. Instead, it is suitable in cases where there is a need for building rule-based applications using the extensional ontological information as the back-end base model, following the declarative rule paradigm rather than the procedural programming. The domain where SPARSE targets at is fully compliant with this requirement. Based on the antipattern ontology, the rule-based program of SPARSE operates over the extensional knowledge of the antipattern ontology (objects), retrieving relevant antipatterns according to an initial set of symptoms.

In this section we present the basic characteristics of the mapping procedure of the Pellet inferred KB on the COOL model of CLIPS that is necessary for the legibility of the antipattern retrieval algorithms that we present in Section 4.4. The interested reader may refer to Meditskos and Bassiliades (2008a) and Meditskos and Bassiliades (2008b) for more technical details.

4.3.1. Mapping ontology concepts and roles

The mapping procedure of the antipattern's ontology TBox is based on the subsumption hierarchy that is computed by Pellet and results in an OO schema of classes with attributes. There are two basic characteristics:

- For each named ontology concept, there is a corresponding OO class. Therefore, only the named ontology concepts are mapped on the OO model. Any restriction and anonymous concept are used only by the Pellet DL reasoner in order to perform concept classification and instance realization.
- For each ontology role, there is at least one attribute (multislot) definition in an OO class. The class where the attribute is defined is determined based on the ontology domain restrictions and the attribute type is determined based on the range restrictions.

In that way, every named ontology concept and role is accessible in the OO model and they can be used in COOL production rules in order to match appropriate objects. To exemplify, the Symptom concept is mapped on the following defclass construct.

```
(defclass Symptom
   (is-a Thing)
   (multislot symptomToConsequence
   (type INSTANCE-NAME)
   (allowed-classes Consequence))
(multislot symptomToSymptom
   (type INSTANCE-NAME)
   (allowed-classes Symptom))
(multislot symptomToCause
   (type INSTANCE-NAME)
   (allowed-classes Cause)))
```

4.3.2. Mapping ontology instances and role values

With the mapping algorithms of concepts and roles the goal is to generate the necessary infrastructure on which the objects will be defined, allowing instance-related information to be queried during the development of rule-based applications. There are two basic characteristics:

- Each instance class membership relationship in the ontology is mapped on an appropriate object class type declaration in the OO model. This mapping is defined in such a way, so that each ontology instance has a corresponding OO object that can be retrieved by querying the corresponding OO classes to the ontology concepts where the instance belongs in the ontology.
- An ontology instance and its corresponding object have the same values in the corresponding roles/attributes.

The above relationships are defined in terms of the Unique Name Assumption (UNA), since the instance equality of OWL cannot be represented directly in the OO model. To exemplify, we present a simplified definition, for legibility purposes, of a symptom instance.

```
(make-instance [Focus_on_cost] of Symptom
   (symptomToSymptom [Direct_pressure])
   (symptomToConsequence [Failure_to_deliver])
   (description "The focus becomes cost than
                the actual delivery")
   (title "Focus on cost"))
```

Note that the description and title attributes are defined in the Thing class, since they do not have any domain restriction (see Section 3.2.1). Therefore they are accessible by all the objects of the OO KB since every class is a direct or indirect subclass of the Thing class, such as the Symptom class.

4.4. Searching for antipatterns

The main functionality of SPARSE is to propose antipatterns based on a set of symptoms that users select from the antipattern ontology. The antipatterns that are returned by SPARSE can be classified into two categories:

- *Symptom-based matched antipatterns.* These are the antipatterns that contain one or more user-selected symptoms.
- *Relevant antipatterns.* SPARSE proposes also a set of antipatterns that might be relevant to the symptom-based returned antipatterns, examining their causes and consequences.

In the rest of this section we describe in detail the two antipattern matching procedures and we elaborate on the explanation

capabilities of SPARSE that justifies the presence of a specific antipattern in the result set.

