An Ontology Model to Support the Automated Evaluation of Software

Raúl García-Castro*, Miguel Esteban-Gutiérrez*, Mick Kerrigan‡ and Stephan Grimm§
*Ontology Engineering Group. Facultad de Informática, Universidad Politécnica de Madrid, Spain
‡STI Innsbruck. Universität Innsbruck, Austria
§FZI Research Center for Information Technology, Germany

***Abstract*—Even though previous research has tried to model Software Engineering knowledge, focusing either on the entire discipline or on parts of it, we lack an integrated conceptual model for representing software evaluations, and we also lack the information related to them that supports their definition and enables their automation and reproducibility.**

This paper presents an extensible ontology model for representing software evaluations and evaluation campaigns, i.e., worldwide activities where a group of tools is evaluated according to a certain evaluation specification using common test data.

During the development of the ontologies, we have reused current standards and models and have linked these ontologies with some renowned ones.

I. Introduction

The SEALS European project[1] is developing an infrastructure for the evaluation of semantic technologies, the *SEALS Platform*, that will provide independent computational and data resources for evaluating such technologies. This platform will allow users to define and execute evaluations on their own and will support the organization and execution of evaluation campaigns, i.e., worldwide activities where a group of tools is evaluated according to a certain evaluation specification with common test data.

One core component in the development of any infrastructure supporting software evaluations is the definition of the data model to be used for representing software evaluations, evaluation campaigns and the rest of the entities managed by such a platform.

Nevertheless, even if there is some progress in reaching consensus in the Software Engineering discipline and in the content of its knowledge areas – the best example is the initiative that led to the Guide to the Software Engineering Body of Knowledge (SWEBOK) [1] – we still lack formal and reusable models for representing information common in the software evaluation area.

Previous research has tried to model Software Engineering knowledge related to software evaluation, both in general (e.g., [2]) or focusing on parts of the discipline (e.g., software quality characteristics as covered in [3], [4], [5] and software measurement in [6], [7]), but we lack an integrated model for representing software evaluations, the different information related to them, and software evaluation campaigns.

The main contribution of this paper is an explicit conceptual model for representing software evaluations and evaluation campaigns. Our design principles when defining this model were that it is

- machine-processable to support the automation of the evaluation process,
- exhaustive to allow evaluations to be reproducible,
- interoperable to allow interchanging evaluation-related information between different systems, and
- extensible because it will have to be expanded to be used in concrete evaluations and evaluation campaigns.

To cover these requirements we decided to use ontologies for representing such a model. Ontologies are formal and explicit specifications of a conceptualization [8] that allow representing consensual knowledge, are easily extensible, and support interoperability at the knowledge level.

Furthermore, we encourage the use of the ontologies presented in this paper to represent software evaluation information. To facilitate this, we have reused current standards and models and linked our ontologies to some renowned ones.

This paper is structured as follows. Sections II and III present the main entities related to a software evaluation activity and the life cycle of these entities, respectively. Section IV introduces the upper ontology used to represent common entities and their properties and Sections V, VI and VII discuss the ontologies used to represent software evaluations, evaluation execution requests and evaluation campaigns, respectively. Section VIII refers to previous work related to the one presented in this paper and, finally, Section IX draws conclusions from this work and proposes future lines of research.

II. Software Evaluation Entities

Our model revolves around the notion of *evaluation*, which is largely inspired by the notion of evaluation module as defined by the ISO/IEC 14598 standard on software product evaluation [9]. However, it is not our intention to fully cover this standard but to focus on the entities required to describe software evaluations for their automated execution.

As illustrated in Figure 1, in any *evaluation* a given set of *tools* are exercised, following the workflow defined by a given *evaluation description* and using determined *test data*. As an outcome of this process, a set of *evaluation results* is produced.

