ROoST: Reference Ontology on Software Testing

Article *in* Applied ontology · March 2017
DOI: 10.3233/AO-170177

3 authors, including:

Erica Ferreira de Souza
Federal Technological University of Paraná, Brazil
22 PUBLICATIONS **87** CITATIONS
SEE PROFILE

Ricardo de Almeida Falbo
Universidade Federal do Espírito Santo
172 PUBLICATIONS **1,661** CITATIONS
SEE PROFILE

Applied Ontology 0 (2017) 1–30
IOS Press

ROoST: Reference Ontology on Software Testing

Érica Ferreira de Souza [a,*], Ricardo de Almeida Falbo [b] and Nandamudi Lankalapali Vijaykumar [c]
[a] *Department of Computer, Federal Technological University of Paraná, UTFPR, Cornélio Procópio/PR, Brazil*
[b] *Department of Computer Science, Federal University of Espírito Santo, UFES, Vitória/ES, Brazil*
[c] *Laboratory of Computing and Applied Mathematics, National Institute for Space Research, INPE, São José dos Campos/SP, Brazil*

Abstract. Software testing is a complex and critical process for achieving product quality. Its importance has been increasing and well recognized, and there is a growing concern in improving the accomplishment of this process. In this context, Knowledge Management (KM) emerged as an important supporting approach to improve the software testing process. However, managing relevant testing knowledge requires effective means to represent and to associate semantics to a large volume of testing information. To address this concern, we have developed a Reference Ontology on Software Testing (ROoST). ROoST establishes a common conceptualization about the software testing domain, which can serve several KM-related purposes, such as defining a common vocabulary for knowledge workers with respect to the testing domain, structuring testing knowledge repositories, annotating testing knowledge items, and for making search for relevant information easier. In this paper, we present ROoST, and we discuss how it was developed using two ontology pattern languages. Moreover, we discuss how we evaluated ROoST following four complementary approaches: assessment by humans, data-driven evaluation, ontology testing, and application-based evaluation.

Keywords: Software Testing, Ontology, Reference Ontology, Ontology Evaluation, Knowledge Management

1. Introduction

Software Verification & Validation (V&V) activities intend to ensure, respectively, that a software product is being built in conformance with its specification, and that it satisfies its intended use and the user needs (IEEE, 2004). Software testing consists of dynamic V&V of the behavior of a program against the expected behavior (Bourque and Fairly, 2014). To be effective, testing activities should be integrated into a well defined controlled testing process. Nowadays, the importance of testing processes is widely recognized, and there is a growing concern in how to improve the accomplishment of this process (TMMi, 2012).

During the entire testing process, a significant amount of information is generated. Software testing is a knowledge intensive process, and thus it is important to provide computerized support for tasks of acquiring, processing, analyzing and disseminating testing knowledge for reuse (Andrade et al., 2013). Thus, Knowledge Management (KM) emerges as an important means to manage software testing knowledge since KM principles can help capturing and representing testing knowledge in an affordable and manageable way.

There are many benefits of managing software testing knowledge, such as (Souza et al., 2015a): (i) selection and application of better suited techniques; (ii) cost reduction; (iii) increase of test effectiveness; and (iv) competitive advantages. However, there are also problems, such as (Souza et al., 2015a): (i) employees are normally reluctant to share their knowledge; (ii) knowledge sharing may increase the employee workload; and (iii) existing communication systems are not appropriate.

In this context, one of the main challenges is how to represent knowledge. A KM system should support integrating information from disparate sources for a decision maker handling information that someone

*Corresponding author: Érica Ferreira de Souza, Department of Computer, Federal Technological University of Paraná, UTFPR. Av. Alberto Carazzai, 1640, 86300-000, Cornélio Procópio/PR, Brazil. E-mail: ericasouza@utfpr.edu.br

else conceptualized and represented. So, the KM system must minimize ambiguity and imprecision in interpreting shared information. This can be achieved by representing the shared information using ontologies (Kim, 2000). As pointed out by Staab et al. (2001), ontologies are particularly important for KM. They bind KM activities together, allowing a content-oriented view of KM. Ontologies define a shared vocabulary to be used in the KM systems to facilitate knowledge communication, integration, search, storage and representation (Benjamins et al., 1998).

Considering this context, to represent the software testing knowledge, we need a software testing ontology. More specifically, we need a reference domain ontology, i.e. a domain ontology that is constructed with the main goal of making the best possible description of the domain as realistic as possible. A reference domain ontology is a special kind of conceptual model representing a model of consensus within a community. It is a solution-independent specification with the aim of making a clear and precise description of domain entities for the purposes of communication, learning and problem-solving (Guizzardi, 2007). A reference ontology on the software testing domain can be used for several KM-related purposes, such as defining a common vocabulary for knowledge workers regarding the testing domain, structuring knowledge repositories, annotating knowledge items, and for making search for relevant information easier.

By means of a Systematic Literature Review (SLR), we looked for ontologies in the software testing domain (Souza et al., 2013). For analyzing these ontologies, we considered some of the characteristics pointed out by d'Aquin and Gangemi (2011) as characteristics that are presented in "beautiful ontologies". In our analysis, we considered the following characteristics: having a good domain coverage; implementing an international standard; being formally rigorous; implementing also non-taxonomic relations; following an evaluation method; and reusing foundational ontologies. As a result, we found 12 ontologies. However, after analyzing them, we concluded that they are insufficient for representing software testing knowledge in a KM system. We highlight following to justify why they are insufficient: most ontologies have limited coverage; the studies do not discuss how the ontologies were evaluated; none of the analyzed testing ontologies is truly a reference ontology; and none of them is grounded in a foundational ontology. Briefly, we concluded that the software testing community should invest more efforts to get a well-established reference software testing ontology. Therefore, we decided to build another one, which we called ROoST (Reference Ontology on Software Testing). The purpose of ROoST is to define a shared vocabulary regarding the testing domain to be used in KM initiatives.

In this paper, we present ROoST, which comprises sub-ontologies focusing on the software testing process and its activities, the artifacts that are used and produced by those activities, the testing environment, and testing techniques for test case design. Moreover, we discuss how ROoST was evaluated following four complementary approaches: assessment by humans, data-driven evaluation, ontology test, and application-based evaluation.

This paper is organized as follows. In Section 2, we present the main concepts related to software testing. Moreover, we briefly present the testing ontologies already published in the literature. Section 3 discusses the Ontology Engineering process we followed to develop ROoST. Section 4 presents ROoST. Section 5 presents an operational version of ROoST. Section 6 discusses how ROoST was evaluated. Section 7 compares ROoST with the testing ontologies presented in Section 2. Finally, in Section 8, we present our final considerations.

2. Background

In this section, we briefly present the main concepts related to Software Testing, as well as existing ontologies on software testing.

2.1. Software Testing

To achieve quality software products, it is essential to perform Verification & Validation (V&V) activities throughout the software development process. This is recognized by most international software process quality standards, such as CMMI (CMMI, 2010) and ISO 12207 (ISO, 2008), as well as by the Software Engineering Body of Knowledge (SWEBOK) (Bourque and Fairly, 2014). In particular, there is the ISO/IEC/IEEE 29119 (IEEE, 2013) set of standards that is devoted to software testing.

V&V activities can be static and dynamic. Dynamic V&V activities require the execution of a program, while static V&V activities do not. Static V&V are typically done by means of technical reviews and inspections. Dynamic V&V are done by means of testing (Mathur, 2012). Thus, software testing consists of the dynamic V&V of the behavior of a program on a finite set of test cases, against the expected behavior (Bourque and Fairly, 2014). A test case is a set of inputs, execution conditions and the expected result of a certain program or unit. The simplest definition of software testing is "Testing is the process of executing a program with the intent of finding errors" (Myers, 2004).

Testing activities are supported by a well defined testing process. The testing process consists of several activities, namely: Test Planning, Test Case Design, Test Execution and Test Result Analysis (Bourque and Fairly, 2014); (Ammann and Offutt, 2008); (Black and Mitchell, 2011). Briefly, key aspects of Test Planning include, among others, coordination of personnel, management of available test facilities and equipment, scheduling testing activities, and planning for possible undesirable outcomes. Test Case Design aims at designing the test cases to be executed. Test Cases should be implemented as Test Scripts. During Test Execution, test cases are executed, producing actual results. Finally, in the Test Result Analysis, test results are evaluated to determine whether or not tests have been successful in identifying defects.

Testing process comprises testing activities, and provides a guide to testing teams for supporting achieving test objectives effectively (Bourque and Fairly, 2014). Moreover, techniques, levels, artifacts and environment are also integrated into the testing process. Following, these concepts are briefly discussed:

Testing Techniques: According to Mathur (2012), testing techniques can be classified into: *Black-box Testing Techniques*, which generate test cases relying only on the input/output behavior, without the aid of the code that is under test. Examples of black-box testing techniques include equivalence partitioning and boundary-value analysis; *White-box Testing Techniques*, which use the structure of the code under testing for generating test cases. Control flow, Data flow and Coverage testing are examples of white-box testing techniques; *Defect-based Testing Techniques*, which design test cases aimed at revealing categories of likely or predefined faults, such as Mutation Testing; and *Model-based Testing Techniques*, which are based on requirements formally specified, for example, using one or more mathematical or graphical notations such as Statecharts, Finite State Machines (FSM) and others.

Test Level: Three important test levels can be distinguished, namely: *Unit Testing*, *Integration Testing* and *System Testing*. During *Unit Testing*, the focus is on the unit or the individual components that have been developed. The goal is to ensure that the unit functions correctly in isolation. When units are integrated and a large component or a subsystem formed, programmers perform *Integration Testing*. During *Integration Testing*, the goal is to ensure that a collection of components function as desired. On the other hand, when the entire system has been built, its testing is referred to as *System Testing* (Mathur, 2012). System testing is concerned with the behavior of the entire system (Bourque and Fairly, 2014); (Perry, 2006).

Test Artifact: Test artifacts are produced and used throughout the testing process. Documentation is an integral part of the formalization of the test process (Bourque and Fairly, 2014). According to IEEE (1998), testing artifacts include, among others, Test Plan, Test Procedure, Test Case, and Test Results. During the test planning activity, a *Test Plan* is developed. The *Test Plan* describes how the test should be performed, providing a roadmap for future testing activities. During, the test case de-

sign activity, *Test Cases* are generated. A *Test Case* comprises the input data, expected results, steps and general conditions for running the test case. *Test Cases* are documents. In order to be executed, they should be implemented. Thus, during test coding, *Test Code* is generated. Once implemented, tests can be run. During test execution, *Test Code* is run, and *Test Results* are recorded. Finally, during the test result analysis activity, *Test Results* are analyzed and a *Test Analysis Report* is produced.

Test Environment: According to Perry (2006), software testers are most effective when they work in an environment that encourages and supports well-established testing policies and procedures. The test environment encompasses the entire structure where the test is performed and considers both hardware and software, as well as the human resources involved in testing (Everett and Raymond, 2005); (Perry, 2006).

2.2. Testing Ontologies

The efficiency of the testing process can be improved by reusing testing-related knowledge, since testing is a complex and knowledge intensive process. For instance, during test case design, a test case designer could benefit from reusing past experiences related to choosing which technique to apply, or even by reusing a test case. In order to facilitate communication, integration, search, and representation of testing knowledge, a reference domain ontology on software testing can be used. Such ontology is useful for defining the shared vocabulary to be used in the KM system.

In order to look for a domain ontology in software testing, we conducted a Systematic Literature Review (SLR), including questions related to their coverage of the software testing domain, and how they were developed. A SLR is a form of secondary study that uses a well-defined process to identify, analyze and interpret the available evidences. The research method applied was defined based on the guidelines for SLRs given by Kitchenham and Charters (2007). The referred SLR was published in (Souza et al., 2013).

As a result of the SLR, 12 different ontologies were identified and each one was analyzed. The following testing ontologies were found: Software Testing Ontology for Web Service (STOWS) ((Huo et al., 2003); (Zhu and Huo, 2005); (Hong, 2006); (Yufeng and Hong, 2008); (Zhu and Zhang, 2012)), OntoTest ((Barbosa et al., 2006); (Nakagawa, 2009)), TaaS Ontology ((Yu et al., 2008); (Yu et al., 2009)), Test Ontology Model (TOM) (Bai et al., 2008), MND-TMM Ontology(MTO) (Ryu et al., 2011), and the ontologies proposed in (Li and Zhang, 2012), (Arnicans et al., 2013), (Guo et al., 2011), (Nasser et al., 2009), (Sapna and Mohanty, 2011), (Cai et al., 2009) and (Anandaraj et al., 2011).

Although there are a relatively large number of ontologies on software testing published in the literature (12 ontologies), there are still problems related to the establishment of an explicit common conceptualization with respect to this domain. To analyze these ontologies, we considered some of the characteristics pointed out by d'Aquin and Gangemi (2011) as characteristics that are presented in "beautiful ontologies". In (d'Aquin and Gangemi, 2011), these characteristics are classified along three types of evaluation dimensions: syntactic and formal structure, conceptual coverage and task, and pragmatic or social sustainability. Some of these characteristics are hard to evaluate, since there isn't much information about them in the papers presenting the corresponding ontologies. For instance, in the Pragmatic and Social Sustainability dimension, there are characteristics that are difficult to evaluate based on the papers, such as the result of an evolution; wide usage or acceptance; commercial impact; recommended by industry; applications built on top of it; and successful personal experience in building apps with it. Thus, in our analysis, we considered the following characteristics: (i) a good domain coverage; (ii) implementing an international standard; (iii) formally rigorous; (iv) implementing also non-taxonomic relations; (v) following an evaluation method; and (vi) reusing foundational ontologies.