4.4.1. Symptom-based matched antipatterns

The matching of antipatterns based on symptoms is performed by a set of production rules that traverse the antipattern objects of the COOL KB and select the ones that satisfy one or more user-selected symptoms. The results are ordered according to a relevance score that is assigned to each matched antipattern.

Assuming that $Obj(A)$ denotes the set of all the objects of the class A and that $o.p$ denotes the set of values of the object o for the attribute p, the symptom-based antipattern matching procedure is depicted in Algorithm 1.

Algorithm 1. Symptom-based antipattern matching

```
   Input: The set Q of query symptoms
   Output: The ordered set of matched antipatterns
1  Set P ← ∅,S ← ∅,I ← ∅,U ← ∅
2  Map < Antipattern,Score > R ← ∅
3  forall a ∈ Obj(Antipattern) do
4      Set V ← a.hasSymptom
5      forall s ∈ Q ∧ s ∈ V do
6          If IsPrimarySymptom(s,a) then P ← P ∪ {s}
7          else if IsSecondarySymptom(s,a) then
           S ← S ∪ {s}
8          else if IsImplicitSymptom(s,a) then
           I ← I ∪ {s}
9          else if IsUndefinedSymptom(s,a) then
           U ← U ∪ {s}
10     Set P' ← a.hasPrimarySymptom
11     Set S' ← a.hasSecondarySymptom
12     Set I' ← a.hasImplicitSymptom
13     Set U' ← GetUndefinedSymptoms(a)
14     Score score ← (w_p·|P|+w_s·|S|+w_i·|I|+w_u·|U|) / (w_p·|P'|+w_s·|S'|+w_i·|I'|+w_u·|U'|)
       // w_p,w_s,w_i,w_u ∈ [0..1]
15     if score ≠ 0 then R ← R ∪ { < a,score > }
16 return sort_desc,Score(R)
```

More specifically, for each object antipattern a (line 3) the algorithm collects its primary, secondary, implicit and undefined symptoms that exist also in the set of the user-selected symptoms Q (lines 5–9). An undefined symptom s is a symptom that has not been categorized as primary, secondary or implicit in an antipattern a, that is,

$$\begin{aligned} \mathtt{isUndefinedSymptom}(s,a) \iff\ = s \notin a.\mathtt{hasPrimarySymptom} \wedge \\ s \notin a.\mathtt{hasSecondarySymptom} \wedge \\ s \notin a.\mathtt{hasImplicitSymptom} \end{aligned}$$

For each retrieved antipattern, the algorithm assigns a score based on the number of symptoms that it matches against its total number of symptoms. The way each symptom category affects the overall score can be adjusted using four weights. The default weight values of SPARSE is $w_p = w_u = 1$ and $w_s = w_i = 0.4$. Finally, the results are sorted in descending order based on their score and are presented to the user.

4.4.2. Proposing relevant antipatterns

Apart from the symptom-based matched antipatterns, SPARSE proposes also a set of potentially useful antipatterns that are relevant to the former antipattern in terms of their causes and consequences. Algorithm 2 summarizes the approach. More specifically, the algorithm finds antipatterns that have common causes and/or consequences with one or more symptom-based matched antipatterns. In this case, for simplicity, we do not take into consideration any category difference of causes and consequences (primary, secondary, implicit and undefined), but we treat them as equivalent. The overall score of an antipattern is computed as the mean value of the partial scores of the antipattern with the symptom-based retrieved antipatterns that matches (lines 11–14).

```
   Input: The set SB of symptom-based matched antipatterns
   Output: The ordered set of matched antipatterns
1  Map < Antipattern,Score > R ← ∅
2  forall a' ∈ Obj(Antipattern) do
3      Score score ← 0
4      matches ← 0
5      forall a ∈ SB do
6          CA ← a.hasCause
7          CA' ← a'.hasCause
8          CO ← a.hasConsequence
9          CO' ← a'.hasConsequence
10         partial_score ← |CA∩CA'|/|CA∪CA'| + |CO∩CO'|/|CO∪CO'|
11         if partial_score ≠ 0 then
12             score ← score + partial_score
13             matches ← matches + 1
14     if score ≠ 0 then R ← R ∪ {< a', score/matches >}
15 return sort_desc,Score(R)
```