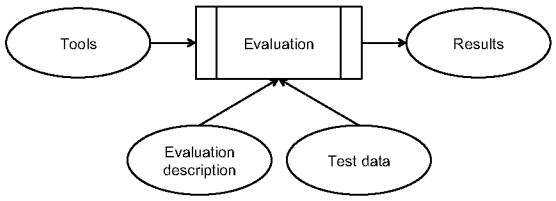

Fig. 1. Main entities in a software evaluation scenario

This high-level classification of entities can be further refined as needed. For example, in the context of SEALS, tools are classified into different types of semantic technologies according to their functional scope, namely, *ontology engineering tools*, *ontology storage systems*, *matching tools*, etc.

Similarly, it is also possible to distinguish different types of test data: *persistent test data* (those whose contents are stored in and physically managed by the evaluation platform), *external test data* (those whose contents reside outside the evaluation platform and whose life cycle is not controlled by it), and *synthetic test data generators* (pieces of software that can generate synthetic test data on-demand according to some determined configuration).

In accordance with the approach followed in the IEEE 1061 standard for a software quality metrics methodology [10], evaluation results are classified according to their provenance, differentiating *raw results* (those evaluation results directly generated by tools) from *interpreted results* (those generated from other evaluation results).

Besides, our entities include not only the results obtained in the evaluation but also any contextual information related to such evaluation, a need also acknowledged by other authors [6]. To this end, we also represent the information required for automating the execution of an evaluation description in the platform that, with the other entities presented, allows obtaining traceable and reproducible evaluation results.

Finally, another type of entities are *evaluation campaigns*, which represent the information needed to support the organization and running of campaigns for the evaluation of different (types of) participating tools. An evaluation campaign contains one or more *evaluation scenarios*, which include the evaluation description and test data to be used for carrying out the evaluation and the tools that will be evaluated.

Each of the abovementioned entities is composed of two different elements: the *data* that define the entity itself and the *description* of the entity, that is, the set of metadata that characterizes the entity (both generally and specifically) and enables the provision of the discovery mechanisms required for entity integration, consumption, and administration by the evaluation platform.

III. Entities Life Cycle

Different entities have different life cycles in the evaluation platform. This section describes the life cycles of the most relevant entities.

Tools, test data, and evaluation descriptions are defined in the platform as *artifacts*, which are a collection of *artifact versions*; for example, a particular tool can have a number of different tool versions that evolve over time.

Figure 2 shows state diagrams for artifacts and artifact versions, including the possible states, the operations that alter the state, and the operations that retrieve the entity information (data and metadata) in dotted arrows. It can be observed that, once registered in the platform, artifacts can always be retrieved and have a single state until they are deregistered. On the other hand, artifact versions have two states, published and unpublished; in the former state artifact versions can only be retrieved, and in the latter state they can only be updated. In this way, evaluations can only be performed using fixed (i.e., published) artifact versions.

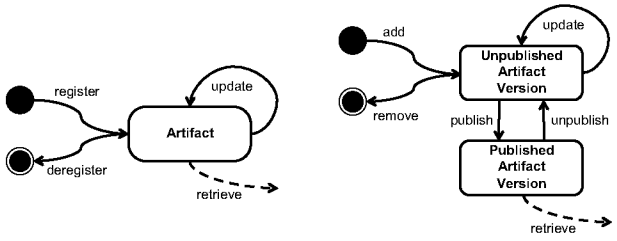

Fig. 2. Life cycle of artifacts (left) and artifact versions (right).

Evaluation results (raw results and interpretations) are defined as artifacts with no version information. Additionally, once registered they cannot be updated.

Evaluation descriptions are processed by the evaluation platform through *execution requests*. An execution request encapsulates the execution needs that a particular user has at some point in time, i.e., which evaluation description is to be executed, which tools shall be evaluated, which test data shall be used for its evaluation, etc.

During its life cycle, an execution request transits among eight different states, as shown in Figure 3. The starting state of an execution request is that of *"pending"*, which takes place whenever a new execution request is created.

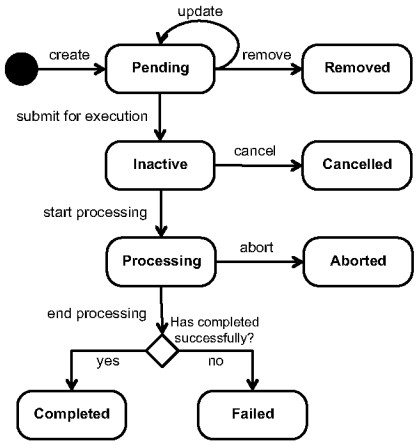

Fig. 3. Life cycle of an execution request.

