2013 17th IEEE International Enterprise Distributed Object Computing Conference Workshops

Using Ontology Patterns for Building a Reference Sofware Testing Ontology

E. F. Souza
Applied Computing
National Institute for Space Research, INPE
São José dos Campos/SP, Brazil
erica.souza@lac.inpe.br

R. A. Falbo
Department of Computer Science,
Federal University of Espírito Santo, UFES
Vitória/ES, Brazil
falbo@inf.ufes.br

N. L. Vijaykumar
Lab. of Comp. and Applied Math.
National Institute for Space Research, INPE
São José dos Campos/SP, Brazil
vijay@lac.inpe.br

***Abstract*— Software testing is a critical process for achieving product quality. Its importance is more and more recognized, and there is a growing concern in improving the accomplishment of this process. In this context, Knowledge Management emerges as an important supporting tool. However, managing relevant knowledge to reuse is difficult and it requires some means to represent and to associate semantics to a large volume of test information. In order to address this problem, we have developed a Reference Ontology on Software Testing (ROoST). ROoST is built reusing ontology patterns from the Software Process Ontology Pattern Language (SP-OPL). In this paper, we discuss how ROoST was developed, and present a fragment of ROoST that concerns with software testing process, its activities, artifacts, and procedures.**

Keywords— Software Testing; Ontology; Ontology Design Patterns; Ontology Pattern Language.

I. INTRODUCTION

Software testing consists of dynamic verification and validation of the behavior of a program on a finite set of test cases, against the expected behavior [1]. Testing activities are performed during the entire software development and maintenance processes, and therefore, software testing is a process embedded in the software development process [1, 2].

Software testing is a critical process for achieving quality in software products. Nowadays, its importance is widely recognized, and there is a growing concern in how to improve the accomplishment of this process. In this context, Knowledge Management (KM) emerges as an important means to manage software testing knowledge, and to improve the software testing process. There are many benefits of implementing KM in the software testing domain, such as [3]: (i) selection and application of better suited techniques; (ii) cost reduction; (iii) test effectiveness increase; and (iv) competitive advantages. However, there are also problems, such as [3]: (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.

With respect to KM systems, one of the main problems is how to represent knowledge. A KM system should support the integration of information from disparate sources, wherein a decision maker manipulates information that someone 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 [4]. As pointed out by Staab et al. [5], ontologies are particularly important for KM. They bind KM activities together, allowing a content-oriented view of KM. Ontologies define shared vocabulary to be used in the KM systems to facilitate communication, integration, search, storage and knowledge representation [6].

In order to develop a system for managing 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 [7]. A reference ontology on the software testing domain can be used for several KM-related purposes, such as for structuring knowledge repositories, for annotating knowledge items, and for making searching easier.

By means of a Systematic Literature Review (SLR), we looked for software testing ontologies. As a result, we found 12 ontologies. However, analyzing them, we concluded that they were inappropriate for our purposes, and thus we decided to build another one, which we called ROoST (Reference Ontology on Software Testing). ROoST has been developed by means of reusing and extending patterns of the Software Process Ontology Pattern Language (SP-OPL) [8].An ontology pattern language is a network of interrelated domain-related ontology patterns that provides holistic support for solving ontology development problems in a specific domain. SP-OPL is an OPL for the software process domain.

In this paper, we partially present the first version of ROoST, focusing on the software testing process, its activities, artifacts that are used and produced by those activities, and testing techniques for test case design. Moreover, we discuss how we used SP-OPL patterns to develop ROoST.

This paper is organized as follows. In Section II, we present the main concepts related to software testing, and briefly analyze the ontologies we found when we performed the SLR, discussing why we decided not to use them. Section III gives an overview of the Ontology Engineering approach we followed to develop ROoST. Section IV presents ROoST, and discusses how SP-OPL patterns were used to develop it. Section V discusses related works, comparing ROoST to some

978-0-7695-5085-5/13 $26.00 © 2013 IEEE
DOI 10.1109/EDOCW.2013.10

of the ontologies presented in Section II. Finally, in Section VI, we present our final considerations.

II. SOFTWARE TESTING AND ONTOLOGIES

Testing concepts, techniques, levels, artifacts and resources are integrated into the testing process [9]. The main activities of this process are: Test Planning, Test Case Design, Test Coding, Test Execution and Test Result Analysis [1]. Similarly to other aspects of a project, first, testing must be planned. Key aspects of planning include defining the test environment for the project, planning and scheduling testing activities, and planning for possible undesirable outcomes [1, 10]. Test planning results are documented in a Test Plan. Test case design aims at designing the test cases to be run. Test cases are documented, and then implemented. During test execution, test code is run, producing actual results, which are then analyzed to determine whether tests have been successful or not.

Testing activities are performed at different levels. Unit testing focuses on testing the units or individual components that have been developed. Integration testing takes place when such units are integrated. Finally, system testing occurs when the entire system is tested [11].

Test case design techniques (or simply testing techniques) provide systematic guidelines for designing test cases. The principle underlying them is to be as systematic as possible in identifying a representative set of program behaviors [1]. Testing techniques can be classified, among others, as [1, 9, 10, 11]: White-box Testing Techniques, which are based on information about how the software has been designed and coded; 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; Defect-based Testing Techniques, which aim at revealing categories of likely or predefined faults; Model-based Testing Techniques, which are based on models, such as Statecharts, Finite State Machines and others.

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

Aiming at building a KM system for supporting testing processes, we performed a Systematic Literature Review (SLR), looking for software testing ontologies. A SLR is a secondary study that uses a well-defined method to identify, analyze and interpret the available evidence obtained from primary studies, in a way that is unbiased and (to a degree) repeatable. The goal is to integrate and synthesize evidence related to some research questions [12]. As a result of the SLR, we found the following testing ontologies: STOWS (Software Testing Ontology for Web Service) [13, 14], OntoTest [15, 16], TaaS Ontology [17, 18], and the ontologies proposed in [19], [20], [21], [22], [23], [24], [25], [26] and [27]. Although there are a large number of ontologies on software testing, we notice that there are still problems related to the establishment of an explicit common conceptualization regarding this domain. In being applied to KM, a software testing ontology must take several characteristics into account.