With respect to domain coverage, we noticed that most ontologies had very limited coverage. Ontologies with higher coverage were: STOWS, OntoTest and TaaS Ontology.

Some of the ontologies considered international standards, namely: OntoTest, TOM and the ontologies proposed by Arnicans et al. (2013), Sapna and Mohanty (2011), and Cai et al. (2009). Others, on the

other hand, did not consider international standards (or at least did not mention them). This is the case of STOWS and TaaS Ontology.

The next two characteristics (being formally rigorous and also implementing non-taxonomic relations) are very important for a reference ontology. Ideally, a reference ontology must be a heavyweight ontology, and thus it must comprise conceptual models that include concepts, and relations (of several natures), and also axioms describing constraints and allowing to derive information from the domain models. Taking this perspective into account, most of the existing ontologies present problems.

Five ontologies (the ones proposed by Guo et al. (2011), Nasser et al. (2009), Ryu et al. (2011), Cai et al. (2009) and Anandaraj et al. (2011)) are just Web Ontology Language (OWL) artifacts (i.e., operational ontologies). The ontologies proposed by Arnicans et al. (2013) and Cai et al. (2009) are, in fact, taxonomies, and thus, they do not qualify as ontologies. STOWS is mainly a set of taxonomies of basic concepts, including some properties and few relations. There are taxonomies of *Tester*, *Context*, *Testing Activities*, *Testing Methods*, and *Testing Artifacts*, but there are important relations missing. For instance, which are the artifacts produced and required by a testing activity? Without relations between the concepts, questions such as this one cannot be answered. Moreover, there are two "compound concepts" in STOWS that are defined on the bases of the basic concepts: capability and task. Capability, for instance, is modeled as a composite entity, which parts are *Activity*, *Method*, an optionally *Environment*, *Context*, and *Data* (a subtype of Artifact). This model is questionable, since it puts together objects and events as part of *Capability*. Objects (or endurants) exist in time; while events (or perdurants) happen in time (Guizzardi and Halpin, 2008). So what is a Capability? An object or an event? This shows that this ontology presents problems.

TaaS Ontology presents very simple models. The Unified Modeling Language (UML) class diagrams presented in (Yu et al., 2008) and (Yu et al., 2009) do not specify multiplicities of the relationships. Moreover, like STOWS, most of the relationships are modeled as aggregations (whole-part relations in UML). This approach is very questionable from an ontological point of view. For instance, there is a core concept called Test Task, which is modeled as composed by *TestActivity*, *TestType*, *TargetUnderTest*, *TestEnvironment* and *TestSchedule*. Similar to the analysis of STOWS, the composite object Test Task aggregates endurants and perdurants.

Although probably the most complete ontology among the ones achieved through the SLR, OntoTest also presents problems. First, there are sub-ontologies that were not published yet, namely the *Testing Process*, *Testing Phase*, *Testing Artifact*, and *Testing Procedure* sub-ontologies. Second, OntoTest does not properly link the concepts to the sub-ontologies. For instance, albeit in the *Main Software Testing Ontology* there is a relationship between *Testing Step* and *Test Resource*, there are not relationships between their subtypes. This is an important part of the software testing conceptualization that needs to be made explicit.

Regarding ontology evaluation, none of the publications in the SLR discussed how the proposed ontologies were evaluated, except the one proposed by Arnicans et al. (2013), who say that a software testing expert has analyzed the ontology fragment related to testing techniques.

Finally, concerning the reuse of foundational ontologies, none of the ontologies analyzed have used one. This can be considered a problem, because important distinctions made in Formal Ontologies may be disregarded as clearly noticed in the brief analysis done (as in the aforementioned cases of STOWS and TaaS Ontology). The lack of truly ontological foundations puts in check the truthfulness of those ontologies.

Thus, as the main finding of the SLR, we concluded that the software testing community has still a lot of work to do, in order to achieve a reference software testing ontology. Once developed a good quality reference testing ontology, an operational version of it can be designed and implemented.

3. Ontology Engineering Approach

In order to develop ROoST, the Systematic Approach for Building Ontologies (SABiO) (Falbo, 2014) was adopted. SABiO was originally conceived for supporting the development of domain reference on-

tologies (Falbo et al., 1998). Currently, SABiO incorporates best practices commonly adopted in Software Engineering and Ontology Engineering, and also addresses the design and coding of operational ontologies (Falbo, 2014). As Figure 1 shows, SABiO development process comprises five main phases, namely (Falbo, 2014):

1. Ontology Purpose Identification and Requirements Elicitation: this phase aims at eliciting ontology requirements, and comprises four activities: Purpose and Intended Uses Identification; Requirements Elicitation; Competency Questions Identification; and Ontology Modularization. These activities are performed in an iterative way.

2. Ontology Capture and Formalization: the main goal of this phase is to capture the domain conceptualization based on the competency questions. SABiO advocates that concepts and relations in a reference domain ontology should be analyzed in the light of a foundational ontology. The relevant concepts, relations, properties and axioms should be identified and organized, and a conceptual model should be built in a graphical language. SABiO suggests the use of OntoUML, a UML class diagram profile that incorporates important foundational distinctions made by the Unified Foundational Ontology (UFO) (Guizzardi, 2005). This phase comprises four activities: Conceptual Modeling, Definition of the Terms in a Dictionary of Terms, Definition of Informal Axioms, and Definition of Formal Axioms.

3. Ontology Design: in this phase the conceptual specification of the reference ontology should be transformed into a design specification by taking into account a number of issues, ranging from architectural to technological non-functional requirements, to target a particular implementation environment. The design phase, thus, aims at bridging the gap between the conceptual modeling of reference ontologies and the coding of them in terms of a particular operational (machine-readable) ontology language (e.g. OWL).

4. Ontology Implementation: this phase implements the ontology in the chosen operational language.

5. Ontology Testing: this phase refers to dynamic verification and validation of the behavior of the operational ontology on a finite set of test cases, against the expected behavior regarding the competency questions. In this sense, SABiO's testing phase is competency questions-driven. In addition, validation testing can be performed by using the operational ontology in actual software applications, according to the intended uses originally proposed for the ontology.

SABiO considers also five supporting processes (see Figure 1) (Falbo, 2014): Knowledge Acquisition, Documentation, Configuration Management, Reuse, and Evaluation.

Knowledge Acquisition occurs mainly in the initial phases of the ontology development process. Conventional methods and techniques for knowledge acquisition and for requirements elicitation applies. Domain experts are the main source for knowledge acquisition. Other important sources of knowledge are consolidated bibliographic material, such as classical books, international standards, glossaries, lexicons, classification schemes, and reference models (Falbo, 2014). For developing ROoST, besides involving domain experts in the ontology development, several references were used, including international standards. The main literature references used for building ROoST were: (IEEE, 1990); (IEEE, 1998); (Myers, 2004); (Bourque and Fairly, 2014); (Pressman, 2006); (Black and Mitchell, 2011); (Mathur, 2012); (IEEE, 2013). The purpose of using different references was to achieve consensus on ROoST's concepts and relations, since an ontology should capture a shared conceptualization.

Results from the ontology development process must be documented. The main documents developed, as well as the source code of the operational ontologies, must have their configuration managed. Thus, once approved, they must be submitted to the **Configuration Management**, where they will be controlled at least concerning changes, versions, and delivery (Falbo, 2014).

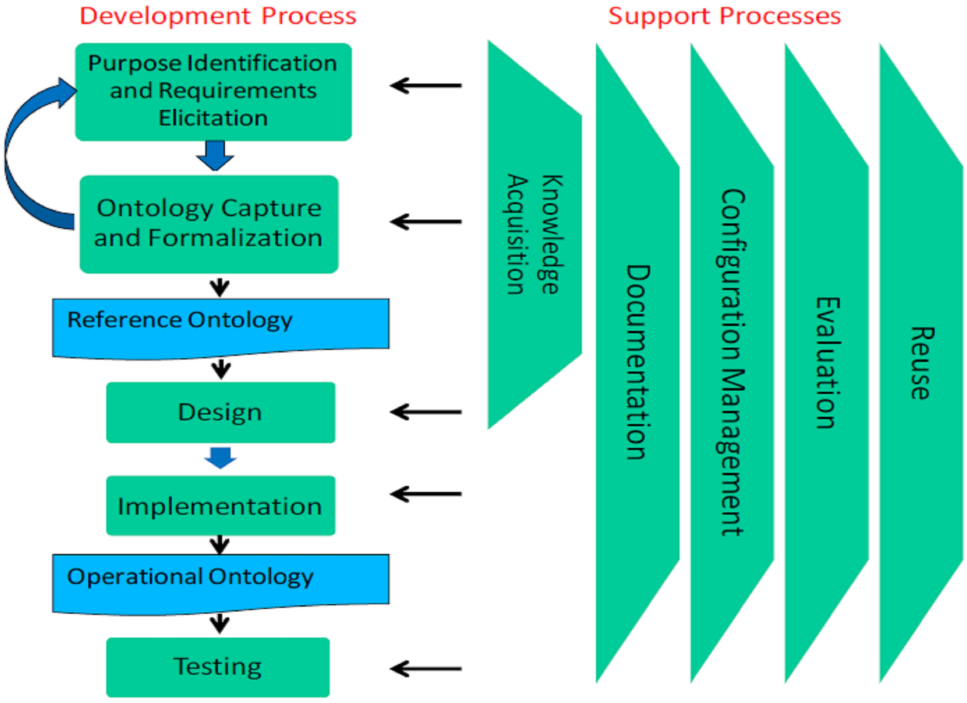

Fig. 1. Systematic Approach for Building Ontologies (SABiO) (Falbo, 2014)

Along the development process, there are many opportunities for reusing conceptualizations already established for the domain in hands. The purpose of the **Reuse Process** is to explore these opportunities. Sources for ontology reuse include: existing domain ontologies, core ontologies, foundational ontologies, and ontology patterns (Falbo, 2014). ROoST has been developed by reusing and extending ontology patterns of the Software Process Ontology Pattern Language (SP-OPL) (Falbo et al., 2013) and the Enterprise-Ontology Pattern Language (E-OPL) (Falbo et al., 2014). An Ontology Pattern Language (OPL) (Falbo et al., 2013) is a network of interconnected domain-related ontology patterns that provides holistic support for solving ontology development problems for a specific domain. An OPL offers a set of interrelated domain patterns, plus a process with explicit guidance on what problems can arise in that domain, informing the order to address these problems, and suggesting one or more patterns to solve each specific problem. SP-OPL is an OPL for the software process domain, addressing aspects such as Standard Process Definition, Project Process Definition and Scheduling, and Process Execution. E-OPL (Falbo et al., 2014) is an OPL for the enterprise domain, addressing aspects such as Organization Arrangement, Team Definition, Institutional Roles, Institutional Goals, and Human Resource Management. Both OPLs are derived from core ontologies that were built grounded in the Unified Foundational Ontology (UFO) (Guizzardi, 2005); (Guizzardi et al., 2008); (Guizzardi et al., 2013).

Finally, with respect to SABiO's **Evaluation Process**, it comprises two main perspectives: (i) *Ontology Verification*: aims to ensure that the ontology is being built correctly, in the sense that the output artifacts of an activity meet the specifications imposed on them in previous activities. (ii) *Ontology Validation*: has an objective to ensure that the right ontology is being built, that is, the ontology fulfills its specific intended purpose. In order to evaluate ROoST, different evaluation approaches were applied, namely: assessment by humans, data-driven evaluation, and application-based evaluation (Brank et al., 2005). For evaluating the reference ontology as a whole (intended purposes, competency questions and conceptual models), expert judgment was performed. For evaluating if the reference ontology is able to represent real world situations, the concepts and relations of ROoST were instantiated using testing data extract from an actual project, in a data-driven approach to ontology evaluation. Furthermore, ROoST was implemented in OWL and the resulting operational ontology was also tested in an ontology testing approach. Finally, we developed a

KM application based on ROoST to support managing software testing knowledge, and ROoST was also evaluated in this context, in an application-based approach to ontology evaluation.

In the next section ROoST is presented. In Section 5, we present an OWL operational version of ROoST. Finally, in Section 6, we discuss how ROoST were evaluated.

4. ROoST: Reference Ontology on Software Testing

As mentioned earlier, ROoST is a reference domain ontology, i.e. a domain ontology built with the main goal of making the best possible description of the testing domain. ROoST was developed for establishing a common conceptualization about the software testing domain, focusing on the testing process, to support the communication between the stakeholders involved in such process. In particular, ROoST is designed to be used as a core conceptual model to be used for several purposes, such as:

- To support human learning on the software testing process: Testing process is very important for software product quality assurance. However, it is often neglected. Practitioners, many times, do not know basic concepts on testing. Thus, ROoST can be used to support learning by humans of the key concepts related to software testing process.

- As a basis for structuring and representing knowledge related to software testing: Testing process is knowledge intensive. For instance, a tester shall know several testing techniques for applying them properly. Test cases can be reused, as well as lessons learned regarding testing. In this context, it is very useful to apply Knowledge Management (KM) techniques. ROoST can be used for structuring a testing knowledge repository, and for supporting searches in it.