At this point, the execution request can be updated, removed, or submitted for execution. Whereas the first operation

does not change the state of the execution request, the other two do change it: on the one hand, when the execution request is removed, the state transits to the *"removed"* state, a state in which no further operations are possible[2]; on the other hand, when the execution request is submitted to execution, the state transits to the *"inactive"* state. Beyond this point, the execution request shall not be further modified.

While the execution request is inactive, two possible courses of action can take place: it can be cancelled or it can start being processed. In the first case, the state transits to the *"cancelled"* state, a state in which, again, no further operations are possible. The latter case takes place once the execution requirements of the execution request are fully satisfied and, then, the state transits to the *"processing"* state.

Once the execution request is being processed, three possible outcomes may occur: (1) The execution request may be completed successfully, and thus the state transits to the *"completed"* state. (2) Some failure might prevent completing the execution of the execution request, causing the state to transit to the *"failed"* state. (3) It is also possible that the processing of the evaluation request is aborted (e.g., due to an abnormal duration time), thus forcing the state to transit to *"aborted"*. Regardless of the course of action, no further operations over the execution request will be carried out.

As can be seen, execution requests are not disposed by the evaluation platform. On the contrary, regardless of the execution request's internal state, its information is available to the user at any time, providing a complete and historical view of the evaluation activities over time.

IV. Upper Ontology

The entities presented above share a number of common properties. In this section we explore these properties and design the upper ontology that will be used to represent them. We have reused the Dublin Core vocabulary [11] because it already defines a consensual set of properties for describing resources.

Every entity managed by the evaluation platform can be described in terms of a top-level entity, which can be further specialized. As mentioned in section III, tools, test data, evaluation descriptions, and results are described in terms of an artifact, which is a collection of artifact versions. In the case of results, only an artifact is used to describe them since they do not include version information.

Given this information, we define three classes within the upper ontology, namely, *Entity*, *Artifact*, and *ArtifactVersion*, where the last two are specializations of the first one, as can be seen in Figure 4.

Entity descriptions include their creator, creation time, name, identifier, and description. Regarding artifacts, we also store the person responsible for the artifact, an URL with further information about it, and the collection of artifact versions (identifying the current version). Each artifact version

[2]That is, the state of the execution request will not be changed beyond this point.

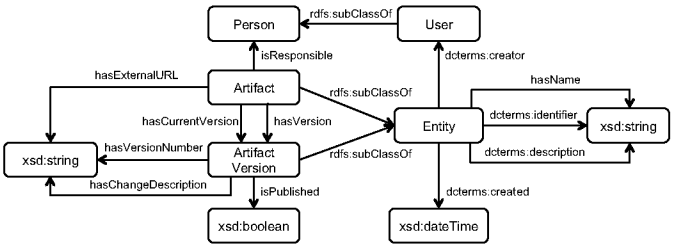

Fig. 4. Overview of the upper ontology.

is described by a version number, a description of the changes related to the version, and a flag to indicate whether it is published or not (as explained in Section III).

We have to note that we differentiate between a person and a platform user as a specific type of person, since only the latter can create entities in the platform. For describing them, we extend the FOAF [12] and VCard [13] vocabularies.

V. Modeling Software Evaluations

Figure 5 provides an overview of the main classes and properties defined for modeling software evaluations. Although they are not shown in the figure, between each type of artifact and its version we have defined subproperties of *hasVersion* (e.g., "*Tool hasToolVersion ToolVersion.*"). Next, we describe the main entities represented in the figure.