In an experiment trying mainly to identify good practices in ontology design, D'Aquin and Gangemi [28] have identified some characteristics of quality ontologies. These characteristics were grouped in three dimensions: (i) formal structure, (ii) conceptual coverage and task, and (iii) pragmatic or social sustainability. In order to evaluate the existing testing ontologies for choosing one that is adequate for our purpose (KM), we focus on the first dimension, and in part on the second one, namely conceptual coverage. The characteristics included in these dimensions are [28]:

- **Structure:** Reuses foundational ontologies; designed in a principled way; formally rigorous; also implements non-taxonomic relations; strictly follows an evaluation methodology; is modular, or embedded in a modular framework.
- **Conceptual coverage:** Provides important reusable distinctions; has a good domain coverage; implements an international standard; provides an organization to unstructured or poorly structured domains.

Unfortunately, some of these characteristics are difficult to evaluate, since there isn't much information about them in the papers presenting the corresponding ontologies. Thus, in our evaluation, we focused on the most easily discernible features, namely: 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.

Regarding the first characteristic (having a good domain coverage), we notice that most ontologies have very limited coverage. The ontologies presented in [20] and [25] address only the concept of test case. The ontology presented in [22] models only testing artifacts and relationships between them. The ontology presented in [21] addresses only state machine based testing. The ontology presented in [24] focuses on scenario-based testing. The ontologies that have higher coverage are: STOWS, OntoTest, and TaaS.

Some of the ontologies take international standards into account. OntoTest is based on 1st edition of ISO/IEC 12207; the ontology presented in [19] was created based on the "Standard glossary of terms used in Software Testing" of the ISTQB; the ontology presented in [22] is based on the Unified Modeling Language (UML) 2.0 Test Profile (U2TP); and the ontologies presented in [24] and [26] are based on the SWEBOK [1]. On the other hand, there are ontologies that do not consider international standards (or at least do 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 our purposes. We are looking for a reference ontology similar to the one defined by Guizzardi [7]. This

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, we can notice that most of the existing ontologies present problems.

The ontology proposed in [23] is, in fact, an OWL implementation of a specific testing maturity model developed by the authors (the Ministry of National Defense-Testing Maturity Model (MND-TMM)), and thus it does not qualify as a reference ontology. This is also the case of the ontologies presented in [20] and [27], which are just OWL artifacts.

The ontologies presented in [19] and [26] are, in fact, taxonomies, and thus, in our view, they do not qualify as ontologies (or at most, they are lightweight ontologies). STOWS [13, 14] 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 [29]. So what is a Capability? An object or an event? This shows that this ontology presents problems.

TaaS Ontology presents very simple models. UML class diagrams presented in [17] and [18] do not specify multiplicities of the relationships. Moreover, like STOWS, most relationships are modeled as aggregation relations in UML, what 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*. Analogously to the analysis on STOWS, the composite object *Test Task* aggregates endurants and perdurants.

OntoTest [15, 16] is a modular ontology, built in layers, and represented in UML, and also implemented in OWL. A few axioms are also defined in first order logic. OntoTest is composed by a "Main Software Testing Ontology", and six sub-ontologies [15]: Testing Process, Testing Phase, Testing Artifact, Testing Step, Testing Resource, and Testing Procedure sub-ontologies. The Main Software Testing Ontology is presented in [15, 16]. It is a simple model that includes six concepts. According to this model, a *Testing Process* is composed by *Testing Steps*, and it has also *Testing Phases*. A *Testing Step* requires *Testing Resources*, adopts *Testing Procedures*, consumes and generates *Testing Artifacts*, and depends on other *Testing Steps*. *Testing Artifacts* can depend on other *Testing Artifacts*, and can be composed by other *Testing Artifacts*. Finally, a *Testing Procedure* can be supported by *Testing Resources*, and is adequate to *Testing Process*. OntoTest's Testing Step sub-ontology introduces the concept of *Testing Activity*, indicating that a *Testing Step* is composed by *Testing Activities*, while *Testing Activities* are not further decomposed, in a clear reference to ISO/IEC 12207 standard, which is organized in three levels of events: process, activity and task. The remainder of this sub-ontology consists of two large taxonomies: taxonomies of *Testing Step* and *Testing Activity*. The Testing Resource sub-ontology has a taxonomy of types of resources. We did not find papers presenting the OntoTest's Testing Process, Testing Phase, Testing Artifact, and Testing Procedure sub-ontologies. So, we think OntoTest is a work in progress.

Although probably the most complete ontology among the ones we analyzed through the SLR, OntoTest is not enough for our purposes. First, we need the missing sub-ontologies. Second, OntoTest does not properly link the concepts in the sub-ontologies. Albeit in the Main Software Testing Ontology there is a relationship between Testing Step and Test Resource, there aren't 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 works we investigated in the SLR discusses how the ontologies they propose were evaluated.

Finally, concerning the reuse of foundational ontologies, none of the ontologies analyzed in our SLR have used one. In our view, this is a problem. Foundational ontologies are domain-independent commonsense theories constructed by aggregating suitable contributions from areas such as descriptive metaphysics, philosophical logics, cognitive science and linguistics. A foundational ontology can help in making explicit the underlying ontological commitments of domain ontologies [7]. In the brief analysis we did (as in the aforementioned cases of STOWS and TaaS Ontology), we noticed that important distinctions made in Formal Ontologies were disregarded in most cases. The lack of truly ontological foundations made us to question the appropriateness of those ontologies for our purposes.

Thus, as the main result of our SRL, we concluded that the software testing community has still a lot to advance towards a reference software testing ontology, and motivated us to build ROoST. It is worthwhile to say that, initially, we considered evolving OntoTest instead of building a new ontology from scratch, since it was the most complete ontology among those analyzed. Analyzing OntoTest, we concluded that it was based on the Software Process Ontology (SPO) proposed in [30], and then we looked at this ontology. More recently, SPO was partially reengineered in the light of the Unified Foundational Ontology (UFO) [31], and now it is described as an Ontology Pattern Language (SP-OPL) [8]. Since there are important differences between the SPO version that inspired OntoTest,

and the most recent one, we abandoned the idea of evolving OntoTest, and started a new effort for developing ROoST.