- As a reference model for integrating software tools supporting the testing process: Testing is a complex process, and thus requires tool support. Several different types of tools are required to support the software testing process, such as tools supporting test case design, environments for running test cases, issue tracking systems, and control version systems. ROoST can be used to support semantic integration of tools.

- As a reference model for annotating testing resources in a semantic documentation approach: By adding ontology-based annotations to testing documents, we can reach semantic documents, i.e. documents that know about their own content so that automated processes can know what to do with them (Uren et al., 2006). Once semantically annotated, we can extract knowledge and link contents from different documents according to the shared ontology. By merging the content extracted from several documents (including testing documents), we are able to achieve a more holistic view of the knowledge available in a project or organization (Arantes and Falbo, 2010).

In order to cover this scope, ROoST should be able to answer the following competency questions:

CQ01. In which project a given testing process occurred?
CQ02. How is a testing process structured in terms of testing activities and sub-activities?
CQ03. When did a testing process start and when did it end?
CQ04. When did a testing activity start and when did it end?
CQ05. On which activities does a testing activity depend on to be performed?
CQ06. What is the test level of a testing activity?
CQ07. What are the artifacts produced in a testing activity?
CQ08. What are the artifacts used by a testing activity?
CQ09. How do testing artifacts relate to each other?
CQ10. Which are the testing techniques adopted in a testing activity devoted to designing test cases?

CQ11. To which test levels does a testing technique apply?
CQ12. Which are the testing techniques applied to derive a given test case?
CQ13. Which human resources participate in a testing activity?
CQ14. When did a human resource participation start and when did it end?
CQ15. Which hardware resources are used in a testing activity?
CQ16. When did a hardware resource participation start and when did end?
CQ17. Which software resources are used in a testing activity?
CQ18. When did a software resource participation start and when did it end?
CQ19. Which are the hardware, software, and human resources that comprise the testing environment of a project?

Besides the functional requirements for ROoST, posed as competency questions, we also elicit non-functional requirements, namely:

- ROoST should take the main books and standards on software testing into account, in particular ISO/IEC/IEEE 29119 (IEEE, 2013). This non-functional requirement is aligned to the characteristic of beautiful ontologies of being based on international standards.
- Since the main focus of ROoST is on the testing process, it should be integrated to a Software Process Ontology, as well as being grounded in a foundational ontology. This non-functional requirement is aligned to the characteristic of beautiful ontologies of reusing foundational ontologies.

Based on ROoST's functional and non-functional requirements, we decided to use the Software Process Ontology Pattern Language (SP-OPL) (Falbo et al., 2013). SP-OPL has three entry points, i.e. three different ways to start using its patterns. Each entry point defines a pattern in the language that can be used first, independently from other patterns, and defines a way to traverse the OPL. The first entry point (EP1) is to be chosen when the requirements for the domain ontology being developed include problems related to *Standard Process Definition*. The second entry point (EP2) is to be chosen when the requirements for the ontology being developed include problems related to *Project Process Definition and Scheduling*, but standard process definition is out of scope. Finally, the third entry point (EP3) is to be chosen when only problems related to the *Software Process Execution* are to be addressed by the ontology being developed (Falbo et al., 2013). Thus, for developing ROoST, we chose EP3 as entry into SP-OPL, since our interest is to represent knowledge involved in the execution of testing processes. Figure 2 shows the SP-OPL patterns accessible from EP3. From EP3, the first pattern to be applied is *Process and Activity Execution* (PAE) pattern. From this pattern, the ontology engineer can achieve others patterns that address problems related to human resource participation (*Human Resource Participation* - HRPA), hardware and software resource participation *(Resource Participation* - RPA), procedures adopted (*Procedure Participation* - PRPA), and work product inputs and outputs (*Work Product Participation* - WPPA). These patterns were reused.

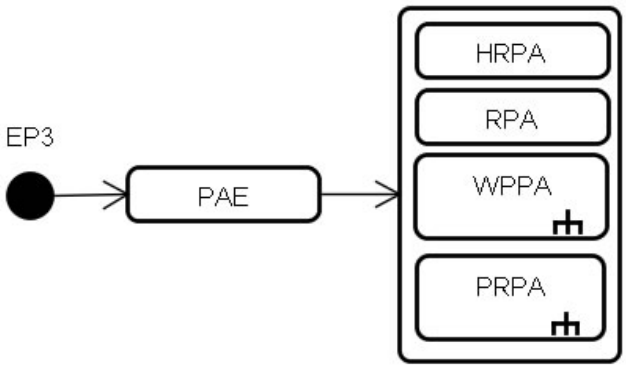

Fig. 2. SP-OPL patterns accessible from the entry point EP3 (Falbo et al., 2013)

With respect to the competency questions, it is important to emphasize that SP-OPL lead to the text for the questions originally defined for ROoST. First, we posed an initial set of competency questions

that should be answered by ROoST. This initial set helped us to select the relevant SP-OPL's patterns. Once patterns of the SP-OPL were applied, their competency questions were reused and adapted to the software testing domain. Finally, in an iterative way, as we developed the conceptual models, new competency questions popped up. Thus, we achieved the set of 19 competency questions previously listed. It is important to highlight that very specific questions about the software testing domain that do not have a counterpart in SP-OPL were also considered. This is the case of *CQ06*, *CQ09*, *CQ10*, *CQ11*, *CQ12* and *CQ19*.

Since the testing domain is complex, ROoST was developed in a modular way, as suggested by SABiO. ROoST has four modules (sub-ontologies). Figure 3 presents a UML package diagram, showing the sub-ontologies that comprise ROoST and the relationships between them. The dependencies between the sub-ontologies indicate that concepts and relations from a sub-ontology are used by the dependent sub-ontology.

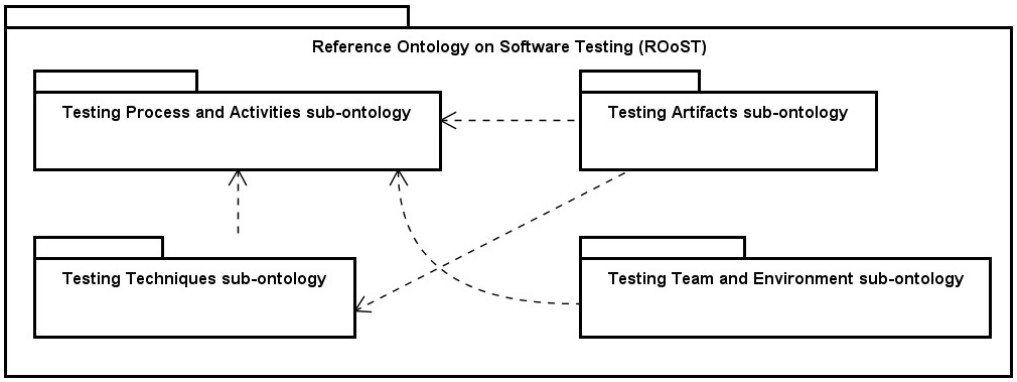

Fig. 3. ROoST: sub-ontologies

In the following subsections, ROoST sub-ontologies are presented, as well as how we applied the reused patterns in their development. Concepts reused from SP-OPL are shown in grey, and they are preceded by the pattern acronym (e.g., PAE::). The conceptual models presented in the sequel are written in OntoUML.

4.1. Testing Process and Activities sub-ontology

This sub-ontology addresses the competency questions *CQ1* to *CQ6*. To answer them, the Process and Activity Execution (PAE) pattern was reused. PAE concepts were extended to the testing domain, as shown in Figure 4. *Testing Process* is a subtype of *Specific Performed Process*, since a testing process occurs in the context of the entire software process (*General Performed Process*) of a *Project*. A testing process, in turn, is composed by testing activities, and thus *Testing Activity* is considered a subtype of *Performed Activity*. Similarly to *Performed Activity*, *Testing Activity* can be further divided into *Composite* and *Simple Testing Activity*.

Besides specializing concepts, relationships were also specialized from PAE. For instance, in PAE, there is a whole-part relationship between *Specific Performed Process* and *Performed Activity*. The whole-part relationship between *Testing Process* and *Testing Activity* is a subtype of the former. Whenever a ROoST relationship is a subtype of another relationship defined in SP-OPL, the same name is used for both.

Looking at the literature (Bourque and Fairly, 2014); (Black and Mitchell, 2011); (Mathur, 2012), it is possible to establish that the testing process consists of, at least, the following activities: *Test Planning*, *Test Case Design*, *Test Coding*, *Test Execution*, and *Test Result Analysis*. Thus, these activities are subtypes of *Testing Activity*. Moreover, *Test Planning* is a *Composite Testing Activity*. Although not shown in Figure 4, test planning involves several sub-activities, such as defining the testing process, allocating people and resources for performing its activities, analyzing risks, and so on. On the other hand, *Test Case Design*, *Test Coding*, *Test Execution* and *Test Result Analysis* are considered *Simple Testing Activities*. Test Planning

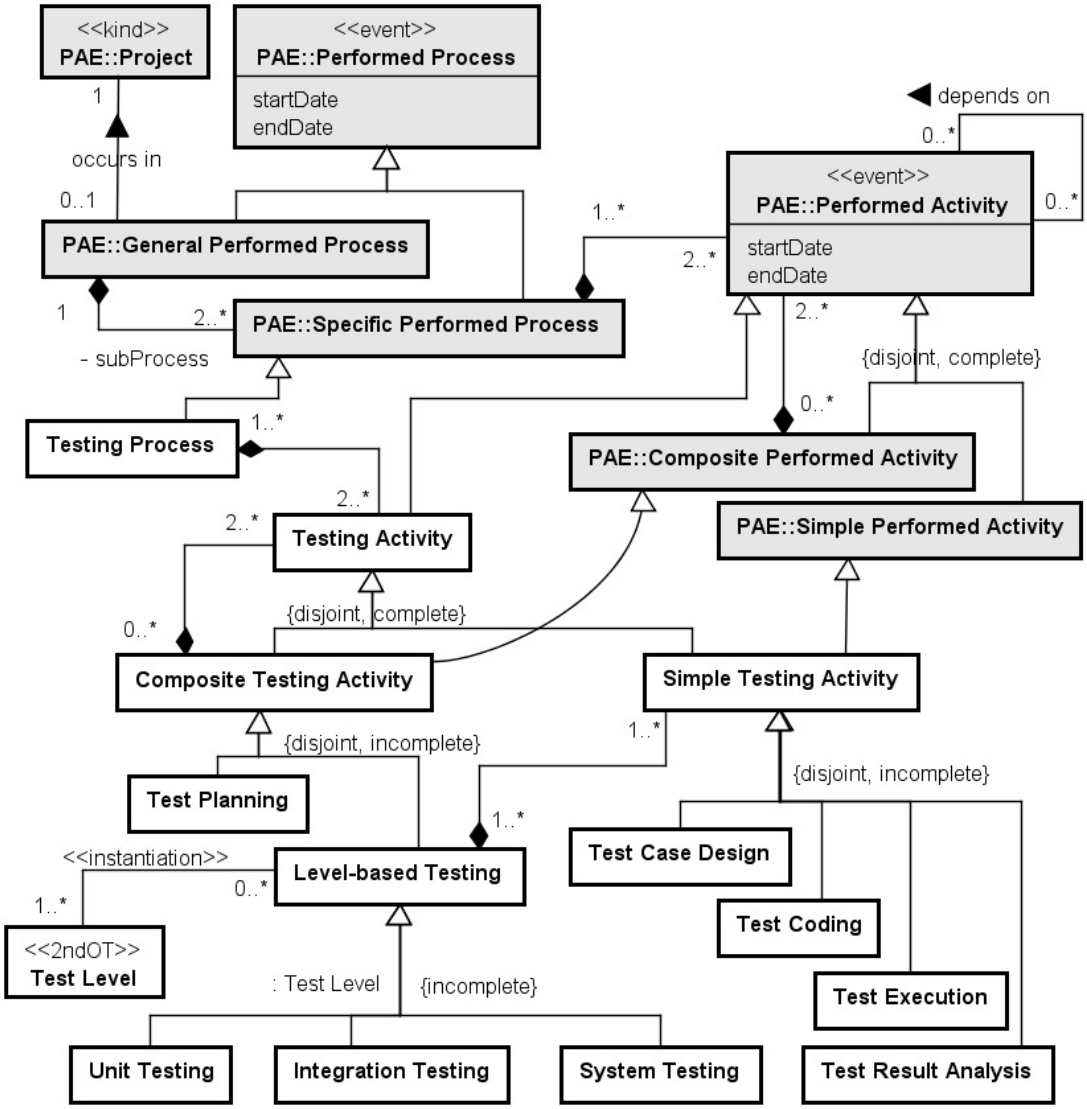

Fig. 4. ROoST's Testing Process and Activities sub-ontology.

activity is not explicitly decomposed into sub-activities in ROoST, since, by inspecting the literature, we could not reach a consensus regarding which sub-activities comprise Test Planning.

Software testing is usually carried out at different test levels (Mathur, 2012). *Simple Testing Activities* are grouped according to the *Test Level* to which they are related, forming *Level-based Testing* activities (*CQ6*). Thus, *Level-based Testing* is a subtype of *Composite Testing Activity*. In Figure 4, the three most cited testing levels in the literature are made explicit, namely: *Unit Testing*, *Integration Testing* and *System Testing*. However, there may be others, such as *Regression Testing*.