4.4.3. Explanations

SPARSE implements an explanation mechanism that presents to users in textual form the relationships that resulted in the inclusion of a specific antipattern in the result set. In the case of a symptom-based matched antipattern, SPARSE presents the user-selected symptoms that the antipattern satisfies, along with their category. In the case of a relevant antipattern to one or more symptom-based matched antipatterns, SPARSE presents all the relationships that the antipattern shares with the symptom-based matched antipatterns, along with their category. In the next section, we present the capabilities of SPARSE, along with the basic steps that should be followed in order to define and detect antipatterns.

5. Using sparse

In order to populate SPARSE with new antipatterns, software project managers can use any OWL ontology editor, e.g. Protégé,[3] in order to define antipattern ontology concepts and roles (as described in Section 3). Besides populating the antipattern ontology, the main functionality of SPARSE is to propose related antipatterns according to the symptoms that exist in a software project. In this section, the process of creating new instances of the antipattern ontology is described. This section also describes how SPARSE can be used in order to propose directly related but also semantically retrieved antipatterns, according to a set of symptoms that exist in a software project.

5.1. Populating the OWL antipattern ontology using protégé

The first step that software project managers need to do before populating the ontology with new antipatterns is to understand the existing antipatterns and antipattern attributes that are

[3] http://protege.stanford.edu/.

already documented in the ontology. As already mentioned, for the purposes of this paper, the OWL antipattern ontology has been populated with data from 31 antipatterns that exist on the Web. The ontology also contains 63 symptoms, 65 causes and 51 consequences. Therefore, it is important to use an ontology editor in order to explore the existing antipatterns and attributes by reading the associated description that is provided in plain text in the ontology. This will enable users of SPARSE to reuse existing antipattern attributes for the creation of new antipatterns. Alternatively, software project managers can use SPARSE in order to understand the existing antipatterns and the reasons why they are related. Understanding the data that already exists in the ontology, is an important step towards reducing redundancy and duplication of antipatterns or antipattern attributes.