III. Ontology Engineering Approach

In order to develop ROoST, we adopted SABiO (Systematic Approach for Building Ontologies) [32]. SABiO prescribes an iterative process comprising the following activities: purpose identification and requirement specification, ontology capture, ontology formalization, reuse of existing ontologies, ontology evaluation, and ontology documentation.

With respect to ontology reuse, we reused patterns from SP-OPL [8]. SP-OPL is a core ontology on software processes. As a core ontology, SP-OPL provides a precise definition of the structural knowledge in the field of software processes that spans across different application domains in this field [8]. Moreover, SP-OPL is grounded on the Unified Foundational Ontology (UFO) [29, 33].

As an ontology pattern language, SP-OPL contains a set of interrelated ontology patterns related to the software process domain, plus a map providing explicit guidance on what problems can arise in this universe of discourse, informing the order to address these problems, and suggesting one or more patterns to solve each specific problem [8].

SP-OPL organizes 30 patterns and has three entry points, i.e. three different ways to enter in the pattern language. The choice of the entry point from which to enter in the SP-OPL depends on the focus of the ontology engineer. When the requirements for the domain ontology being developed include problems related to definition of standard software processes, the entry point is the Standard Process Structure (SPS) pattern, from which other patterns related to the definition of standard software processes can be achieved. The second entry point is the Software Process Planning (SPP) pattern, which deals with how a software process is planned in terms of sub-processes and activities. From this pattern, other patterns related to the definition of a software process for a project and scheduling it can be achieved. Finally, the third entry point in SP-OPL is the Process and Activity Execution (PAE) pattern, which deals with representing process and activity occurrences. From this pattern, the ontology engineer can achieve others that address problems related to resource participation, procedures adopted, and work product inputs and outputs [8].

For developing ROoST, we chose the third entry point (EP3), since we are interested in representing knowledge involved in the execution of testing processes. Fig. 1 shows the SP-OPL patterns accessible from this entry point. Due to space limitations, in this paper we do not discuss the fragment of ROoST regarding the test environment, which were developed using the patterns RPA and HRPA, shown detached in grey.

The Process and Activity Execution (PAE) pattern represents the occurrences of processes and activities in the context of a project. The Human Resource Participation (HRPA) pattern represents the participation of a human resource in an activity occurrence. The Resource Participation (RPA) pattern represents the participations of resources (hardware and software) in activity occurrences. The Work Product Participation (WPPA) pattern represents the participations of artifacts in activity occurrences. Finally, the Procedure Participation (PRPA) pattern represents the participation (adoption) of procedures in activity occurrences.

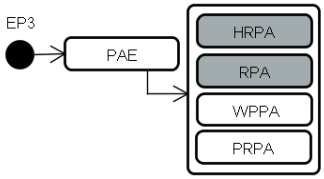

Fig. 1. SP-OPL patterns accessible from the entry point EP3.

Besides the 30 patterns described in the SP-OPL map, SP-OPL has two supplementary patterns: Work Product Taxonomy (WPT), which describes types of artifacts, and Procedure Taxonomy (PRT), which describes types of procedures. We also reused these two patterns. Figures 2 and 3 present the conceptual models of the PAE and WPPA patterns. PRPA, PRT and WPT patterns, although used in this paper, are not presented, due to space limitations.

Patterns in SP-OPL are described using a form with several fields, including: name, intent, competency questions that the pattern aims to answer, conceptual model, axiomatization, and foundations (ontological analysis taking ontological distinctions of UFO into account). The conceptual models of the SP-OPL patterns are encoded in OntoUML [33], a UML profile that enables modelers to make finer-grained modeling distinctions between different types of classes and relations according to some ontological distinctions put forth by UFO.

Fig. 2 shows the conceptual model of PAE pattern [8]. The intent of this pattern is to represent the occurrences of processes and activities in the context of a project, and their mereological structure. The following competency questions are addressed by this pattern: (PAE-CQ1) What is the project in which context a given process/activity occurrence occurred? (PAE-CQ2) How is a process occurrence structured in terms of sub-processes and activities? (PAE-CQ3) When did a process occurrence start and when did it end? (PAE-CQ4) When did an activity occurrence start and when did it end? (PAE-CQ5) From which activity occurrences does an activity occurrence depend on?

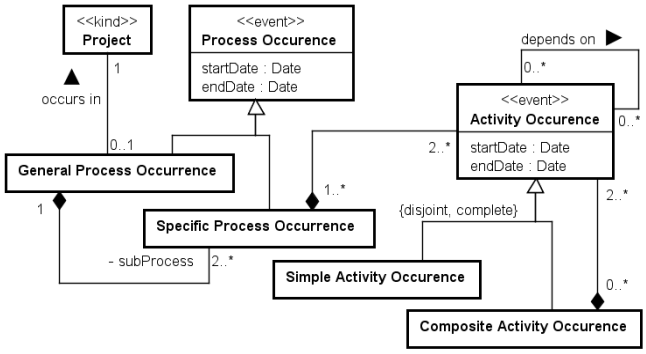

Fig. 2. The Process and Activity Execution (PAE) ontology pattern.

Process Occurrences and *Activity Occurrences* are complex events, and the whole-part relations between events are strict partial order. In the software process domain, there are two main kinds of *Process Occurrences*: *General Process*

Occurrence and *Specific Process Occurrence*. A general process occurrence is the whole execution of the process defined for a *Project*. It is composed by specific process occurrences, allowing an organization to decompose a general process into sub-processes. A specific process occurrence, in turn, is decomposed into *Activity Occurrences*. Activity occurrences can be simple or composite. A composite activity occurrence is a complex event that is composed by other activity occurrences. A simple activity occurrence is not composed by other activity occurrences, but it is still a complex event in UFO [29], since it is composed by other events representing the participations of human resources, hardware and software resources, artifacts, and procedures in the activity occurrence [8].