To answer *CQ1*, two axioms already defined in PAE were reused. They say that the relationship *occurs in* between *General Performed Process* and *Project* can be extended to the sub-processes and activities that compose the former.

$$\textbf{(A1)} \quad \forall gpp : GeneralPerformedProcess; \quad p : Project, \quad spp : SpecificPerformedProcess \quad occursIn(gpp, p) \wedge partOf(spp, gpp) \rightarrow occursIn(spp, p)$$

$$\textbf{(A2)} \quad \forall spp : SpecificPerformedProcess; \quad p : Project, \quad a : PerformedActivity \quad occursIn(spp, p) \wedge partOf(a, spp) \rightarrow occursIn(a, p)$$

4.2. Testing Artifacts sub-ontology

The Testing Artifacts sub-ontology addresses the competency questions *CQ7* to *CQ9*. To answer *CQ7* and *CQ8*, the Work Product Participation (WPPA) pattern was reused. Figure 5 shows this pattern. According to this pattern, a *Performed Activity* can have as one of its parts an *Artifact Participation*. *Artifact participations* can be of three types: (i) *Artifact Creation*, meaning that the artifact is created during the activity occurrence, and thus it is an output of this activity (the */produces* derived relation); (ii) *Artifact Usage*, meaning that the artifact is only used during the activity, and thus it is only an input for the activity (the */uses* derived relation); and (iii) *Artifact Change*, meaning that the artifact is changed during the activity, and thus it is both input and output of the activity. Since in ROoST we are not interested in modeling the events representing the artifact participations, but only which artifacts were used and produced by a testing activity, only the derived relationships */uses* and */produces* are modeled, instead of modeling the artifact participations.

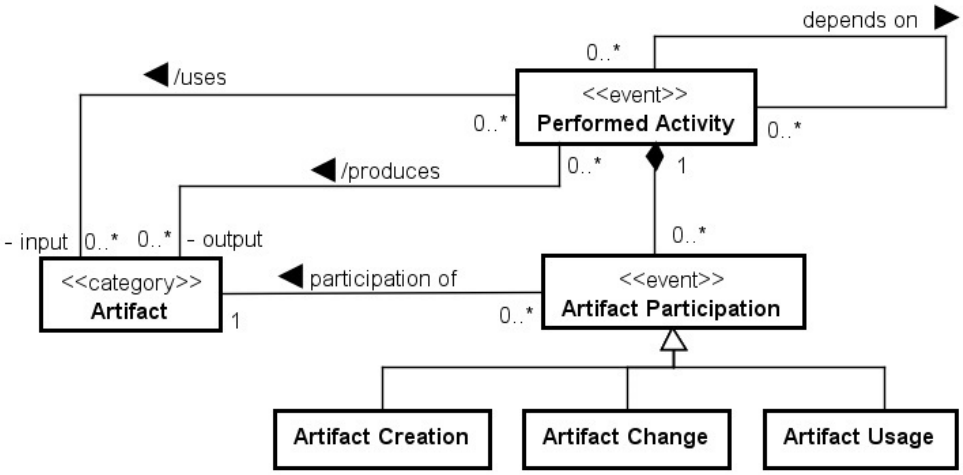

Fig. 5. The "Work Product Participation" (WPPA) pattern

An important issue for ROoST is to describe the types of artifacts that are produced and used during the testing process. Thus, a pattern not originally considered in SP-OPL was proposed and applied: the Work Product Types (WPT) pattern. In WPT, a taxonomy of software artifacts is defined including, among others, the following types of artifacts: *Document*, which refers to artifacts consisting of textual statements usually associated with organizational patterns that define how they should be produced; *Software Item*, referring to a piece of software, produced during the execution of a software process, but not considered a complete Software Product, being an intermediate result. *Code*, which concerns to portions of code written in a programming language; and *Information Item*, referring to data used or produced during the software process. *Artifact Type* is a second order type, whose instances include Document, Software Item, and Information Item, among others.

During the software testing process, several artifacts are used and produced. An important issue for ROoST is to precisely define the relationships between testing activities and testing artifacts (*CQ7* and *CQ8*), as well as the relationship between the artifacts (*CQ9*). In order to make this part of the testing domain conceptualization explicit, the relationships *uses* and *produces* from WPPA are extended to link testing artifacts to the corresponding testing activities in which they are produced or used. Moreover, relationships between the testing artifacts are defined, as shown in Figure 6.

During *Test Planning*, a *Test Plan* is produced. In *Test Case Design*, different artifacts are used for deriving test cases, such as requirements specifications, diagrams, programs, and so on. *Artifacts* used for deriving test cases play the role of *Test Case Design Input*. The main outputs of a *Test Case Design* activity

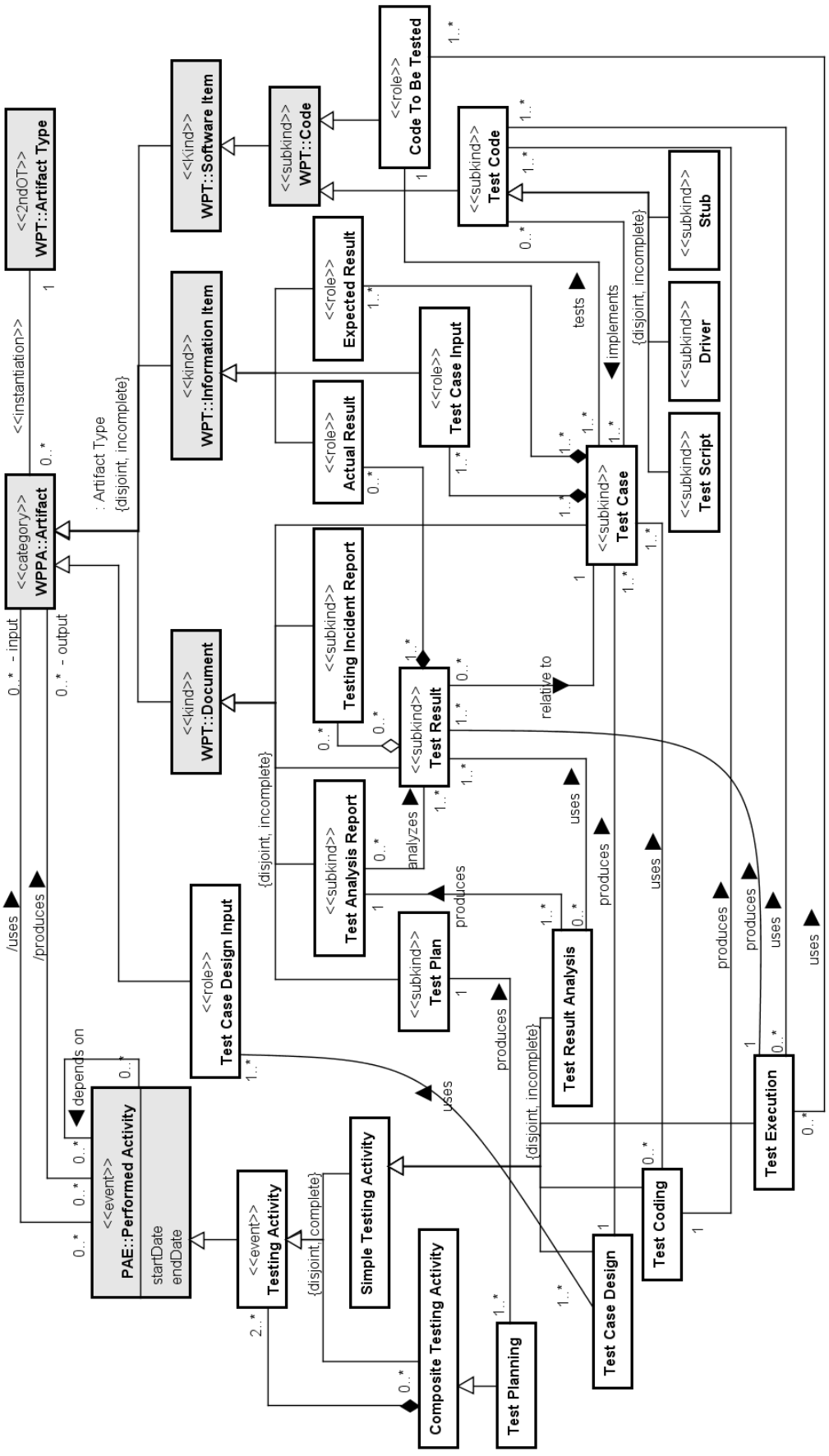

Fig. 6. The ROoST's Testing Artifacts sub-ontology.

are *Test Cases*. A *Test Case* aims to test a *Code To Be Tested*, and specifies the *Test Case Input* and the *Expected Result*.

Test Case Input and *Expected Result* are roles played by *Information Items* in a test case, and are part of it. Whatever code fragments (such as programs, modules, and the whole system code) that have a *Test Case* designed for them play the role of *Code To Be Tested*. It is worth highlighting that "role" in this work is used in the sense of UFO, i.e. a role is an anti-rigid specialization of a sortal such that the specialization condition is a relational one (Guizzardi, 2005). Consider, as an example, the role *Code To Be Tested*. It is an anti-rigid specialization of *Code* (a subkind in UFO), such that the specialization condition is to be the target code of a *Test Case* (*tests* relation). The relational property of being the code to be tested by a *Test Case* is part of the very definition of the role *Code To Be Tested*. Whenever a concept in ROoST is stereotyped with <<role>>, this view applies.

During a *Test Coding*, *Test Cases* are implemented as *Test Code*. *Test Code* is a portion of code that is to be run for executing a given set of test cases. There are different subtypes (*subkind*) of *Test Code*, among them: *Test Script*, *Driver* and *Stub*. A *Test Script* comprises a sequence of actions for the execution of a *Test Case*. A *Driver* is a test code used to invoke a module under test. A driver typically provides inputs, controls and monitors the execution of the module being tested, and reports test results. A *Stub* is a test code that is used as a proxy for a software module. The stub is used to test another component or module that calls the stub or otherwise depends on it.

Test Execution requires as input the *Test Code* to be run and the *Code To Be Tested*. If a *Test Execution* executes a *Test Case*, then the *Test Case* should use a *Test Code* that implements the *Test Case*, and *Test Execution* should also use a *Code to be Test* that is tested by the *Test Case* (see Axiom A3). As an output of this activity, *Test Results* are produced. A *Test Result* is relative to a *Test Case*. Following this relationship, it is possible to know the *Test Case Input* and *Expected Result* to which an *Actual Result* must be compared during *Test Result Analysis* (see axiom A4). *Actual Result* is the role played by an *Information Item* when it is part of a *Test Result*.

(A3) $\forall te : TestExecution, \quad tc : TestCase \quad executes(te, tc) \rightarrow (\exists tco : TestCode, \quad ctbt : CodeToBeTested \quad uses(te, tco) \wedge implements(tco, tc) \wedge uses(te, ctbt) \wedge tests(tc, ctbt))$

(A4) $\forall ar : ActualResult, \quad er : ExpectedResult \quad comparedWith(ar, er) \rightarrow (\exists tc : TestCase, tr : TestResult \quad partOfTestResult(ar, tr) \wedge relativeTo(tr, tc) \wedge \quad partOfTestCase(er, tc))$

A test execution can achieve a result (*Actual Result*), but it can also fail. A *Testing Incident Report* reports an incident. Incidents may be defects or bugs, but may also be perceived problems, anomalies that are not necessarily defects. In an incident, what is initially recorded is the information about the failure (not about the defect) that was generated during test execution. The information about the defect that caused that failure would come to light when someone (e.g. a developer) begins to look into the failure, but this is out of the scope of software testing. A *Testing Incident Report* is used to register any event found during the execution of a software test that requires investigation. Thus, a *Test Result* contains either an *Actual Result*, or a *Testing Incident Report*, or both. Moreover, a *Test Result* must include one of them, as defined by the following axiom:

(A5) $\forall tr : TestResult \rightarrow \exists art : Artifact(ActualResult(art) \vee TestingIncidentReport(art)) \wedge partOf(art, tr))$

Finally, during a *Test Result Analysis*, *Test Results* are analyzed and a *Test Analysis Report* is produced.

WPPA pattern also defines an important axiom for ROoST to answer *CQ5*. This axiom says that if an artifact *art* is an output of an activity *a1*, and *art* is also an input to another activity *a2*, then *a2* depends on *a1*.

(A6) $\forall a1, a2 : PerformedActivity, \quad art : Artifact \quad (produces(a1, art) \wedge uses(a2, art) \rightarrow dependsOn(a2, a1))$

From this axiom, it is possible to infer important dependencies between testing activities, namely: *Test Coding* depends on *Test Case Design*; *Test Execution* depends on *Test Coding*; *Test Result Analysis* depends on *Test Execution*. Moreover, the *depends on* relationship is transitive (A7). Thus, we can also infer, for instance, that *Test Execution* depends on *Test Case Design*.

(A7) $\forall a1, a2, a3 : PerformedActivity \quad (dependsOn(a3, a2) \wedge \quad dependsOn(a2, a1) \rightarrow dependsOn(a3, a1))$

4.3. Testing Techniques sub-ontology

This sub-ontology addresses the competency questions *CQ10* to *CQ12*. In order to answer them, the Procedure Participation (PRPA) pattern was reused. According to the PRPA pattern, *Procedures* may have been adopted to support the accomplishment of *Performed Activities*. Analogously to WPPA, PRPA includes a concept for the events representing the procedure participations in performed activities. However, since in ROoST we are not interested in representing those events, but only the procedures adopted by a testing activity, only the relationship */adopts* is modeled, as Figure 7 shows.