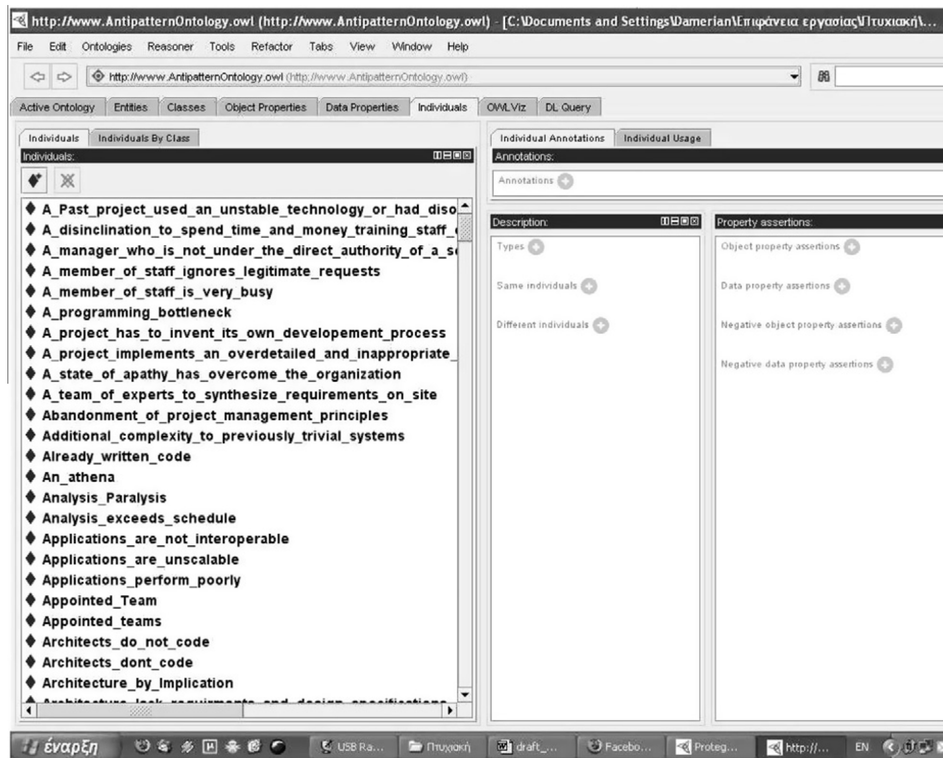


Fig. 5. The individuals tab in Protege 4.0.

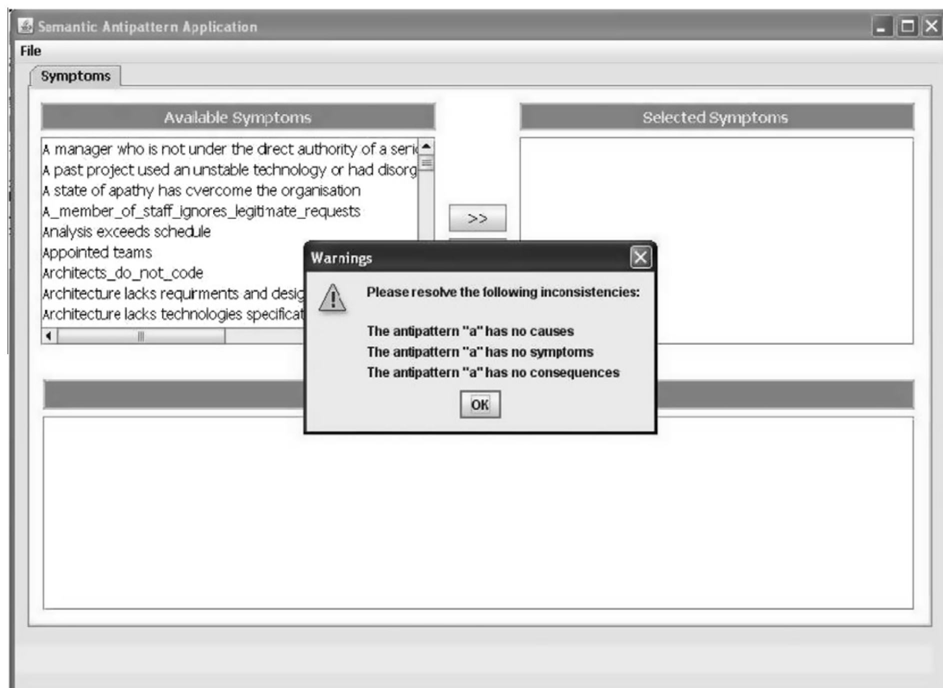


Fig. 6. Example SPARSE window containing error messages because the required concepts for a new antipattern have not been defined properly.

The following properties can be defined for a new antipattern or antipattern attributes: title, description, causeToCause, causeToConsequence, causeToSymptom, symptomToCause, symptomToConsequence, symptomToSymptom, consequenceToConsequence, hasCause, hasSymptom, hasConsequense. In order to create a new antipattern instance, an ontology editor should be used, such as Protégé. In this case, the above ontology roles can be found in the individual tab of Protege (Fig. 5) through which new antipatterns and attributes can be created.

The proposed order in which roles and concepts should be entered is the following: first Consequences and possible consequenceToConsequence values should be entered. Symptoms should then be entered together with symptomToConsequence values. Causes can now be defined together with