Fig. 3 presents the conceptual model of the WPPA pattern [8]. The intent of this pattern is to represent the participation of artifacts (as input or output) in an activity occurrence. The following competency questions are addressed by this pattern: (WPPA-CQ1) Which artifacts are produced in (are an output of) an activity occurrence? (WPPA-CQ2) Which artifacts are used in (are an input to) an activity occurrence? (WPPA-CQ3) When did an artifact participation in an activity occurrence start and when did it end?

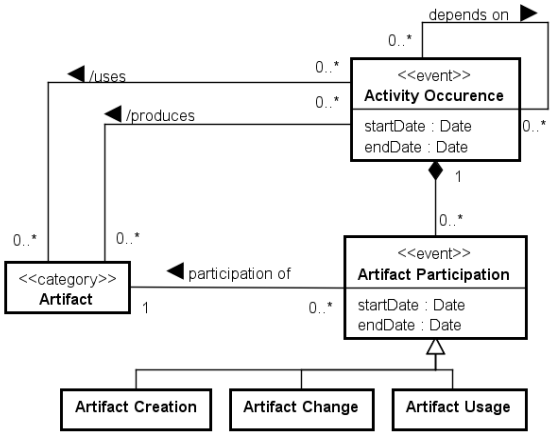

Fig. 3. The Work Product Participation (WPPA) ontology pattern.

An *Activity Occurrence* can have as its parts *Artifact Participations*, which are also events. An *Artifact Participation* is the participation of a single *Artifact* in an *Activity Occurrence*. 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 occurrence; (ii) *Artifact Usage*, meaning that the artifact is only used during the activity occurrence, and thus it is only an input for the activity occurrence; and (iii) *Artifact Change*, meaning that the artifact is changed during the activity occurrence, and thus it is both input and output for the activity occurrence [8]. It is worthwhile to point out that both *produces* and *uses* relations between *Activity Occurrence* and *Artifact* are derived from the participations of the artifacts in the activity occurrence. Thus they are represented preceded by a bar (/).

Although the main concepts in ROoST have a counterpart in SP-OPL, when extending SP-OPL conceptualization for the testing domain, we had also to introduce new concepts that are not described in SP-OPL. In those cases, in order to maintain the alignment with UFO, we also analyzed the concepts introduced in ROoST in the light of UFO.

IV. ROOST: A SOFTWARE TESTING ONTOLOGY

The purpose of ROoST is to define a shared vocabulary regarding the testing domain to be used in knowledge management initiatives to facilitate communication, integration, search, and representation of test knowledge. In order to achieve this purpose, ROoST should be able to answer the following competency questions:

CQ1. What is the project in which a given testing process/activity occurrence occurred?

CQ2. How is a testing process structured in terms of testing activities and sub-activities?

CQ3. When did a testing process start and when did it end?

CQ4. When did a testing activity start and when did it end?

CQ5. From which activities does a testing activity depend on to be performed?

CQ6. What are the test levels typically considered in testing?

CQ7. What are the artifacts produced in a testing activity?

CQ8. What are the artifacts used by a testing activity?

CQ9. 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 a testing technique can be applied?

CQ12. Which are the testing techniques applied to derive a given test case?

Once defined the competency questions, we looked for the patterns in SP-OPL to be reused. It is important to emphasize that SP-OPL drove the rewriting of the competency questions originally defined for ROoST. Moreover, some competency questions were not initially identified, and they were taken directly from SP-OPL. In both cases, we had to specialize them to the software testing domain. 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 CQ9 and CQ12.

ROoST is developed in a modular way. Currently, ROoST has four modules (sub-ontologies). In this paper we present three of them. Due to space limitations, we do not present the Testing Environment sub-ontology. In the following subsections we present the other ROoST sub-ontologies and discuss how we applied the reused patterns in their development. The conceptual models presented in this section are encoded in OntoUML [33]. Concepts reused from SP-OPL are shown in grey, and they are preceded by the pattern acronym (e.g., PAE::).

A. Testing Process and Activities sub-ontology

This sub-ontology addresses the competency questions CQ1 to CQ6. To answer them, we reused PAE pattern. CQ1 to CQ5 have a direct correspondence to PAE-CQ1 to PAE-CQ5, and thus we specialized PAE concepts to the testing domain, as shown in Fig. 4. *Testing Process Occurrence* is a subtype of

Specific Process Occurrence, since a testing process occurs in the context of the entire software process (*General Process Occurrence*) of a *Project*. A testing process, in turn, is composed by testing activities, and thus *Testing Activity Occurrence* is considered a subtype of *Activity Occurrence*. As well as *Activity Occurrence*, *Testing Activity Occurrence* can be further divided into *Composite* and *Simple Testing Activity Occurrences*.

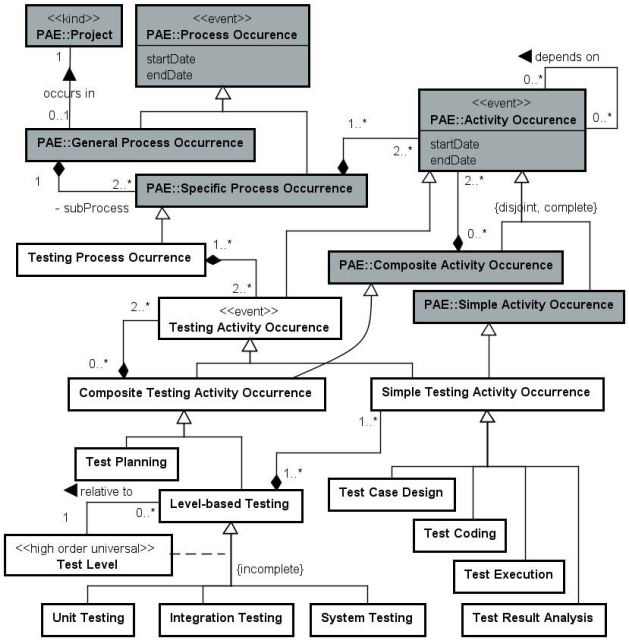

Fig. 4. ROoST's Testing Process and Activities sub-ontology.