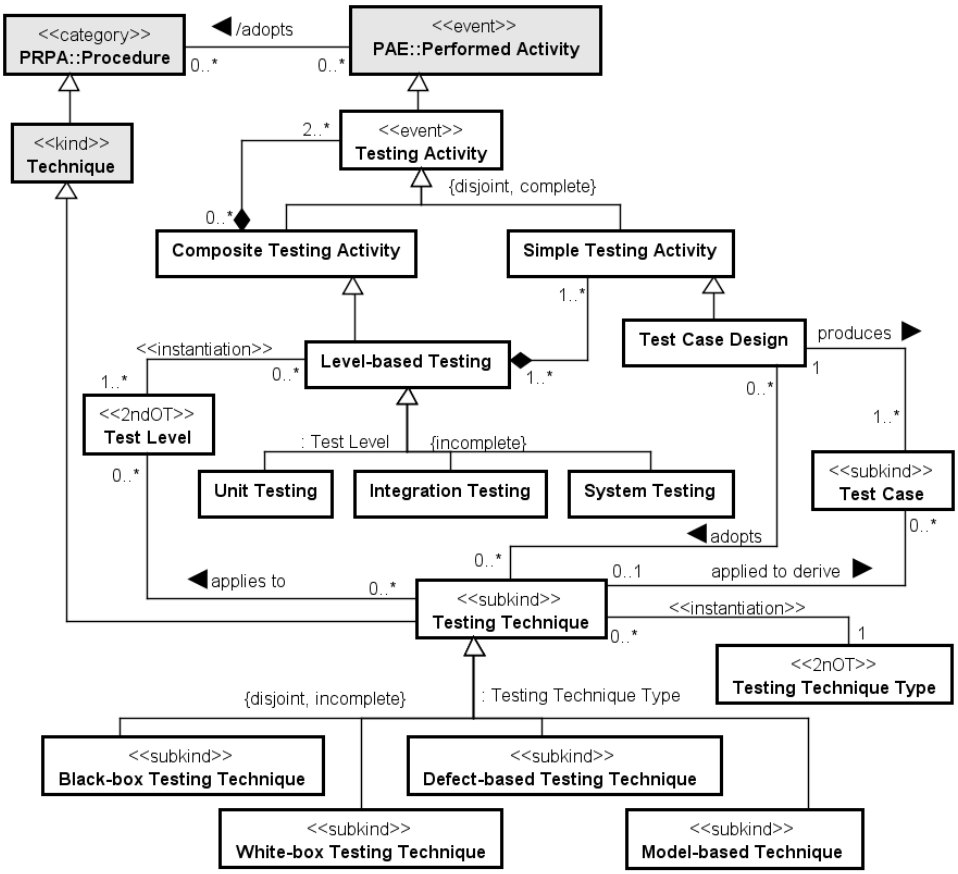

Fig. 7. ROoST's Testing Techniques sub-ontology.

To answer *CQ10*, one subtype of *Procedure* is introduced: *Technique*. This concept is further extended as *Testing Technique*. There are several subtypes of *Testing Technique*, among them: *Black-box*, *White-box*, *Defect-based*, and *Model-based Testing Techniques*. These testing techniques can be adopted by activity occurrences of the type *Test Case Design*.

Some testing techniques are more appropriate to certain test levels. To answer *CQ11*, a relationship between *Testing Technique* and *Test Level* is introduced. *Black-box Testing Techniques*, for example, apply

to all test levels. Most *White-box Testing Techniques*, on the other hand, are suitable only for *Unit Testing*, although some also apply to *Integration Testing*. In general, they are not suitable for *System Testing*, because it is difficult in practice to derive test cases based on the source code when the entire system is considered (Mathur, 2012). To represent this constraint that a *Level-Based Testing* can only adopt *Testing Techniques* applicable to the corresponding *Test Level*, axiom (A8) is defined. Note that *Unit Testing*, *Integration Testing*, and *System Testing* are typical instances of *Test Level*, which is the criterion for the generalization set of *Level-based Testing*.

$$\textbf{(A8)} \quad \forall tcd : TestCaseDesign, \quad lbt : LevelBasedTesting, tt : TestingTechnique, \quad tl : TestLevel \quad adopts(tcd, tt) \wedge partOfLevelBasedTesting(tcd, lbt) \wedge instanceOf(lbt, tl) \rightarrow appliesTo(tt, tl)$$

Finally, for designing a specific test case, a testing technique is applied. Thus, for answering *CQ12*, we added a relationship between *Testing Technique* and *Test Case* to link a test case to the testing technique applied in its design. If a testing technique was used in a certain test case design, then the test case design activity that produced this test case should have adopted this testing technique, as defined by the following axiom:

$$\textbf{(A9)} \quad \forall tc : TestCase, \quad tt : TestingTechnique, \quad tcd : TestCaseDesign \quad designedAccordingTo(tc, tt) \quad \wedge \quad produces(tcd, tc) \rightarrow adopts(tcd, tt)$$

4.4. Testing Team and Environment sub-ontology

The Testing Environment sub-ontology addresses the competency questions *CQ13* to *CQ19*. Figure 8 shows the conceptual model of this sub-ontology. It was developed using the patterns *Human Resource Participation* - HRPA, and *Resource Participation* - RPA. According to the RPA pattern, during a *Performed Activity*, *Resources* are used. In this pattern, two important types of resources are considered, since they are very relevant in the context of software processes: *Hardware Resource* refers to the use of a *Hardware Equipment* in an activity, and *Software Resource* refers to the use of a *Software Product* in an activity.

A *Test Environment* is defined for a *Project* and is composed by *Test Hardware Resources* and *Test Software Resources*. *Test Hardware Resources* and *Test Software Resources* are the roles played by a *Hardware Resource* and a *Software Resource*, respectively, when they are used by a *Testing Activity*. *Testing Activities* also uses the whole *Test Environment*.

Testing Activities are performed by *Human Resources*. A *Human Resource* can play different testing roles (*CQ13*), such as *Test Manager*, *Test Case Designer* and *Tester*.

For properly addressing aspects related to human resource organization in test teams, we decided to use the Enterprise Ontology Pattern Language (E-OPL) (Falbo et al., 2014). We chose the first entry point of E-OPL, and follow a path through the following patterns: Simple Organization Arrangement (SOAR), Organizational Team Definition (OTD), Team Roles (TEAR), and Team Allocation (TEAA). Concepts reused from E-OPL are shown in yellow in Figure 8, and they are preceded by the pattern acronym (e.g., SOAR::).

By applying the above mentioned patterns, we are able to model aspects related to teams. An *Organization* might have *Organizational Teams*. An organizational team is composed by human resources. A *Team Allocation* links the *Team Members* (the role a *Human Resource* plays when she is allocated to a Team) to the *Team*, in a given period of time. Moreover, *Team Allocation* also establishes the *Human Role* that the team member must play in that *Team Allocation*.

By extending this enterprise model to the context of test teams, we capture that *Test Team* is a subtype of *Organizational Team*. A *Test Team Allocation* links a *Test Team Member* to a *Test Team*, in a given period of time (*CQ14*), establishing the *Testing Role* that she must play in that *Test Team Allocation*.

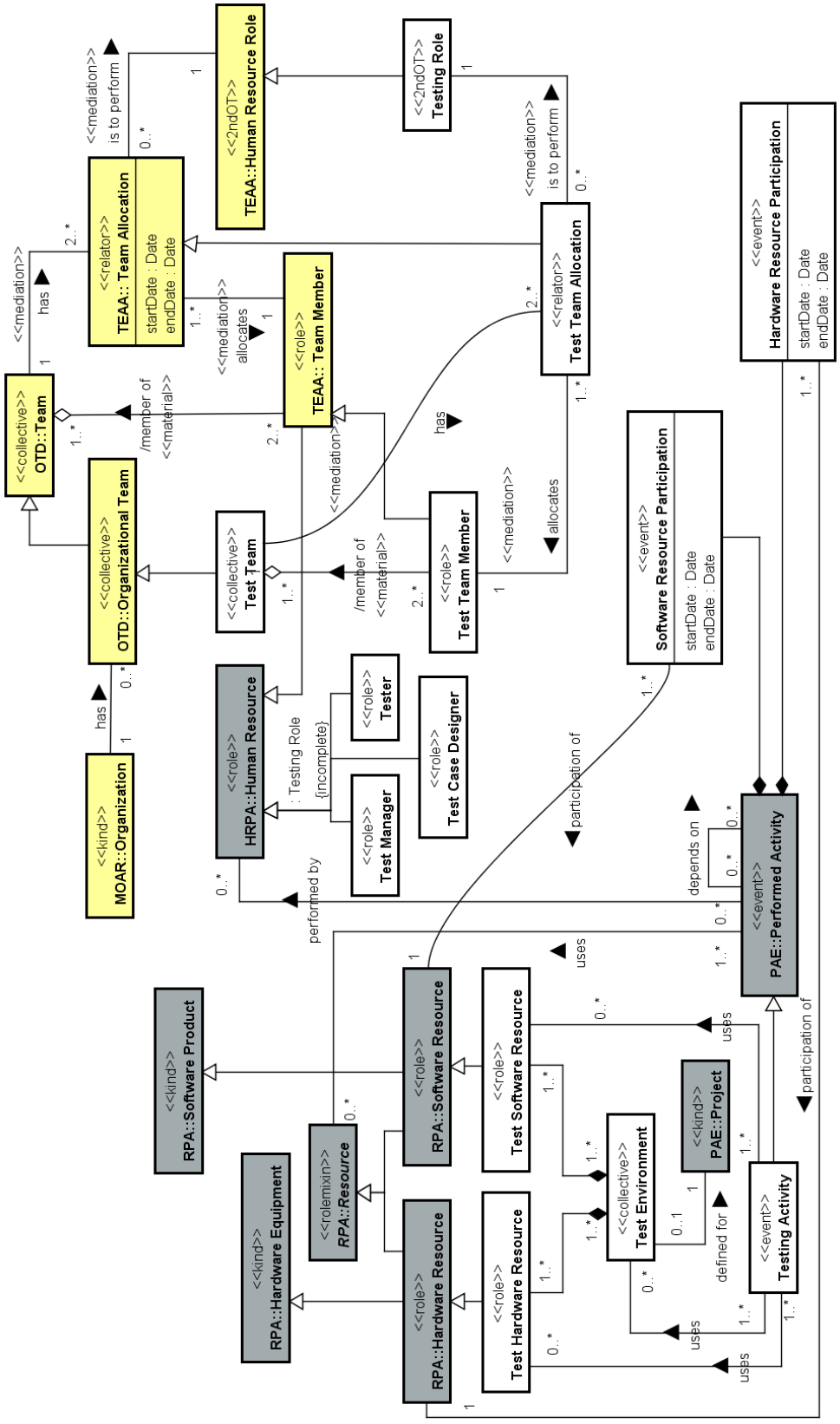

Fig. 8. ROoST's Testing Environment sub-ontology.

5. An OWL Operational Version of ROoST

Reference ontologies are to be used in an off-line manner to assist humans in tasks such as meaning negotiation and consensus establishment. If its machine-readable version is required to meet some intended use, an operational version must be designed and implemented. The same reference ontology can be used to produce a number of different operational versions, each one considering a target language/environment. Unlike reference ontologies, operational ontologies are not focused on representation adequacy, but are designed with the focus on guaranteeing desirable computational properties. A design phase, thus, is necessary to bridge the gap between the conceptual modeling of reference ontologies and the coding of them in terms of a specific operational ontology language (such as, for instance, OWL, RDFS, F-Logics). Issues that should be addressed in the design phase are, for instance: determining how to deal with the differences in expressibility of the languages that are used in each of these phases; or how to produce lightweight specifications that maximize specific non-functional requirements, such as reasoning performance (Guizzardi, 2007); (Falbo et al., 2013).

Since there are intended uses of ROoST that require its operational version (such as to serve as basis for integrating software tools supporting the testing process or for annotating testing resources in a semantic documentation approach), we need to continue SABiO's development process, and design and implement it in a machine-readable language. We chose OWL as target operational language, since it is the most used language in the scenarios we intend to use this operational version, named here OWL-ROoST. For implementing OWL-ROoST, we follow the transformation rules from OntoUML to OWL proposed in (Barcelos et al., 2013). These rules guide the transformation of OntoUML concepts and relations to OWL classes and properties.

Artifact exemplifies the case of a concept of the type <<category>>. According to UFO (Guizzardi, 2005), a category groups rigid instances of classes with different principles of identity, in our case, instances of classes *Document*, *InformationItem* and *Code*, which are kinds (and thus, are stereotyped with <<kind>>). *Artifact* is a superclass of *Document*, *InformationItem* and *Code*. Concepts of the type <<kind>> are mapped as disjoint subclasses. The concepts of the type <<subkind>> are also mapped as subclasses of their respective superclasses. For example, *Document* is a <<kind>> and has as <<subkind>> types the following documents: *Test Plan, Test Result, Test Case* and *Testing Incident Report*. These concepts are mapped to disjoint OWL subclasses, as shown in Figure 9. In this figure, we present part of the transformation discussed above, considering test-related Document subtypes and following the same format that transformations are present in (Barcelos et al., 2013).