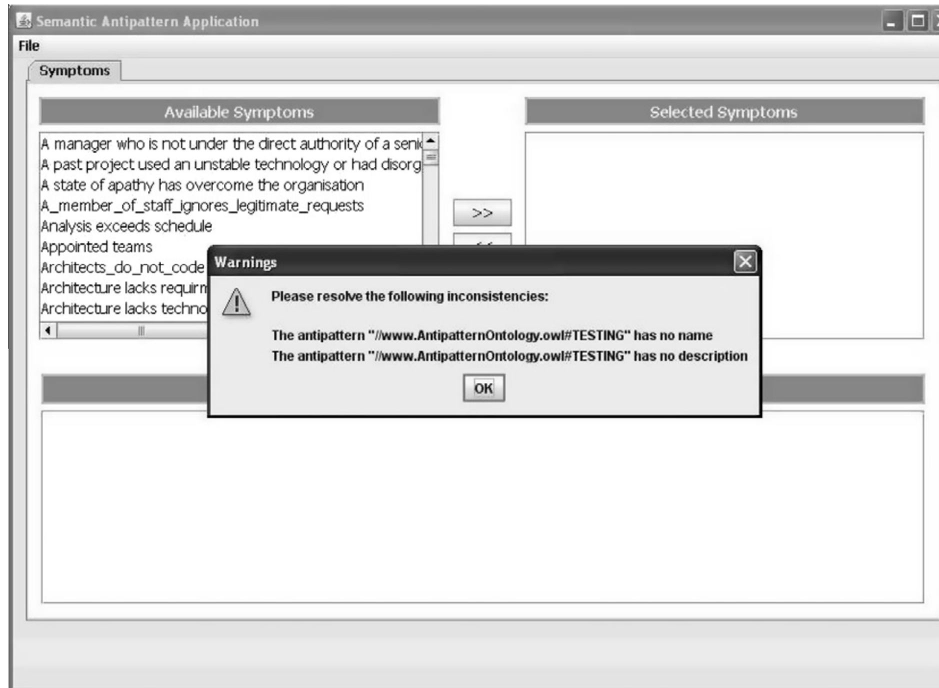


Fig. 7. Example SPARSE window containing error messages because the title and description fields have not been completed.

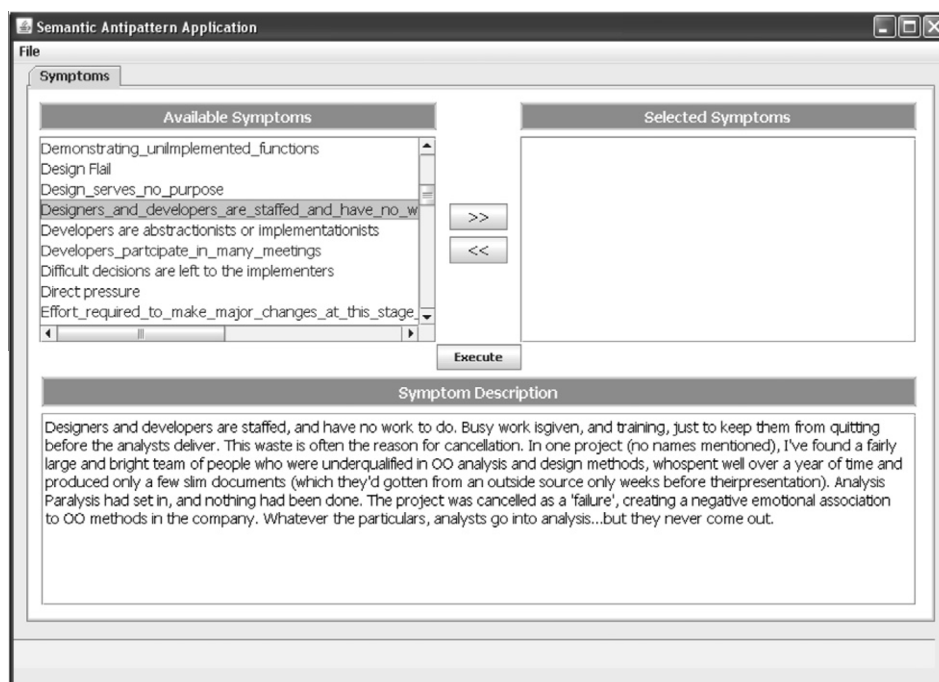


Fig. 8. Selecting a set of symptoms that exist in a software project using SPARSE.

causeToConsequence and causeToSymptom values. Finally, users can define the antipattern and relate it with existing symptoms, causes and consequences through the hasCause, hasSymptom and hasConsequense roles. Users should be able to familiarize themselves fairly quickly with the concepts and roles that already exist in the OWL ontology. For example, every antipattern should have at least one cause, symptom and consequence. Also every symptom should have at least one or more symptom to consequence value. If the predefined concepts and roles are not taken into account, the inconsistencies will appear on a window when launching SPARSE (Fig. 6).