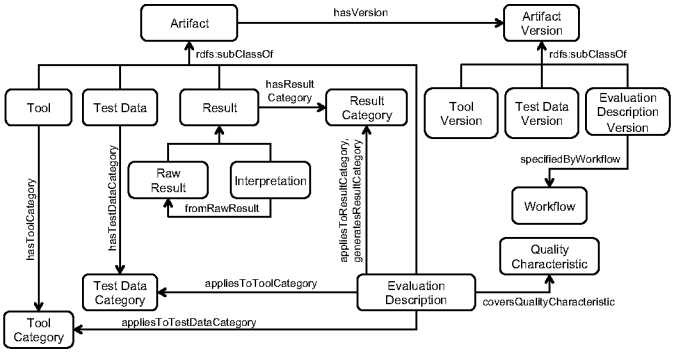

Fig. 5. Main classes and properties for modeling software evaluations.

Tools are defined by classifying them into a certain category, and the definition of each tool version specifies its particular capabilities, the hardware platform and operating system upon which it can be executed, and its execution requirements.

As described in Section II, we cover three types of test data, namely, persistent test data, external test data, and synthetically generated test data. In order to allow the evaluations to be reproducible, only persistent test data will be used in them. For external test data, further details need to be specified, such as the location where they can be accessed and when they are persisted into the platform as a test data version, or the date when they were retrieved. Similarly, when persistent test data is synthetically created by running a test data generator, the test data generator used for creation must be stored, as well as the configuration used to execute the generator.

Besides specifying test data using this hierarchy, these test data must be categorised to facilitate their future retrieval (e.g., interoperability test data, scalability test data, etc.) and each test data version must define the tool capabilities that the test data version can exercise. Also, we group evaluation test data in test suites for a meaningful analysis of the produced evaluation results, as suggested by [6].

We also mentioned in Section II that results specialize into raw results and interpretations. Regardless of the type of result, we need to categorise any result generated and to specify which raw result was used to create a certain interpretation.

The definition of an evaluation description includes the categories of tools and test data that can be used with the evaluation description as well as the category of results that the evaluation description produces when it is executed. An evaluation description also specifies which software quality characteristics can be measured with it. We do not impose any software quality model to define these quality characteristics. Nevertheless, we suggest to use the quality characteristics defined in the quality model of the ISO/IEC 9126 standard on software product quality [14].

An evaluation description also includes an execution contract that defines what the evaluation description needs to be executed and the type of results that will be produced. The workflow with the executable specification of an evaluation description is included in the evaluation description versions.

VI. Modeling Execution Requests

An evaluation execution request is associated to a certain evaluation description version. Thus, an execution request is totally coupled with the contract specified by the evaluation description to which the version belongs (see Figure 6). In this way, the evaluation platform is able to verify whether the evaluation description version can be enacted.

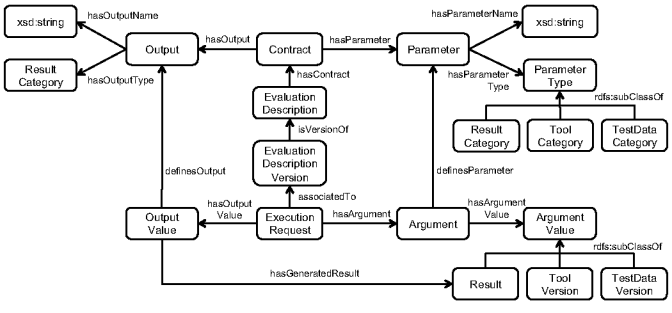

Fig. 6. Main classes and properties for modeling execution requests.

An execution contract defines what an evaluation description needs in order to be executed and the output that it will produce with the specified input. The inputs are specified by their parameter names and by the types of artifacts that can be used within these parameters (i.e., tool, test data and result categories) and the outputs are specified by the output names and the types of artifacts that can be used within these outputs (which are always result categories).

To execute a contract, the execution request must specify the arguments that define each parameter of the contract and the particular values of each argument. Upon execution, the platform will indicate the results that were generated by specifying the output values that define each output of the contract and the particular result generated.

Besides, the platform will also manage information about the life cycle of execution requests (the states presented in Section III and the moments of time when they change) and the computing resources that were used for the execution of evaluation descriptions.

In this way, an insight is provided about the configuration of the evaluation platform that was used for carrying out the evaluation with the objective of enabling the reproducibility of the evaluations afterwards, since the platform configuration may have a direct effect on the results obtained (e.g., in efficiency or scalability evaluations).