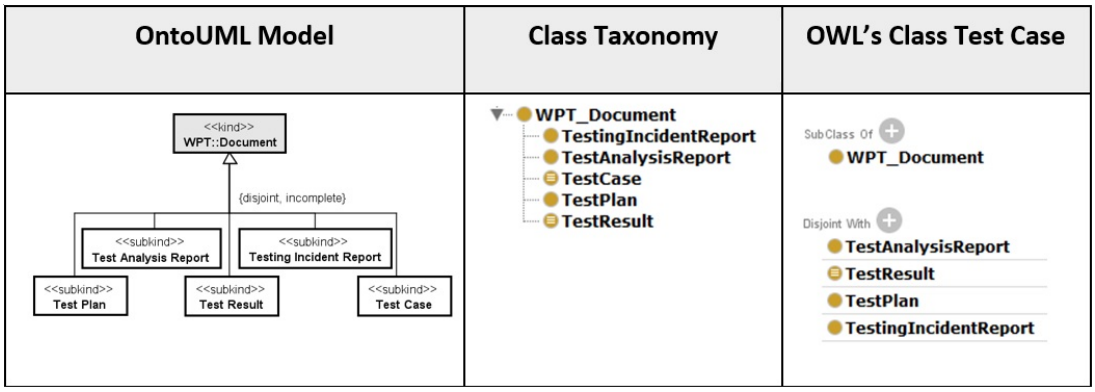

Fig. 9. Example of transformation to OWL

6. ROoST Evaluation

In order to evaluate ROoST, we performed Ontology Verification & Validation (V&V) activities. Considering the guidelines proposed by SABiO, ROoST was evaluated in four steps. First, we performed a verification activity by means of expert judgment, in which we checked if the concepts, relations and axioms defined in ROoST are able to answer its competency questions, in an **assessment by human approach to ontology evaluation** (Brank et al., 2005). Next, since a reference ontology should be able to represent real world situations, to validate ROoST, we instantiated its concepts and relations using testing data extracted from an actual project, in a **data-driven approach to ontology evaluation** (Brank et al., 2005). These two evaluation steps were performed manually, considering the reference ontology. ROoST was then implemented in OWL, and the resulting operational ontology (OWL-ROoST) was also tested in an **ontology testing approach to ontology evaluation** (Vrandecic and Gangemi, 2006). Test cases were designed and exercised in the context of a sub-ontology, in order to check if OWL-ROoST is able to answer the competency questions. Finally, since we developed an application based on ROoST to support managing software testing knowledge, by testing this application with end users, we also evaluated ROoST, in an **application-based approach to ontology evaluation** (Brank et al., 2005). Following each one of these four evaluation steps are described in more details.

***Evaluation Step 1* - Assessment by human approach to ontology evaluation**

ROoST evaluation started with a verification activity, when we manually checked if the concepts, relations and axioms defined in ROoST are able to answer its competency questions (CQs). This approach enabled us to check not only if the CQs were answered, but also whether there were irrelevant elements in the ontology, i.e. elements that do not contribute to answer any of the questions. Table 1 illustrates this verification process, showing which elements of the ontology (concepts, relations, properties and axioms) answer each one of the Competency Questions (CQs) of the Testing Artifacts sub-ontology. Similar approach was applied to the other sub-ontologies.

Table 1: Verifying ROoST concepts, relations and axioms

CQ	Concepts, *Relations* and Properties	Axioms
CQ7	Test Planning *produces* Test Plan Test Case Design *produces* Test Case Test Coding *produces* Test Code Test Execution *produces* Test Result Test Result Analysis *produces* Test Analysis Report	-
CQ8	Test Case Design *uses* Test Case Design Input Test Coding *uses* Test Case Test Execution *uses* Test Code Test Execution *uses* Code To Be Tested Test Result Analysis *uses* Test Result	A3
CQ9	Test Case Input and Expected Result are *part of* Test Case Test Case *tests* Code To Be Tested Test Code *implements* Test Case Test Result *is relative to* Test Case Actual Result and Issue are *part of* Test Result Test Analysis Report *analyzes* Test Result	A4, A5

This evaluation step was performed in parallel with the ontology development. We continuously evaluated if the competency questions were being answered. Moreover, many competency questions were included in the ontology, as we judged that new concepts and relations were in the scope of the ontology. In other words, we followed an iterative approach, merging requirements identification, conceptual modeling and evaluation steps, until we achieved a set of competency questions that were all addressed by the concepts, relations and axioms defined in the ontology.

Evaluation Step 2 - Data-driven approach to ontology evaluation

In order to check if ROoST is able to represent concrete situations of the real world, we instantiated its concepts and relations using testing data extracted from an actual project, called Amazon Integration and Cooperation for Modernization of Hydrological Monitoring Project (ICAMMH Project) (Braga et al., 2009). Table 2 shows part of the instantiation done.

Table 2: ROoST Instantiation

Concept	**Instance**
Project	ICAMMH Project
Testing Process	ICAMMH Testing Process
Black-box Testing Technique	Equivalence partitioning, Boundary-value analysis (black-box techniques applied to derive test cases in the ICAMMH Project)
Test Case	Test Case *P01-256* [Collected by electronic media - Invalid date] (a test case produced in the ICAMMH Project)
Test Case Design Input	Use Case Specification "SAD_MCU_001-Customize Data Collection" (artifact that was used to derive the test case *P01-256*)
Test Case Input	2009-15-11 [Year-month-day/file .txt with month invalid for data collection in header] (input data to the test case *P01-256*)
Code To Be Tested	CollectFormUtil.java (Java class that is to be tested by the test case P01-256)
Test Code (Test Script)	*P01-256 Script* (a test script that implements the test case P01-256)
Actual Result	"Invalid data"

As a result from this evaluation step, we could conclude that ROoST is able to represent real world situations. ICAMMH is a large and complex project, with a great concern for testing. Thus, we expect that the data extracted from it are representative of real world situations. However, we are aware that, since different projects can perform testing in different ways and levels, the examples used in the data-driven evaluation could influence the generality of the evaluation results. For instance, many organizations consider regression testing (testing that verifies that software previously developed and tested still performs correctly after it was changed) as a testing level. ROoST does not explicitly represent regression testing as a testing level, although it is flexible enough to admit this approach. In fact, this motivated us to perform an application-based approach for evaluating ROoST, to get a wider feedback.

Evaluation Step 3 - Ontology Testing

For testing the operational version of ROoST (OWL-ROoST), test cases were designed from the competency questions, in a competency question-driven approach for ontology testing. According to (Falbo,

2014), in this approach, a test case comprises an implementation of a competency question (the specification to be tested) as a query in the chosen implementation environment, plus instantiation data from a fragment of the ontology being tested (input), and the expected results based on the considered instantiation (expected output).

It is important to notice that some of our competency questions (CQs) were defined in a high level of abstraction, which may hinder the design of test cases. Thus, depending on the abstraction level of a CQ, we derived more specific questions from it. So, when necessary, the CQs were rewritten in a format closer to a query script, emphasizing the names of concepts, properties and relations. For example, let us consider CQ09:"*How do testing artifacts relate to each other?*". Since in ROoST there is no general relationship between testing artifacts, to answer CQ09, we need to analyze the conceptual model of ROoST, capture all the relationships between testing artifacts, and create a CQ for each of them. As a result, we identified the following (low abstraction level) CQs:

CQ09.1 - What are the test case inputs of a given test case?
CQ09.2 - What are the expected results of a given test case?
CQ09.3 - What is the code to be tested by a given test case?
CQ09.4 - What are the test codes that implement a given test case?
CQ09.5 - What are the test results relative to a given test case?
CQ09.6 - What are the actual results that comprise a given test result?
CQ09.7 - What are the issues reported in a given test result?
CQ09.8 - What are the test results reported in a test analysis report?

Finally, for each specific CQ, we developed a set of test cases, by implementing the CQs as SPARQL (SPARQL Protocol and RDF Query Language) queries. We chose SPARQL, because it can be used to query both RDF Schema and OWL model to filter out individuals with specific characteristics. Table 3 shows some CQs implemented as SPARQL queries. Table 4 presents some of the test cases that we developed. For performing the test cases, instances considered in the previous evaluation step were added in the corresponding OWL files.

Table 3: SPARQL Queries for the Competency Questions.

Id	CQ	SPARQL Query

CQ01	In which project a given testing process occurred?	`SELECT ?TestingProcess ?Project ?Name`
		`WHERE {`
		`?TestingProcess roost:occurredIn ?Project.`
		`?TestingProcess roost:hasName ?Name.`
		`FILTER(?Name = "Add project name").`
		`}`

CQ07	What are the artifacts produced in a testing activity?	`SELECT DISTINCT ?Activity ?Artifact ?Name`
		`WHERE {`
		`?Activity roost:produces ?Artifact.`
		`?Activity roost:hasName ?Name.`
		`FILTER (?Name = "Add activity name").`
		`}`

Continues

Table 3: Generated test cases

Id	CQ	SPARQL Query
CQ09.01	What are the test case inputs of a given test case?	`SELECT ?TestCaseId ?TestCase ?TestCaseInput WHERE { ?TestCase roost:hasInputs ?TestCaseInput. ?TestCase roost:hasId ?TestCaseId. FILTER (?TestCaseId = Add test case Id). }`
CQ09.02	What are the expected results of a given test case?	`SELECT ?TestCaseId ?TestCase ?ExpectedResult WHERE { ?TestCase roost:hasExpectedResults ?ExpectedResult. ?TestCase roost:hasId ?TestCaseId. FILTER (?TestCaseId = Add test case Id). }`
CQ09.03	What is the code to be tested by a given test case?	`SELECT ?TestCaseId ?TestCase ?CodeToBeTested WHERE { ?TestCase roost:hasCodeToBeTested ?CodeToBeTested. ?TestCase roost:hasId ?TestCaseId. FILTER (?TestCaseId = Add test case Id). }`
CQ12	What are the testing techniques applied to derive a given test case?	`SELECT ?TestCaseId ?TestCase ?TestingTechniques WHERE { ?TestCase roost:derivedApplying ?TestingTechniques. ?TestCase roost:hasId ?TestCaseId. FILTER (?TestCaseId = Add test case Id). }`

Table 4: Test Cases

Test Case Id	CQ	Inputs	Expected Result
T01.01	CQ01	ICAMMH Testing Process	ICAMMH Project
T07.01	CQ07	ICAMMH Test Planning Activity	ICAMMH Test Plan
T09.01	CQ09.01	P01-256	2009-15-11
T09.02	CQ09.02	P01-256	“Invalid data”
T09.03	CQ09.03	P01-256	CollectFormUtil.java (Java class)
T12.01	CQ12	P01-256	Black-box Testing

Finally, after executing a test case, we compared the returned results with the expected results to de-

termine whether the test case passed or failed. If the results match, then OWL-ROoST passed in this test case. Otherwise, we need to analyze if the problem is in the conceptual model of ROoST, in its implementation (OWL-ROoST), or even in the formulation/implementation of the CQs. For running the test cases, we used Protégé. Figure 10 shows an example of the execution of the test case T12.01. As we can see by contrasting the actual result with the expected result, this test case passed.

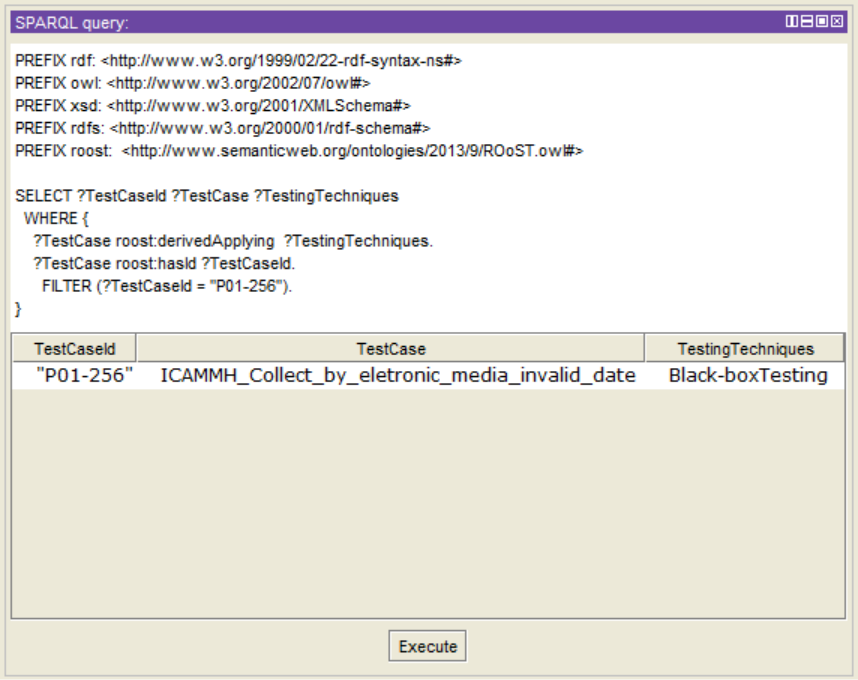

Fig. 10. Example of test case execution

In general, the test cases worked out well. Several problems were detected, most of them related to the implementation of OWL-ROoST, and a few related to the implementation of the test cases. When a problem was detected, we made the necessary changes, and reran the test case. During this evaluation step, we did not detect any problem in the conceptual models of ROoST.