Besides specializing concepts, we have also specialized relationships from PAE. For instance, in PAE, there is a whole-part relationship between *Specific Process Occurrence* and *Activity Occurrence*. The whole-part relationship between *Testing Process Occurrence* and *Testing Activity Occurrence* is a subtype of the former. In this paper, whenever a ROoST relationship is a subtype of another relationship defined in SP-OPL, we use the same name for both.

Looking at the literature [1, 9, 10], it is possible to say 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, we considered that these are subtypes of *Testing Activity Occurrence*. Moreover, we considered that *Test Planning* is a *Composite Testing Activity Occurrence*. Although not shown in Fig. 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, we considered *Test Case Design*, *Test Coding*, *Test Execution* and *Test Result Analysis* as *Simple Testing Activity Occurrences*.

Software testing is usually carried out at different test levels [11]. *Simple Testing Activity Occurrences* are grouped according to the *Test Level* to which they are related, forming *Level-based Testing* activity occurrence (CQ6). Thus, *Level-based Testing* is a subtype of *Composite Testing Activity Occurrence*. In Fig. 4, we made explicit the three most cited testing levels in the literature: *Unit Testing*, *Integration Testing* and *System Testing*. However, there may be other, such as *Regression Testing*.

To answer CQ1, we also needed two axioms defined in PAE that says that the relationship *occurs in* between *General Process Occurrence* and *Project* can be extended to the sub-processes and activity occurrences that compose the former.

$$\forall gpo: GeneralProcessOccurrence;\ p: Project,\ spo: SpecificProcessOccurrence\ occursIn(gpo,p) \wedge partOf(spo,gpo) \rightarrow occursIn(spo,p) \quad (A1)$$

$$\forall spo: SpecificProcessOccurrence;\ p: Project,\ ao: ActivityOccurrence\ occursIn(spo,p) \wedge partOf(ao,spo) \rightarrow occursIn(ao,p) \quad (A2)$$

B. Testing Artifacts sub-ontology

The Testing Artifacts sub-ontology addresses the competency questions CQ7 to CQ9. To answer CQ7 and CQ8, we reused the WPPA pattern. CQ7 and CQ8 are related to WPPA-CQ1 and WPPA-CQ2, respectively. On the other hand, 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 occurrence, WPPA-CQ3 is not relevant for ROoST. As a consequence, we modeled only the derived relationships */uses* and */produces*, instead of modeling the artifact participations.

An important issue for ROoST is to describe the types of artifacts that are produced and used during the testing process. Thus, we reused the WPT pattern too. In WPT, an incomplete taxonomy of software artifacts is defined including, among others, the following kinds of artifacts: *Document*, which refers to artifacts consisting of textual statements usually associated with organizational patterns that define how they should be produced; *Code*, which concerns to portions of code written in a programming language; and *Data*, referring to data used or produced during the software process.

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, we specialized the relationships *uses* and *produces* from WPPA to link testing artifacts to the corresponding testing activities in which they were produced or used. Moreover, we defined relationships between the testing artifacts, as shown in Fig. 5.

During *Test Planning*, a *Test Plan* is produced. In *Test Case Design*, different artifacts are used for deriving test cases, such as requirements specifications, use cases, 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 are *Test Cases*. A *Test Case* aims to test a *Code To Be Tested*, and specifies the *Test Case Input* and the *Expected Result*.

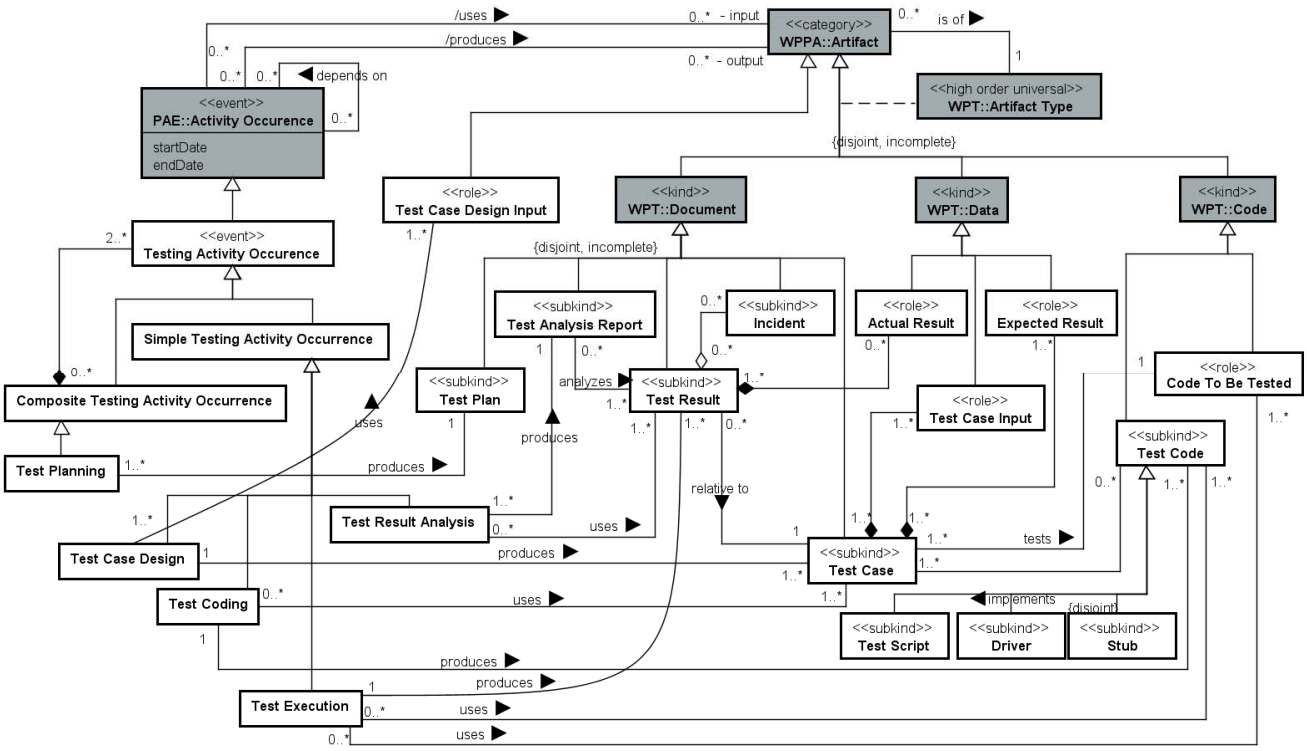

Fig. 5. The ROoST's Testing Artifacts sub-ontology.