VII. Modeling Evaluation Campaigns

As mentioned above, we also want to model information about evaluation campaigns. As shown in Figure 7, these campaigns are activities organized by some users and contain a set of evaluation scenarios that will be executed over some participating tools. For each evaluation scenario, it is necessary to identify the evaluation description version and test data version used, as well as who will be participating (both the user participating and the tool version). Finally, it is necessary to identify the execution requests by which the various evaluation scenarios of a campaign are executed.

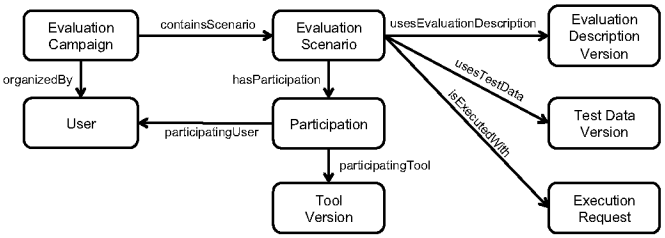

Fig. 7. Main classes and properties for modeling evaluation campaigns.

VIII. Related Work

By the time of writing this paper, we have not found any ontology or other conceptual model that allows representing all the entities involved in a software evaluation activity in an integrated way. What we have found are some models that cover parts of this activity or are related to it that are worth mentioning.

Next, we present an overview of the models that can be used to represent software quality characteristics, software measurement concepts, measurement and evaluation results, and software projects; we also discuss the main similarities and differences between these models and the one presented in this paper.

A. Software Quality Characteristic Ontologies

Different researchers have dealt with the development of ontologies for representing software quality characteristics. For example, [4] includes an analysis over different software quality documentation aspects (standards, publications, etc.) with the goal of identifying common software quality attributes and extracting the relevant concepts and relationships along with their frequency of use; also, [3] and [5] describe ontologies to represent quality characteristics in domain engineering and in the software product audit areas, respectively. Unfortunately, no implementation of any of these ontologies is available to be reused.

In our case, as Section V mentions, we allow defining the quality characteristics that an evaluation description covers although we do not impose any quality model.

B. Software Measurement Ontology

The Software Measurement Ontology (SMO) [7] has been defined by analysing software measurement standards and research proposals and is composed of a set of OWL ontologies for representing software measurement concepts and properties.

The idea behind SMO is similar to the one in our work. In SMO, a measurement result is produced by the measurement performed over an entity following some measurement approach; in our case, an evaluation result is produced by the execution request performed over some tool following an evaluation description.

However, the different scope of both works (SMO tries to cover the whole terminology involving software measurement, whereas we focus on the automated evaluation of software products) shows some differences in the resulting models.

Similarly as in our model, SMO differentiates between the results produced with base measures (raw results) and those produced with derived measures or indicators (interpretations). Nevertheless, in our case we do not differentiate between derived measures and indicators because we do not have a specific interest in the type of measurement approach used to obtain the interpretations.

Moreover, in our model we do not deal with details regarding the concrete tool attributes evaluated, measurement scales, and units of measurement; we leave this information open to be modeled in concrete evaluations as needed.

On the other hand, we define in detail the measurement method (evaluation description) to be used by specifying an executable evaluation workflow, inputs (test data), and a contract.

In the future we will analyse whether to extend our model with some of the general concepts of software measurement included in SMO in order to increase its applicability.

C. The MiniSQUID Model

The MiniSQUID model [6] allows storing the measurements obtained from software measurement programs in companies.

The main differences between this entity-relationship model and our way of representing results are that the MiniSQUID model covers the measurement of any type of software entity in the software development process whereas our model tackles only software products.

Moreover, measurement units and scales are a cornerstone of the MiniSQUID model, whereas we currently leave this aspect open, so that evaluation designers can freely decide the units and scales to be used.

D. Evaluation and Report Language

The Evaluation and Report Language (EARL) [15] defines a vocabulary for expressing test results in RDF(S) [16]. By the time of writing this paper, it is a working draft in the W3C.