Evaluation Step 4* - *Application-based approach to ontology evaluation

As already said before, we developed a KM system to support managing software testing knowledge, called Testing Knowledge Management Portal (TKMP). TKMP is a web application that was developed using ROoST for structuring its knowledge repository. TKMP was built extending a more general Software Engineering KM Portal, which provides more general KM features such as yellow pages and features for managing lessons learned. Yellow pages map professional capabilities, skills and interests of the organization's members. Lessons Learned are positive or negative experiences distilled from projects that should be taken into account in future projects. Thus, TKMP provides general functionalities for creating, evaluating, searching, retrieving, and valuing knowledge items.

Managing testing knowledge is not an easy task, and thus it is better to start with a small-scale initiative. So, we performed a survey to define a scenario for applying KM in software testing (Souza et al., 2015b). The purpose of our survey was to identify which is the most appropriate scenario in the software testing domain, from the point of view of testing stakeholders, for starting a KM initiative. Considering the main findings of the survey, test case design was considered the software testing activity to be first supported, and test cases the main knowledge item to be managed. So, only the relevant information for designing test cases was considered in the scope of TKMP development.

From TKMP, testers can make a query for retrieving test cases and reuse them. TKMP's search feature allows testers to search related projects, informing the desired combination of parameters to refine their queries, such as test level, testing technique, and information about valuations previously done by testers about the test cases. Figure 11 presents the results of TKMP's search functionality for a query using the following parameters: Project - ICAMMH; Test Level - Unit Testing; Testing Technique - Equivalence Partitioning.

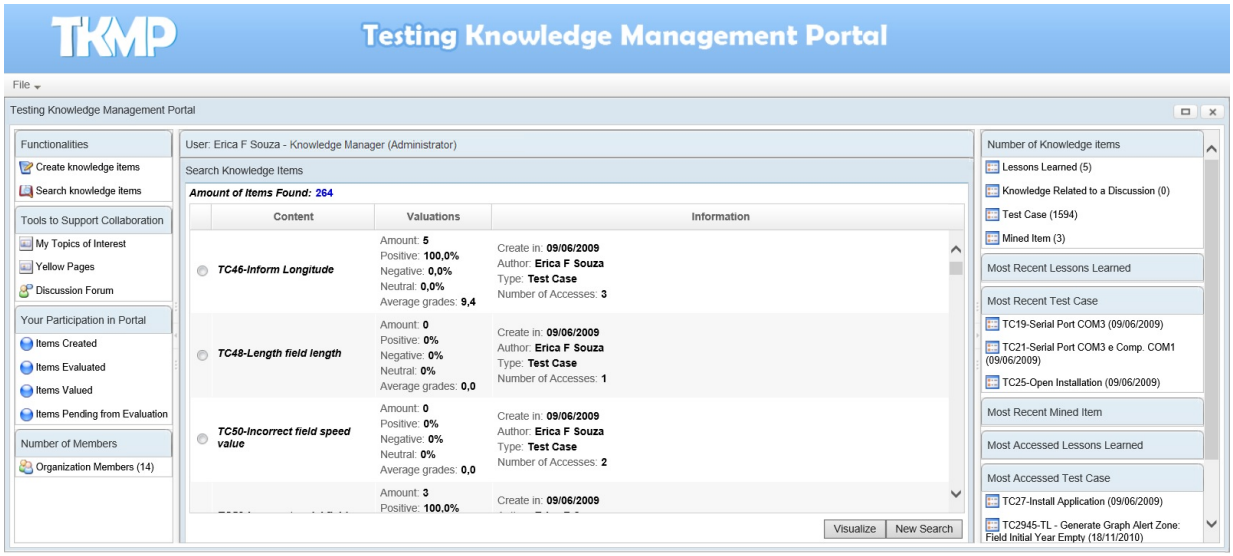

Fig. 11. Search page of TKMP

TKMP's knowledge repository was populated with 1594 test cases extracted from two actual projects: ICAMMH Project and On-Board Data Handling (OBDH) inside Inertial Systems for Aerospace Application (SIA) Project. TKMP was then initially evaluated by the testing leaders of these two projects. We should emphasize that this evaluation is preliminary, since ICAMMH was already finished, and OBDH was still in its initial phase, when we performed the evaluation step. As a result of using TKMP, both the test leaders pointed out that TKMP would be extremely useful for their projects. Both stressed the importance of such a system to support the software testing process, in particular to critical systems, such as the ones from which the test cases were extracted. Moreover, both feel comfortable with the vocabulary used by the system, which is ultimately the one provided by ROoST.

Next, we made TKMP available on the Web, and performed a survey with testing practitioners and students. 43 participants used TKMP and answered a questionnaire. The survey results pointed out that most of the participants (more than 80%) positively evaluated TKMP.

We should highlight, however, that application-based evaluation is an indirect way of evaluating an ontology. Although, TKMP was built based on ROoST, users of TKMP are not really aware of the ontology underlying the system. Moreover, TKMP does not address all the aspects covered by ROoST, but only those related to test cases. By using TKMP, in particular, those features for sharing and reusing test cases, the users are indirectly in touch with the conceptualization underlying the system. Hence, we could evaluate ROoST by means of this application only indirectly and partially.

7. Related Work

A comparison with the ontologies found in the SLR (see Section 2) was performed considering some of the characteristics of "beautiful ontologies" defined by d'Aquin and Gangemi (2011), namely: having a good domain coverage; implementing an international standard; being formally rigorous; implementing also non-taxonomic relations; being modular; being designed in a principle way; following an evaluation method; and reusing foundational ontologies. Table 5 summarizes the comparison of the 12 ontologies found in SLR with ROoST considering aforementioned criteria. For evaluating whether the ontology was developed "being formally rigorous" or not, we take into account if the ontology presents also axioms defined in some formal language.

As Table 5 shows, beside ROoST, OntoTest is the testing ontology that best fits the aforementioned criteria. In sum, the main distinguishing feature of ROoST when contrasted to other testing ontologies is that ROoST was developed taking features of "beautiful ontologies" (d'Aquin and Gangemi, 2011) into account. ROoST was developed following SABiO method, which is a well-established method, used in several ontology development efforts (Falbo, 2014). Moreover, ROoST was built by reusing and extending patterns of SP-OPL and E-OPL. Since SP-OPL and E-OPL are grounded in Unified Foundational Ontology (UFO), ROoST inherits this foundational ground from these patterns. Further, concepts introduced in ROoST were also analyzed in the light of UFO. ROoST is a heavyweight (strongly axiomatized) modular ontology that was built considering several references, including important international standards in the field of software testing. ROoST was evaluated from both verification and validation perspectives, applying different approach (assessment by humans, data-driven approach to ontology evaluation, ontology testing, and application-based approach to ontology evaluation). Finally, concerning its coverage, ROoST covers aspects related to the software testing process and its activities, artifacts that are used and produced by those activities, testing techniques for test case design, and the software testing environment, including hardware, software and human resources.

Table 5: Comparison of software testing ontologies

Ontology	Coverage	References	Engineering Method	Evaluation Method	Being Modular	Types of Relations	Foundational Ontologies	Axiomatization
ROoST	Covers aspects related to the testing process and activities, testing levels, testing artifacts, testing techniques, and testing environment	Several references, including the following international standards: ISO/IEC/IEEE 29119 (Parts 1, 2, 3 and 4) (IEEE, 2013), IEEE Std 829-2008 (IEEE, 1998) and SWEBOK (Bourque and Fairly, 2014)	SABiO	Verification, Validation & Testing activities	Organized in four sub-ontologies: Testing process and Activities, Testing Artifacts, Testing Techniques, and Testing Environment	Also non-taxonomic relations	UFO	Axioms in First Order logics
OntoTest (Barbosa et al., 2006); (Barbosa et al., 2008); (Nakagawa, 2009)	Covers aspects related to testing process and activities, testing artifacts, testing resource, and testing procedures	ISO/IEC 12207	SABiO, METHONTOLOGY	-	Organized in 6 sub-ontologies: Testing Process, Testing Phase, Testing Artifact, Testing Step, Testing Resource, and Testing Procedure sub-ontologies	Also non-taxonomic relations	-	Axioms in First Order logics
STOWS (Zhu and Zhang, 2012)	Covers aspects related to testing activities, testing artifacts, testing methods, testing type, testing environment, and capability	-	-	-	-	Also non-taxonomic relations	-	-
Taas (Yu et al. 2008, Yu et al. 2009)	Covers aspects related to testing process and activities, test type, and test environment	-	-	-	-	Almost all relations are part-whole relations	-	-
(Cai et al., 2009)	Covers software testing fundamentals, test case, test techniques, test related measures and test process	SWEBOK, ISO/IEC 9126	Uschold and King's skeletal methodology for building ontologies	-	-	Only taxonomic relations	-	-
(Sapna and Mohanty, 2011)	Devoted to Scenario-based Testing. It covers, besides scenario related concepts, aspects related to test case/test suite, test steps, test type, test resources, and test prioritization	SWEBOK	METHONTOLOGY, Ontology Development 101	-	-	Also non-taxonomic relations	-	-
(Nasser et al., 2009)	Devoted to State Machine based Testing. It covers, besides state machine related concepts, aspects related to test suite and test step	-	-	-	-	Also non-taxonomic relations	-	-

Continues

Table 5: Conclusion

Ontology	Coverage	References	Engineering Method	Evaluation Method	Being Modular	Types of Relations	Foundational Ontologies	Axiomatization
TOM (Bai et al., 2008)	Focus on Testing Artifacts and related concepts	U2TP	-	-	-	Also non-taxonomic relations	-	-
(Anandaraj et al., 2011)	Covers testing techniques, testing terminology and testing steps	-	A simple method for developing OWL ontologies	-	-	Only taxonomic relations	-	-
(Arnicans et al., 2013)	Semi-automatic generated lightweight ontologies covering the following aspects: testing, tool, software, process, analysis, capability and technique	"Standard Glossary of Terms used in Software Testing" created by ISTQB	An extension of the ONTO6 methodology for semi-automatic generation of lightweight ontologies	An expert in software testing has analyzed the ontology	-	Only taxonomic relations	-	-
MTO (Ryu et al., 2011)	Concepts directly related to MND-TMM, which covers aspects related to test process, test environment, testing techniques and test specification, among others	MND-TMM	-	-	-	Also non-taxonomic relations	-	-
(Li and Zhang, 2012)	Focus on Test Case and related concepts, including testing method, testing step and testing data	-	-	-	-	Also non-taxonomic relations	-	Axioms in First Order Logic
(Guo et al., 2011)	Focus on Test Case and its "properties", namely: id, name, precondition, input, operation (testing process), and expectation result	-	Uschold and King's skeletal methodology	-	-	Only the concept of Test Case and the aforementioned six properties	-	-

8. Conclusions

Currently, software testing is considered a complex process comprising activities, techniques, artifacts, and different types of resources (hardware, software and human resources). Thus, building a complete testing ontology is not a trivial task. Although there are a relatively large number of ontologies on software testing published in the literature (at least 12 ontologies), we notice that there are still issues related to the establishment of an explicit common conceptualization with respect to this domain. So, in this work we developed a Reference Ontology on Software Testing (ROoST), establishing a common conceptualization about the software testing domain.

ROoST is designed to be used as a reference model to be employed for several purposes, such as: to support human learning on the software testing process, as a basis for structuring and representing knowledge about software testing, for integrating software tools supporting the testing process, and for annotating testing resources in a semantic documentation approach. Furthermore, the main difference of ROoST when contrasted to other testing ontologies is that ROoST was developed taking characteristics from "beautiful ontologies" (d'Aquin and Gangemi, 2011) into account. Moreover, in order to support computational tasks such as semantic annotation and reasoning, an OWL version of ROoST was implemented.

For properly dealing with intended uses such as tool integration, KM support and semantic documentation, ROoST should be integrated to other Software Engineering (SE) domain ontologies, such as Software Requirements, Design and Coding. Thus, we have already integrated ROoST to SEON, a Software Engineering Ontology Network (Ruy et al., 2016). An ontology network is a collection of ontologies related together through a variety of relationships, such as alignment, modularization, and dependency. A networked ontology, in turn, is an ontology included in such a network, sharing concepts and relations with other ontologies (Suárez-Figueroa et al., 2012). SEON is designed seeking for: (i) taking advantage of well-founded ontologies (all its ontologies are ultimately grounded in UFO); (ii) providing ontology reusability and productivity, supported by core ontologies organized as Ontology Pattern Languages; and (iii) solving ontology integration problems by providing integration mechanisms (Ruy et al., 2016). In its current version, SEON includes a core ontology about software processes, as well as domain ontologies for the main technical software engineering subdomains, namely requirements, measurement, design, coding and testing (ROoST), and for some management subdomains, namely project management, configuration management, and quality assurance. SEON specification and its OWL version are available at https://nemo.inf.ufes.br/projects/seon/. The current OWL version of SEON is a lightweight version, in the sense that the axioms of SEON's networked ontologies (including ROoST) are not implemented in it. As a future work, we intend to implement the complete axiomatization of SEON (and ROoST), as well as to modularize the OWL code.

With respect to the use of SP-OPL (Software Process - Ontology Pattern Language), some SP-OPL patterns were not entirely used. This showed us that the patterns were too big, reflecting in fact, the need for varying patterns. As a matter of fact, the use of SP-OPL to develop ROoST leads to an important feedback to SP-OPL, which is now under review. A more specific version of an OPL about software processes for ISO standards (ISP-OPL) (Ruy et al., 2015) was developed considering several aspects identified as absent in this study (e.g., artifact types) or inadequate with respect to pattern granularity (the case of all patterns reused to develop ROoST).