Test Case Input and *Expected Result* are roles played by *Data* 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 worthwhile to highlight that "role" in this paper 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 [33]. Take as an example the role *Code To Be Tested*. It is an anti-rigid specialization of *Code* (a *kind* 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* activity occurrence, *Test Cases* are used to derive *Test Code* that implements them. *Test Code* is a portion of code that is to be run for executing a given set of test cases. There are three main subtypes of *Test Code*: *Test Script*, *Driver* and *Stub*.

Test Execution requires as input the *Test Code* to be run and the *Code To Be Tested*. 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*. *Actual Result* is the role played by *Data* when it is part of a *Test Result*.

A test execution can run and achieve a result (*Actual Result*), but it can also fail, generating 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 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. Thus, a *Test Result* contains either an *Actual Result*, or an *Incident*, or both. Moreover, a *Test Result* must include one of them, as defined by the following axiom:

$$\forall tr\ TestResult(tr) \rightarrow \exists art\ (Actual\ Result(art) \\ Incident(art)) \wedge part\ of(art, tr) \qquad (A3)$$

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 occurrence a_1, and *art* is also an input to another activity occurrence a_2, then a_2 depends on a_1.

$$\forall a_1, a_2{:}ActivityOccurrence, art{:}Artifact\ (produces(a_1, art) \wedge \\ uses(a_2, art) \rightarrow dependsOn(a_2, a_1)) \qquad (A4)$$

From this axiom, we can 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 (A5). Thus, we can say, for instance, that *Test Execution* also depends on *Test Case Design*.

C. Testing Techniques sub-ontology

This sub-ontology addresses the competency questions CQ10 to CQ12. In order to answer them, we reused the PRPA and PRT patterns. According to the PRT pattern, *Procedures* are classified into: *Guideline, Method* and *Technique*. According to the PRPA pattern, *Procedures* can be adopted to support the accomplishment of *Activity Occurrences*. Analogously to WPPA, PRPA includes a concept for the events representing procedure participations in activity occurrences. However, since in ROoST we are not interested in modeling those events, but only which procedures were adopted by a testing activity occurrence, we worked only with the derived relationship */adopts*, as shown in Fig. 6.

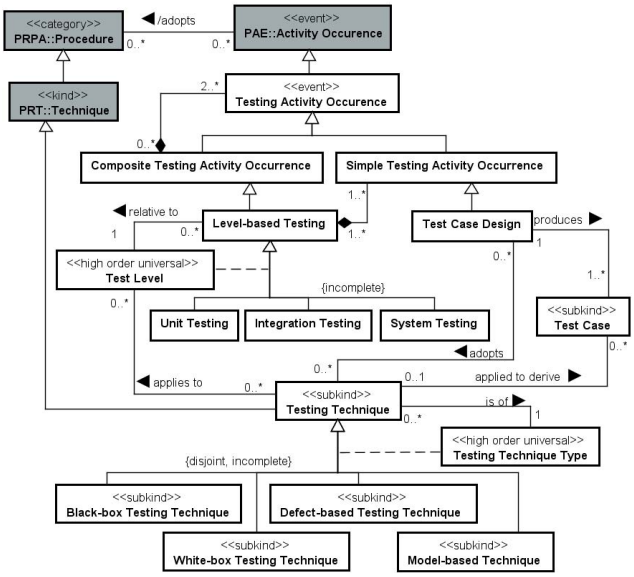

Fig. 6. ROoST's Testing Techniques sub-ontology.

To answer CQ10, we needed only one kind of procedure (*Technique*), and thus this concept was specialized 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, we introduced a relationship between *Testing Technique* and *Test Level*. *Black-box Testing Techniques*, for example, applies to all test levels. *White-box Testing Techniques*, on the other hand, are suitable only for *Unit Testing* and *Integration Testing*. They are not suitable for *System Testing*, because it is difficult in practice to derive tests cases based on the source code when the entire system is considered [11]. *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*.

Finally, for designing a specific test case, a testing technique may be applied. Thus, for answering CQ12, we introduced a relationship between *Testing Technique* and *Test Case*, in order to link a test case to the testing technique applied in its design.

D. ROoST Evaluation

In order to evaluate ROoST, we performed ontology verification & validation activities. ROoST evaluation started with a verification activity, where we checked if the concepts, relations and axioms defined in ROoST are able to answer its competency questions. Table 1 shows part of the verification, showing which elements of the ontology (concepts, relations, properties and axioms) answer CQ1, CQ4, CQ7 and CQ9. We should highlight that there are other axioms from the SP-OPL, and that also help to answer ROoST competency questions, but due to space limitations they were not presented in this paper, and thus are not listed in Table 1.

TABLE 1. ROOST VERIFICATION

Competency Question	Concepts, *Relations* and *Properties*	Axioms
CQ1	Testing Activity Occurrence *part of* Testing Process Occurrence	A1, A2
	Testing Activity Occurrence *subtype of* Activity Occurrence	
	Testing Process Occurrence *subtype of* Specific Process Occurrence	
	Activity Occurrence *part of* Specific Process Occurrence	
	Specific Process Occurrence *part of* Gerneral Process Occurrence	
	Gerneral Process Occurrence *occurs in* Project	
CQ4	Testing Activity Occurrence *subtype of* Activity Occurrence, which contains the properties *startDate* and *endDate*	
CQ7	Test Planning *produces* Test Plan	
	Test Case Design *produces* Test Case	
	Test Coding *produces* Test Code	
	Test Execution *produces* Test Result	
	Test Analysis *produces* Test Analysis Report	
CQ9	Test Case Input, Expected Result *part of* Test Case	A3
	Test Code *implements* Test Case	
	Test Result *is relative to* Test Case	
	Actual Result, Incident *part of* Test Result	
	Test Analysis Report *analyzes* Test Result	

To validate ROoST, we instantiated its concepts and relations with individuals extracted from an actual project, in order to check if the ontology was able to represent concrete situations of the real world. These individuals were extracted from the Amazon Integration and Cooperation for Modernization of Hydrological Monitoring (ICAMMH) project. Table 2 shows part of the instantiation made.