In this case, Protégé should be used again in order to correct the identified inconsistencies and enable SPARSE to be launched properly. Other important roles are the title and description of an antipattern or antipattern attribute. This is mandatory to be completed and if, for example, the title is left uncompleted, SPARSE will display an error message window (Fig. 7) requesting for this inconsistency to be resolved.

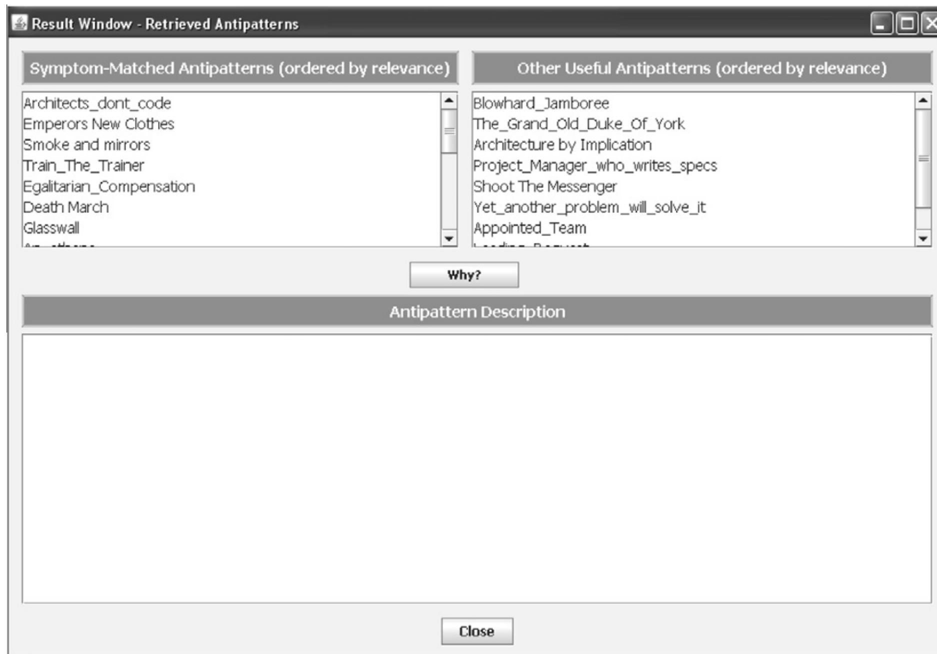


Fig. 9. The result window of SPARSE displaying related antipatterns that might exist in a software project based on the set of selected symptoms.

Fig. 10. The explanation window of SPARSE describing why an antipattern might be a relevant result.

5.2. Using SPARSE to detect related antipatterns

SPARSE is available as a desktop application. Users can download SPARSE together with the OWL antipattern ontology from the Web.[4] SPARSE can be launched by executing the .JAR file. The associated OWL ontology should then be imported from the File menu. SPARSE will then display the available symptoms that can be selected for program execution (Fig. 8).

After selecting a set of symptoms users can execute SPARSE. The result window displays two sets of antipattern (Fig. 9). On the left hand side there are the related symptom-based antipatterns, which are antipatterns linked through their symptoms. Antipatterns that appear on the right hand side are related through causes or consequences. By clicking on an antipattern SPARSE displays the description and the refactored solution of the antipattern.