The approach for representing results in EARL is similar to our approach. EARL states that a result is obtained when an assertor (i.e., who runs the test) tests some subject (i.e., what is being tested) according to a certain test criterion (i.e., a requirement or test case), and in our case we state that an evaluation result is obtained when some tool is evaluated according to certain test data.

Even if the primary goal of EARL was the exchange of test results between Web accessibility evaluation tools, it is designed to be flexible enough to cover other types of test results. This flexibility makes the scope of EARL broader that ours since it takes into account multiple types of assertors and subjects, whereas in our case the only assertor is the evaluation platform and the only type of subject is software.

However, the main difference between EARL and the metadata presented in this paper is their focus; while EARL is mainly centered on testing, our work is mainly focused on evaluation. In EARL test results are expected to inform about whether a subject passes a test or not and to provide pointers to the parts of the subjects relevant to the result, whereas in our case evaluation results describe a subject according to a set of metrics and disregard which parts of the subject are relevant to the results.

While EARL (as well as the previously-mentioned MiniSQUID model) only covers the representation of results, we cover the representation of all the resources involved in the evaluation life cycle (from the description of the evaluation and test data to the evaluation execution) so that the evaluation can be reproduced at a later stage. Another main difference is that we expect the entities involved in the evaluation (tools, evaluation descriptions, and test data) to change over time; therefore, we model these artifacts and their sets of versions in a different way.

However, once EARL becomes a W3C recommendation, we will study the feasibility of defining an alignment between our vocabulary and the one proposed in EARL in order to be able to export the results produced by the SEALS Platform in terms of the EARL vocabulary.

E. Description of a Project

The Description of a Project (DOAP) vocabulary[3] can be used to describe software projects (in particular open

[3] http://trac.usefulinc.com/doap

source ones), different versions of a project, and any related specifications and repositories. Similarly to our case, DOAP also differentiates between a software project and the different versions (releases) of it.

We do not reuse this vocabulary for describing tools because the DOAP terms that are relevant to our case are already included in our current descriptions of the Tool and ToolVersion classes (and their super-classes). Because of this, the alignment between our proposal and DOAP is straightforward in case interchanges between the two vocabularies are required.

IX. Conclusions and Future Work

This paper presents a first version of an ontology model that can be used to represent any information required for evaluating software or for organising an evaluation campaign over software products.

Our work has been guided by the need to obtain an interoperable and extensible model that supports automated and reproducible software evaluations. Even if no standard or model fully covers these needs (mainly because standards cannot cover every specific application need), we have been inspired by existing efforts and have tried to reuse accepted and well-known standards and proposals as much as possible during the development of the ontologies.

The ontologies presented here are generic by design, and concrete evaluations will have to extend them according to their needs. While these extensions will support each specific evaluation, the generic ontologies will allow aggregating multiple evaluations and evaluation results for further processing.

This ontology model has been implemented in terms of lightweight OWL [17] ontologies, which are available in the Web[4]. We decided to use lightweight ontologies because we agree with the authors of [18] when they advice to use ontologies as light as possible to cover project requirements.

We also plan to publish all the data about evaluations and evaluation campaigns as RDF data in the SEALS Platform and we encourage other practitioners to reuse the ontologies presented in this paper in their own evaluations to allow an easier integration of software evaluations and their results.

Even if these ontologies have been obtained through an extensive analysis of the literature and of our current requirements, we expect that new needs are identified and, therefore, these ontologies will change in the future. For example, currently, the ontologies are mainly focused on automated software evaluations, but in the future, they will be extended to allow the insertion of results produced by manual evaluations.

Furthermore, if we want to consistently compare results across different evaluations, we need to model information regarding the scales and units of evaluation results. This information is already covered in other models and right now we leave it open so users can model it according to their specific evaluations. Nevertheless, when needed, this information will be specified in detail in our model.

Finally, as mentioned in the previous section, since software evaluation is highly related to other areas (e.g., software measurement, software testing, etc.) we will analyse whether our ontologies could be applied to these areas and which extensions are required to achieve this goal.

Acknowledgment

This work is supported by the SEALS European project (FP7-238975). Thanks to Rosario Plaza for reviewing the grammar of this paper.