V. RELATED WORKS

There are some ontologies for the software testing domain published in the literature. As discussed in Section 2, some of them are very simple [20, 25], or address testing from a specific point of view [21, 24] or do not present conceptual models, but only OWL implementations [23, 27]. Since we have already commented about those ontologies in Section II, in this section we compare ROoST only to those with higher coverage, namely: OntoTest [15, 16], STOWS [13, 14] and TaaS Ontology [17, 18].

TABLE 2. ROOST INSTANTIATION

Concept	Instance
Project	ICAMMH Project
Black-box Testing Technique	Equivalence partitioning, Boundary-value analysis (black-box techniques applied to derive test cases in the ICAMMH Project)
White-box Testing Technique	Control-flow-based, Data-flow-based criteria (white-box techniques applied to derive test cases in the ICAMMH Project)
Test Case	Test Case *P01-256* [Collect 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*)
Expected Result	Show Message: "Invalid file" (expected result for 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	*P01-256 Script* (a test script that implements the test case *P01-256*)
Actual Result	"Invalid file"

Concerning testing process and activities, ROoST has several commonalities with OntoTest, since they share the same basis (the Software Process Ontology proposed in [30]). OntoTest, as ROoST, makes an analogy between software process and software testing, and both consider that the testing process is decomposed into testing activities (*Testing Step*, in OntoTest) that can be further decomposed. However, OntoTest commits to a more restricted conceptualization: OntoTest considers that a *Testing Process* is composed by *Testing Steps* that, in turn, can be decomposed into *Testing Activities*. *Testing Activities* are not further decomposed. ROoST, on the other hand, admits testing activity decomposition in any level, making clear distinction between *Composite Testing Activity Occurrence* and *Simple Testing Activity Occurrence*. TaaS Ontology considers that the *Test Activity* (analogous to ROoST's *Testing Process Occurrence*) is composed by *Test Case Generation*, *Test Execution*, *Test Monitor*, *Test Resutl Analysis* and *Test Visualization*. STOWS says nothing about decomposing testing activities. There is only a taxonomy of types of activities, which includes activities such as *test planning*, *test case generation*, *test execution*, among others. The OntoTest Testing Step sub-ontology also considers basically the same types of activities as STOWS. However, none of these ontologies integrates this view with the one dealing with test level. OntoTest addresses *Testing Phase* (which corresponds to *Level-based Testing* in ROoST) detached from *Testing Step*. This also occurs in STOWS, which considers test levels as types of *Contexts*, but does not make any relation between *Context* and *Activity*. TaaS ontology addresses this issue using the concept of *Test Stage*, but does not link it to its concept of *Test Activity*. In ROoST, we integrate all aspects related to testing activities in a hierarchy, allowing representing different views on them.

Concerning testing artifacts, TaaS Ontology does not address this issue. In STOWS, *Artifact* has a property *Type* that captures different types of artifacts, but there isn´t a relationship between Artifact and Activity. In OntoTest, there are two general relationships between *Testing Step* and *Testing Artifact* (*consumes* and *is generated*). Moreover, in [15] it is said that "each testing step may involve a number of different artifacts, such as test documents, test diagrams, test cases, test requirements, drivers and stubs, and artifacts under test". However, these artifacts are not modeled, and OntoTest does not make explicit in which activities they are produced or used. In ROoST, we model the main artifacts used and produced during the testing process, and also model which artifacts are used and produced in each activity. Moreover, we establish several relationships between testing artifacts.

Finally, concerning testing techniques, TaaS Ontology does not address this issue. In STOWS, there is the concept of *Method*, but again it is not linked to *Activity*. In OntoTest, there is a relationship between *Testing Step* and *Testing Procedure*, and it is said that "Testing procedures can be categorized in testing methods, testing guidance and testing techniques. […] Functional, structural, error-based and state-based are examples of testing techniques". However, as in the case of artifacts, these aspects are not modeled in OntoTest. ROoST, on the other hand, captures that there are several types of *Testing Techniques*, and that some of them applies to specific *Test Levels*. Moreover, we link *Test Case* to the *Testing Technique*, in order to capture which technique was applied to derive a given test case.

VI. CONCLUSIONS

For properly managing testing knowledge, we need a common understanding of the testing concepts, in order to associate semantics to a large volume of test information. In order to deal with this problem, we developed ROoST (Reference Ontology on Software Testing). In this paper we present a fragment of ROoST, focusing on the software testing process, activities, artifacts that are used and produced by them, and testing techniques for test case design.

The main distinguishing feature of ROoST when contrasted to other testing ontologies is that ROoST was developed taking characteristics of "beautiful ontologies" [28] into account. ROoST was developed in a principled way, following the SABiO method, which is a well-established method, used in several ontology development efforts [32]. Moreover, ROoST was built by means of reusing and extending patterns of the SP-OPL. Since SP-OPL is grounded in the Unified Foundational Ontology (UFO), ROoST inherits this foundational ground from SP-OPL. Further, concepts introduced in ROoST were also analyzed in the light of UFO. ROoST is a heavyweight modular ontology that was built considering several references, including international standards. It was evaluated from both verification and validation perspectives, to check if the ontology is able to answer the competency questions and to place the ontology in a real world situation. Finally, concerning its coverage, ROoST covers aspects related to 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. This last aspect was not discussed in this paper, due to space limitations. Although ROoST presents a good coverage (especially when compared to other testing ontologies), there are still some points to advance, including the competencies of testing workers, i.e. the sets of skills needed to perform different testing activities. This aspect is very important for Knowledge Management (KM) in software testing, and we intend to address it in a near future.

Finally, as a continuation of this work, we intend to develop a KM system to manage testing-related knowledge items. This KM system will be built using ROoST, and its usage will serve a further evaluation of ROoST.