The process of selecting antipatterns has been discussed in Section 4. SPARSE can provide explanations to its users on the reasons that a specific antipattern has been proposed. Fig. 10 illustrates an example of the explanations that SPARSE provides on how a selected symptom linked through causes and consequences of other antipatterns.

6. Summary and conclusion

Semantic Web technologies and knowledge-based systems have provided antipatterns with new knowledge acquisition, representation and sharing options. The proposed intelligent system can bring the software project managers' attention to focus on antipatterns that are specifically suitable to a specific software project. Hence, a software project manager who wishes to detect antipatterns will not require expertise to determine which antipattern is most likely to appear at a given moment.

Antipatterns offer a common vocabulary of terms that helps software project managers to communicate better. However, with the growing number of antipatterns appearing in the literature and the Web with no relationships between each other, antipatterns sometimes contradict, duplicate, or are inconsistent with each other. SPARSE offers a framework to capture software project management knowledge and foster a community of software project management contributors. We believe that this is an important step towards resolving many of the problems that currently plague the antipattern community.

As future work related to the proposed system, SPARSE can be used to create a community of antipattern authors who contribute knowledge by creating new instances of antipatterns. Eventually, SPARSE will motivate the use of antipatterns in software project management. Evaluation of the proposed software tool is necessary in order to determine its suitability as a communication medium for the antipattern community. Finally, the feasibility of using SPARSE as a tool for teaching software project management should also be investigated.

References

Alexander, C. (1979). *The timeless way of building*. Oxford University Press.

Antoniou, G., Damasio, C. V., Grosof, B., Horrocks, I., Kifer, M., Maluszynski, J., et al. (2005). *Combining rules and ontologies. a survey, deliverable REWERSE-DEL-2005-I3-D3*. Institutionen for Datavetenskap, Linkopings Universitetet.

Baader, F. (2003). *The description logic handbook: Theory, implementation and applications*. Cambridge University Press.

Baader, F., & Sattler, U. (2001). An overview of tableau algorithms for description logics. *Studia Logica, 69*, 5–40.

Beckett, D. (2004). *RDF/XML syntax specification (revised), W3C recommendation, W3C*. URL: <http://www.w3.org/TR/rdf-syntax-grammar/>.

Brown, W., Malveau, R., McCormick, H., & Mowbray, T. (1998). *AntiPatterns: Refactoring software, architectures, and projects in crisis*. Wiley Computer Publishing.

Brown, W., McCormick, H., & Thomas, S. (2000). *AntiPatterns in project management*. Wiley Computer Publishing.

de Souza, M. A. F., & Ferreira, M. A. G. V. (2002). Designing reusable rule-based architectures with design patterns. *Expert Systems with Applications, 23*, 395–403.

Dietrich, J. & Elgar, C. (2005). Towards a web of patterns. In *SWESE 2005. Web semantics: Science, services and agents on the world wide web archive* (Vol. 5, pp. 108–116).

Gamma, R., Helm, E., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

Grant, J., & Beckett, D. (2004). *RDF test cases, W3C recommendation*. URL: <http://www.w3.org/TR/rdf-testcases/>.

Grau, B., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P., & Sattler, U. (2008). OWL 2: The next step for OWL, web semantics: Science. *Services and Agents on the World Wide Web, 6*(4), 309–322.

Haarslev, V. & Möller, R. (2003). Racer: A core inference engine for the semantic web. In *Proceedings of the 2nd international workshop on evaluation of ontology-based tools (EON2003)* (pp. 27–36). Sanibel Island, Florida, USA.

Henninger, S. (2006). Disseminating usability design knowledge through ontology-based pattern languages. In *3rd International semantic web user interaction workshop (SWUI 07)*.

Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B., & Dean, M. (2004). *SWRL: A semantic web rule language combining OWL and RuleML*. Tech. rep., W3C member submission. URL: <http://www.w3.org/Submission/SWRL/>.

Krai, J. & Zemlicka, M. (2007). The most important service-oriented antipatterns. In *ICSEA 2007 international conference on software engineering advances*. 25–31 August.