References

[1] A. Abran, J. Moore, P. Bourque, and R. Dupuis, Eds., *SWEBOK: Guide to the Software Engineering Body of Knowledge*. IEEE Press, 2004.

[2] A. Abran, J. J. Cuadrado, E. García-Barriocanal, O. Mendes, S. Sánchez-Alonso, and M. A. Sicilia, "Engineering the ontology for the SWEBOK: Issues and techniques," in *Ontologies for Software Engineering and Software Technology*, C. Calero, F. Ruiz, and M. Piattini, Eds. Springer, 2006, ch. 3, pp. 103–121.

[3] R. A. Falbo, G. Guizzardi, and K. C. Duarte, "An ontological approach to domain engineering," in *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE 2002)*. New York, NY, USA: ACM, 2002, pp. 351–358.

[4] A. Kayed, N. Hirzalla, A. A. Samhan, and M. Alfayoumi, "Towards an ontology for software product quality attributes," in *Proceedings of the 2009 Fourth International Conference on Internet and Web Applications and Services (ICIW 2009)*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 200–204.

[5] R. C. de Boer and H. van Vliet, "QuOnt: an ontology for the reuse of quality criteria," in *Proceedings of the 2009 ICSE Workshop on Sharing and Reusing Architectural Knowledge (SHARK 2009)*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 57–64.

[6] B. A. Kitchenham, R. T. Hughes, and S. G. Linkman, "Modeling software measurement data," *IEEE Trans. Softw. Eng.*, vol. 27, no. 9, pp. 788–804, 2001.

[7] M. F. Bertoa, A. Vallecillo, and F. García, "An ontology for software measurement," in *Ontologies for Software Engineering and Software Technology*, C. Calero, F. Ruiz, and M. Piattini, Eds., 2006, ch. 6, pp. 175–196.

[8] R. Studer, V. R. Benjamins, and D. Fensel, "Knowledge engineering: Principles and methods," *IEEE Transactions on Data and Knowledge Engineering*, vol. 25, no. 1-2, pp. 161–197, 1998.

[9] *ISO/IEC 14598-6: Software product evaluation - Part 6: Documentation of evaluation modules*. ISO/IEC, 2001.

[10] *IEEE 1061-1998. IEEE Standard for a Software Quality Metrics Methodology*. IEEE, December 1998.

[11] DCMI Usage Board, "DCMI Metadata Terms," Dublin Core Metadata Initiative, Recommendation, 14 January 2008. [Online]. Available: http://dublincore.org/documents/dcmi-terms/

[12] D. Brickley and L. Miller, "FOAF vocabulary specification 0.97," FOAF Project, Namespace Document - 3D Edition, 1 January 2010. [Online]. Available: http://xmlns.com/foaf/spec/

[13] H. Halpin, R. Ianella, B. Suda, and N. Walsh, "Representing vCard objects in RDF," W3C, Member Submission, 20 January 2010. [Online]. Available: http://www.w3.org/TR/vcard-rdf/

[14] *ISO/IEC 9126-1. Software Engineering – Product Quality – Part 1: Quality model*. ISO/IEC, 2001.

[15] S. Abou-Zahra and M. Squillace, "Evaluation and Report Language (EARL) 1.0 Schema," W3C, Last Call WD, Oct. 2009. [Online]. Available: http://www.w3.org/TR/2009/WD-EARL10-Schema-20091029/

[16] Brickley, D., Guha, R.V. (editors), "RDF Vocabulary Description Language 1.0: RDF Schema," W3C, Recommendation, 10 February 2004. [Online]. Available: http://www.w3.org/TR/rdf-schema/

[17] D. McGuiness and F. van Harmelen, "OWL Web Ontology Language Overview," W3C, Recommendation, 10 February 2004. [Online]. Available: http://www.w3.org/TR/owl-features/

[18] F. Ruiz and J. R. Hilera, "Using ontologies in software engineering and technology," in *Ontologies for Software Engineering and Software Technology*, C. Calero, F. Ruiz, and M. Piattini, Eds. Springer, 2006, ch. 2, pp. 49–102.