ACKNOWLEDGMENT

The first author acknowledges FAPESP (Process: 2010/20557-1) for the financial grant. The second author acknowledges FAPES (Process Number 52272362/11) for the financial grant. We also acknowledge ITA, ANA, Brazilian Agency of Research and Projects Financing (FINEP) and the Casimiro Montenegro Filho Foundation (FCMF) for providing the data of Project FINEP 5206/06.

REFERENCES

[1] IEEE Computer Society, SWEBOK, "A Guide to the Software Engineering Body of Knowledge," 2004.

[2] P. Ammann, J. Offutt, Introduction to Software Testing. UK: Cambridge University Press, Cambridge, 2008.

[3] E. F. Souza, R. A. Falbo, and N. L. Vijaykumar, "Knowledge Management Applied to Software Testing: A Systematic Mapping," Proc. of the 25th International Conference on Software Engineering and Knowledge Engineering (SEKE 2013), Boston, USA, 2013.

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

[5] S. Staab, R. Studer, H. P. Schurr, and Y. Sure, "Knowledge Processes and Ontologies," IEEE Intelligent Systems, vol. 16, No. 1, 2001.

[6] V. R. Benjamins, D. Fensel, and A. G. Pérez, "Knowledge Management through Ontologies," The 2nd International Conference on Practical Aspects of Knowledge Management (PAKM98), Switzerland, 1998.

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

[8] R. A. Falbo, M.P. Barcellos, J.C. Nardi, and G. Guizzardi, "Organizing Ontology Design Patterns as Ontology Pattern Languages," 10th Extended Semantic Web Conference, Montpellier, France, 2013.

[9] I. Burnstein, Practical Software Testing: a process-oriented approach. 3rd ed. New York: Springer Professional Computing, 2003.

[10] Rex Black and Jamie L. Mitchell. Advanced Software Testing: guide to the ISTQB advanced certification as an advanced technical test analyst. USA:Oreilly & Assoc, 2008.

[11] A. P. Mathur, Foundations of Software Testing. 5rd ed. Delhi, India: Dorling Kindersley (India), Pearson Education in South Asia, 2012.

[12] B. Kitchenham, and S. Charters, "Guidelines for performing Systematic Literature Reviews in Software Engineering," Technical Report, Computer Science and Mathematics Keele University and Departament of Computer Science University of Durham, UK, v. 2.3, 2007.

[13] H. Zhu and Q. Huo, "Developing A Software Testing Ontology in UML for a Software Growth Environment of Web-Based Applications," Software Evolution with UML and XML, H. Yang, ed., pp. 263-295, IDEA Group, Inc., 2005.

[14] H. Zhu and Y. Zhang, "Collaborative Testing of Web Services," In IEEE Transactions on Service Computing, pp. 116 - 130, v. 5, 2012.

[15] E. F. Barbosa, E. Y. Nakagawa, and J. C. Maldonado, "Towards the establishment of an ontology of software testing," International Conference on Software Engineering and Knowledge Engineering, SEKE, 2006.

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

[17] L. Yu, S. Su, and J. Zhao, "Performing Unit Testing Based on Testing as a Service (TaaS) Approach," International Conference on Service Science, pp. 127-131, 2008.

[18] L. Yu, L. Zhang, H. Xiang, Y. Su, W. Zhao, and J. Zhu, "A Framework of Testing as a Service," Management and Service Science, International Conference on, pp. 1- 4, 2009.

[19] G. Arnicans, D. Romans, and U. Straujums, "Semi-automatic Generation of a Software Testing Lightweight Ontology from a Glossary Based on the ONTO6 Methodology," In: Frontiers in Artificial Intelligence and Applications, pp. , 263-276, v. 249, 2013.

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

[21] V. H. Nasser, W. Du, and D. MacIsaac, "Knowledge-based software test generation," International Conference on Software Engineering and Knowledge Engineering, SEKE 2009, pp. 312 – 317, 2009.

[22] X. Bai, S. Lee, W. Tsai, and Y. Chen, "Ontology-Based Test Modeling and Partition Testing of Web Services," IEEE International Conference on Web Services, pp. 465-472, 2008.

[23] H. Ryu, D. Ryu, and J. Baik, "A Strategic Test Process Improvement Approach Using an Ontological Description for MND-TMM," International Conference on Computer and Information Science, pp. 561-566, 2008.

[24] P. G. Sapna and H. Mohanty, "An Ontology Based Approach for Test Scenario Management," ICISTM 2011, pp. 91–100, 2011.

[25] X. Li, and W. Zhang. "Ontology-based Testing Platform for Reusing," International Conference on Internet Platform for Reusing, 2012.

[26] L. Cai, W. Tong, Z. Liu, and J. Zhang. "Test Case Reuse Based on Ontology," Pacific Rim International Symposium on Dependable Computing, pp. 103-108, 2009.

[27] A. Anandaraj, P. Kalaivani, and V. Rameshkumar, "Development of Ontology-Based Intelligent System For Software Testing," In International Journal of Communication, Computation and Innovation, v. 2, 2011.

[28] M. d'Aquin and A. Gangemi, "Is there beauty in ontologies? Applied Ontology," vol. 6, n.3, p. 165–175, 2011.

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

[30] R.A. Falbo and G. Bertollo, "Establishing a Common Vocabulary for Software Organizations Understand Software Processes," in EDOC International Workshop on Vocabularies, Ontologies and Rules for The Enterprise (VORTE), Enschede, The Netherlands. EDOC, Workshop on Vocabularies, Ontologies and Rules for The Enterprise, 2005.

[31] A.C.O. Bringuente, R.A. Falbo, G. Guizzardi, "Using a Foundational Ontology for Reengineering a Software Process Ontology". Journal of Information and Data Management, vol. 2, n. 3, pp. 511-526, 2011.

[32] R. A. Falbo, "Experiences in Using a Method for Building Domain Ontologies," Proc. of the 16th International Conference on Software Engineering and Knowledge Engineering, International Workshop on Ontology In Action, Banff, Canada, 2004.

[33] G. Guizzardi, "Ontological Foundations for Structural Conceptual Models," Universal Press, The Netherlands, ISBN 90-75176-81-3, 2005.