Kuranuki, Y. & Hiranabe, K. (2004). Antipractices: Antipatterns for xp practices. In *Agile development conference*.

Laplante, P., Hoffman, R., & Klein, G. (2007). Antipatterns in the creation of intelligent systems. *IEEE Intelligent Systems, 22*, 91–95.

Laplante, P., & Neil, C. (2006). *Antipatterns: Identification, refactoring and management*. Taylor and Francis.

Liao, S.-H. (2005). Expert system methodologies and applicationsóa decade review from 1995 to 2004. *Expert Systems with Applications, 28*, 93–103.

McCormick, H. (1999). Antipatterns, private correspondence. In *Presentation material – 3rd annual European conference on JavaTM and Object Orientation*. Denmark.

Meditskos, G. & Bassiliades, N. (2008a). HOOPO: A hybrid object-oriented integration of production rules owl ontologies. In *18th European conference on artificial intelligence (ECAI)* (pp. 729–730).

Meditskos, G. & Bassiliades, N. (2008b). *Mapping the extensional knowledge of dl reasoners on the object-oriented model*. Technical report. URL: <http://lpis.csd.auth.gr/publications/meditskos-techReport2008.pdf>.

Moynihan, G. P., Suki, A., & Fonseca, J. D. (2006). An expert system for the selection of software design patterns. *Expert Systems, 23*, 39–52.

Pattern community antipattern catalogue (2010) URL: <http://c2.com/cgi/wiki?AntiPatternsCatalog>.

Riley, G. (1991). Clips: An expert system building tool. In *Proceedings of the technology 2001conference*.

Settas, D. & Stamelos, I. (2007). Using ontologies to represent software project management antipatterns. In *Proceedings of the sofware engineering knowledge engineering conference (SEKE 2007)* (pp. 604–609).

Settas, D. & Stamelos, I. (2007). Towards a dynamic ontology based software project management antipattern intelligent system. In *Proceedings of the nineteenth IEEE international conference on tools with artificial intelligence (ICTAI)* (pp. 186–193).

Settas, D. & Stamelos, I. (2008). Resolving complexity and interdependence in software project management antipatterns using the dependency structure matrix. In R. Lee (Ed.), *Proceedings of the 6th ACIS IEEE intenational conference on software engineering research and applications (SERA 2008), Springer series: Studies in computational intelligence* (Vol. 150).

Settas, D., Bibi, S., Sfetsos, P., Stamelos, I., & Gerogiannis, V. (2006). Using bayesian belief networks to model software project management antipatterns. In *4th ACIS international conference on software engineering research, management and applications* (pp. 117–124). SERA.

Settas, D., Sowe, S., & Stamelos, I. (2009). Addressing software project management antipattern ontology similarity using semantic social networks. *The Knowledge Engineering Review, 24*(3), 287–308.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., & Katz, Y. (2007). Pellet: A practical owl-dl reasoner. *Web Semantics: Science, Services and Agents on the world Wide Web, 5*(2), 51–53.

Software project management antipattern blog (2010) Pardon my dust antipattern. URL<http://blogs.msdn.com/nickmalik/archive/2006/01/19/PMAntipattern-Pardon-My-Dust.aspx>.

Software project management antipattern blog (2010) project managers who write specs antipattern. URL:<http://blogs.msdn.com/nickmalik/archive/2006/01/03/508964.aspx>.

Tsarkov, D. & Horrocks, I. (2006). Fact++ description logic reasoner: System description. In *Third international joint conference on automated reasoning (IJCAR)* (Vol. 4130, pp. 292–297). LNAI.

Turban, E., & Aronson, J. E. (2001). *Decision support systems and intelligent systems* (6th ed.). Prentice International Hall.

Van Harmelen, F. (2003). *Towards the semantic web: Ontology-driven knowledge management*. Wiley.

Wikipedia antipatterns (2010). URL:<http://en.wikipedia.org/wiki/Anti-pattern>.

[4] http://sweng.csd.auth.gr/∼dset/SPARSE/.