As future work, we also intend to continue to explore more deeply how ontologies can be used for managing knowledge in the software testing context, and we intend to extend TKMP considering other parts of ROoST not yet explored. Moreover, we intend to perform a more consistent assessment of TKMP by using it during the accomplishment of actual software projects.

Acknowledgements

The first author acknowledges FAPESP (Process: 2010/20557-1) for the financial grant. The second author acknowledges FAPES (Process Numbers: 52272362/11 and 69382549/2014) and CNPq (Processes

485368/2013-7 and 461777/2014-2) for the financial grant. We also acknowledge: the Brazilian Aeronautics Institute of Technology (ITA); the Brazilian Agency of Research and Projects Financing (FINEP) - Project FINEP 5206/06 - ICA-MMH; and the On-Board Data Handling (OBDH) Project, inside Inertial Systems for Aerospace Application (SIA) Project for providing the data for this project research.

References

Ammann, P. and Offutt, J. (2008). Introduction to Software Testing. UK: Cambridge University Press, Cambridge.

Anandaraj, A., Kalaivani, P., Rameshkumar, V. (2011). Development of Ontology-Based Intelligent System for Software Testing. In. International Journal of Communication, Computation and Innovation, vol. 2, 157-161.

Andrade, J., Ares, J., Martínez, M., Pazos, J., Rodríguez, S., Romera, J., Suárez, S. (2013). An architectural model for software testing lesson learned systems. Information and Software Technology, vol. 55, 18-34.

Arnicans, G., Romans, D., Straujums, U. (2013). Semi-automatic Generation of a Software Testing Lightweight Ontology from a Glossary Based on the ONTO6 Methodology. In Frontiers in Artificial Intelligence and Applications, 263-276.

Arantes, L.O. R.A. and Falbo, L.O. (2010). An Infrastructure for Managing Semantic Document, Proceedings 14th International Enterprise Distributed Object Computing Conference Workshops EDOCW 2010. Los Alamitos: IEEE Computer Society, 235-244.

Bai, X., Lee, S., Tsai, W., Chen, Y. (2008). Ontology-Based Test Modeling and Partition Testing of Web Services. International Conf. on Web Services, 465-472.

Barbosa, E. F., Nakagawa, E. Y., Maldonado, J. C. (2006). Towards the establishment of an ontology of software testing. In. International Conference on Software Engineering and Knowledge Engineering (SEKE), San Francisco, CA, vol. 1, 522-525.

Barbosa, E. F., Nakagawa, E. Y., Riekstin, A. C., Maldonado, J. C. (2008). Ontology-based Development of Testing Related Tools. In International Conference on Software Engineering & Knowledge Engineering (SEKE), San Francisco, CA.

Barcelos, P. P. F., Santos, V. A., Silva, F. B., Monteiro, M. E. and Garcia, A. S. (2013). An automated transformation from OntoUML to OWL and SWRL. In: The Ontology Research Seminar in Brazil (ONTOBRAS), Belo Horizonte, MG, 130-141.

Benjamins, V. R., Fensel, D., Pérez A. G. (1998). Knowledge Management through Ontologies. The 2^{nd} International Conference on Practical Aspects of Knowledge Management (PAKM), Switzerland.

Black, R. and Mitchell, J. L. (2011). Advanced software testing: guide to the ISTQB advanced certification as an advanced technical test analyst. 3. ed. USA: Rocky Nook.

Bourque, P. and Fairley, R. E. (2014). Guide to the Software Engineering Body of Knowledge, SWEBOK, Version 3.0, IEEE Computer Society Press, Los Alamitos, CA, USA.

Braga, G., Romano, B. L., Campos, H. F., Vieira, R., Cunha, A. M. and Dias, L. A. V. (2009). Integrating amazonic heterogeneous hydrometeorological databases. Information Tecknology: New generations. Sixth International Conference on, Las Vegas, NV, 119-124.

Brank, J., Grobelnik, M. and Mladenic, D. (2005). A survey of ontology evaluation techniques. Proc. of 8^{th} Int. multi-conf. Information Society, 166-169.

Cai, L., Tong, W., Liu, Z., Zhang, J. (2009). Test Case Reuse Based on Ontology. Pacific Rim International Symposium on Dependable Computing, 103-108.

d'Aquin, M. and Gangemi, A. (2011). Is there beauty in ontologies?. Journal Applied Ontology, vol. 6, n. 3, 165-175.

Everett, G. D. and Raymond, M. J. (2007). Software testing across the entire software development life cycle. 3. ed. Canada: Published by John Wiley & Sons.

Falbo, R. A., Barcellos, M.P., Nardi, J.C., Guizzardi, G. (2013). Organizing Ontology Design Patterns as Ontology Pattern Languages. 10^{th} Extended Semantic Web Conference, Montpellier, France.

Falbo, R. A. (2014). SABiO: Systematic Approach for Building Ontologies. 1^{st} Joint Workshop Onto.Com/ODISE on Ontologies in Conceptual Modeling and Information Systems Engineering, Rio de Janeiro.

Falbo, R. A., Ruy, F. B., Guizzardi, G., Barcellos, M. P. and Almeida, J. P. A. (2014). Towards an enterprise ontology pattern language. SYMPOSIUM ON APPLIED COMPUTING, Gyeongju, Korea, 323-330.

Falbo, R. A., Menezes, C. S., Rocha, An. R. (1998). A Systematic Approach for Building Ontologies. 6^{th} Ibero-American Conference on Artificial Intelligence, Lisbon, Portugal, Lecture Notes in Computer Science, 349-360.

Guizzardi, G. (2005). Ontological foundations for structural conceptual models. The Netherlands: Universal Press, ISBN 90-75176-81-3.

Guizzardi, G. (2007). On Ontology, Ontologies, Conceptualizations, Modeling Languages and (Meta) Models. Vasilecas, O., Edler, J., Caplinskas, A. (Org.). Frontiers in Artificial Intelligence and Applications, Databases and Information Systems IV. IOS Press, Amsterdam.

Guizzardi, G. and Halpin, T. (2008). Ontological foundations for structural conceptual models. Applied Ontology. The Netherlands: Universal Press. Vol. 3, 1-2, 1-12.

Guizzardi, G., Falbo, R.A., Guizzardi, R.S.S. (2008). Grounding software domain ontologies in the Unified Foundational Ontology (UFO): the case of the ODE software process ontology. XI Iberoamerican Workshop on Requirements Engineering and Software Environments (Recife, Brazil), 244-251.

Guizzardi, G., Wagner, G., Falbo, R.A., Guizzardi, R.S.S, Almeida, J.P.A. (2013). Towards Ontological Foundations for the Conceptual Modeling of Events. 32^{th} International Conference on Conceptual Modeling, Hong-Kong, China, 327-341.

Guo, S., Zhang, J., Tong, W., Liu, Z. (2011). An Application of Ontology to Test Case Reuse. International Conference on Mechatronic Science, Electric Engineering and Computer, Jilin, China, 19-22.

Huo, Q., Zhu, H., Greenwood, S. (2003). A Multi-Agent Software Environment for Testing Web-based Applications. In 27^{th} International Computer Software and Applications Conference (COMPSAC), Dallas, TX, USA, 210-215.

Hong, Z. (2006). A Framework for Service-Oriented Testing of Web Services. Computer Software and Applications Conference. COMPSAC'06. 30^{th} Annual International, Chicago, IL, 145-150.

IEEE. (1990). The Institute of Electric and Electronic Engineers (IEEE): standard glossary of software engineering terminology. IEEE Standard 610. 12-1990, New York, NY, USA.

IEEE. (1998). The Institute of Electric and Electronic Engineers (IEEE): Standard for software test documentation. IEEE Standard 829-1998, New York, NY, USA.

IEEE. (2004). IEEE Standard for Software Verification and Validation, Std 1012, New York, NY, USA.

IEEE. (2013). The international standard for software testing. Software and systems engineering - Software Testing. ISO/IEC/IEEE 29119 Software Testing (Part 1, 2, 3 and 4).

ISO/IEC/IEEE. (2008). Standard for Systems and Software Engineering - Software Life Cycle Processes. IEEE Std 12207-2008. c1-138. 10.1109/IEEESTD.2008.4475826.

Kim, H. M. (2000). Developing Ontologies to Enable Knowledge Management: Integrating Business Process and Data Driven Approaches. AAAI Workshop on Bringing Knowledge to Business Processes.

Kitchenham, B and Charters, S. (2007). Guidelines for performing Systematic Literature Reviews in Software Engineering, School of Computer Science and Mathematics Keele University and Department of Computer Science University of Durham, UK, v. 2.3.

Li, X. and Zhang, W. (2012). Ontology-based Testing Platform for Reusing. Internet Computing for Science and Engineering (ICICSE), 86-89.

Myers, G. J. (2004). The art of software testing. 2. ed. Canada: John Wiley and Sons.

Mathur, A. P. (2012). Foundations of software testing. 5. ed. India: Dorling Kindersley (India), Pearson Education in South Asia.

Nakagawa, E.Y., Barbosa, E.F., Maldonado, J.C. (2009). Exploring ontologies to support the establishment of reference architectures: An example on software testing. Software Architecture. WICSA/ECSA, Joint Working IEEE/IFIP Conference on, Cambridge, 249-252.

Nasser, V. H., Du, W., MacIsaac, D. (2009). Knowledge-based software test generation. In. International Conference on Software Engineering and Knowledge Engineering (SEKE), 312-317.

Perry, W. E. (2006). Effective methods for software testing. 3. ed. Canada: Wiley Publishing, Inc.

Pressman, R. S. (2006). Software engineering: a practitioner's approach. 6. ed. New York: McGraw-Hill series in computer science.

Ryu, H., Ryu, D., Baik, J. (2011). A Strategic Test Process Improvement Approach Using an Ontological Description for MND-TMM. In. International Conference on Computer and Information Science, 561-566.

Ruy, F. B., Falbo, R.A., Barcellos, M.P., Guizzardi, G., Glaice K. S. (2015). An ISO-based Software Process Ontology Pattern Language and its Application for Harmonizing Standards. SIGAPP Appl. Comput. Rev. Vol. 15, 27-40.

Ruy, F. B., Falbo, R.A., Barcellos, M.P., Costa, S. D., Guizzardi, G. (2016). SEON: A Software Engineering Ontology Network. In: Proceedings of the 20th International Conference on Knowledge Engineering and Knowledge Management (EKAW), Bologna, Italy, 527-542.

Sapna, P. G. and Mohanty, H. (2011). An Ontology Based Approach for Test Scenario Management. ICISTM, vol. 141, 91-100.

Software Engineering Institute (SEI). (2010). CMMI for Development. Improving processes for developing better products and services. Version 1.3, Technical Report ESC-TR-2010-33.

Souza, E. F., Falbo, R. A., Vijaykumar, N. L. (2013). Ontology in software testing: a systematic literature review. In: The Ontology Research Seminar in Brazil (ONTOBRAS), Belo Horizonte, MG. Belo Horizonte: CEUR Workshop. vol. 1041, 71-82.

Souza, E. F., Falbo, R. A. Vijaykumar, N. L. (2015a). Knowledge management initiatives in software testing: A mapping study. Information and Software Technology. vol. 57, 378-391

Souza, E. F., Falbo, R. A., Vijaykumar, N. L. (2015b). Using the findings of a Mapping Study to conduct a Research Project: A Case in Knowledge Management in Software Testing. In: 41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Funchal, Madeira, 208-215.

Staab, S., Studer, R., Schurr, H. P., Sure, Y. (2001). Knowledge Processes and Ontologies. IEEE Intelligent Systems, vol. 16, No. 1, 26-34.

Suárez-Figueroa, M.C., Góez-Pérez, A., Motta, E., Gangemi, A. (2012). Ontology Engineering in a Networked World. Springer Science & Business Media.

TMMi Fundation (2012). Test Maturity Model integration (TMMi). Release 1.0, Ireland.

Vrandecic, D. and Gangemi, A. (2006). Unit Tests for Ontologies. In: On the Move to Meaningful Internet Systems 2006: OTM Workshops, vol. 4278, 1012-1020.

Uren, V., Cimiano,P., Iria, J., Handschuh, S., Vargas-Vera, M., Motta, E., Ciravegna, F. (2006). Semantic Annotation for Knowledge Management: Requirements and a survey of the state of the art, Journal of Web Semantics: Science, Services and Agents on the World Wide Web, vol. 4, 14-28.

Uschold, M., King, M. (1995). Towards a Methodology for Building Ontologies. Workshop on Basic Ontological Issues in Knowledge Sharing (IJCAI), 4-13.

Yufeng, Z. and Hong, Z. (2008). Ontology for Service Oriented Testing of Web Services. Symposium on Service-Oriented System Engineering, Jhongli, 129-134.

Yu, L., Su, S., Zhao, J. (2008). Performing Unit Testing Based on Testing as a Service (TaaS) Approach. In. International Conference on Service Science, 127-131.

Yu, L., Zhang, L., Xiang, H., Su, Y., Zhao, W., Zhu, J. (2009). A Framework of Testing as a Service. International Conf. on Management and Service Science, 1-4.

Zhu, H. and Huo, Q. (2005). Developing A Software Testing Ontology in UML for a Software Growth Environment of Web-Based Applications. Software Evolution with UML and XML, IDEA Group, 263-295.

Zhu, H. and Zhang, Y. (2012). Collaborative Testing of Web Services. In. IEEE Transactions on Service Computing, vol. 5, 116-130.