A Quality Perspective of Software Evolvability Using Semantic Analysis

Philipp Schugerl[1], Juergen Rilling[1], René Witte[1], Philippe Charland[2]
[1]Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
{p_schuge, rilling, rwitte}@cse.concordia.ca
[2]System of Systems Section
Defence R&D Canada – Valcartier
Quebec, Canada
philippe.charland@drdc-rddc.gc.ca

Abstract

Software development and maintenance are highly distributed processes that involve a multitude of supporting tools and resources. Knowledge relevant to these resources is typically dispersed over a wide range of artifacts, representation formats, and abstraction levels. In order to stay competitive, organizations are often required to assess and provide evidence that their software meets the expected requirements. In our research, we focus on assessing non-functional quality requirements, specifically evolvability, through semantic modeling of relevant software artifacts. We introduce our SE-Advisor that supports the integration of knowledge resources typically found in software ecosystems by providing a unified ontological representation. We further illustrate how our SE-Advisor takes advantage of this unified representation to support the analysis and assessment of different types of quality attributes related to the evolvability of software ecosystems.

Keywords: Ontology, software evolvability

1. Introduction

Constant changes in our society and in technology do not only affect our daily lives, but also software development processes and the requirements our software systems have to satisfy. Developers are responsible for identifying functional requirements, implementing them, and ensuring that these requirements evolve according to customer needs during system maintenance. However, from an organization perspective, it is not sufficient to only satisfy these functional requirements. Software systems must also satisfy non-functional requirements (NFR) such as security, safety, and dependability [1], which are critical for the success of most software systems.

One of the major technology shifts in recent decades affecting our society has been the widespread use of the Internet, resulting in new forms of social interaction and the need to organize knowledge across this new medium. It provides an enabling technology for the ongoing globalization of software development and the support for multi-site development. Consequently, organizations must deal with new and highly collaborative software ecosystems that involve distributed (virtual) workspaces, processes, and knowledge management.

NFRs are often referred to as the qualities of a system and can be divided into two main categories: (1) execution qualities, such as performance and usability, which are observable at run time; and (2) evolution qualities, such as testability, maintainability, extensibility, and scalability, which are embodied in the static structure of the software system.

Existing software development process models still mainly focus on capturing functional software requirements and NFRs are not well integrated into these models or processes. The situation is further complicated since the assessment of NFRs in software systems is often a critical aspect for the acceptance of a final software product.

The work in this article is a continuation of our previous research on semantic modeling of software artifacts [10] and process modeling [2]. In this research, we extend our semantic software engineering environment (SE-Advisor) to support the evaluation and assessment of various non-functional quality attributes found in evolving software ecosystems. We argue that ontologies not only promote and support the conceptual representation of a software ecosystem, but also support semantic modeling for quality information representation and communication [3, 4]. Furthermore, we illustrate enhancements to our SE-Advisor to support the analysis of NFRs as they can be found in evolving software ecosystems.

Our research is significant for several reasons: (1) We introduce a semantically rich unified model for

various knowledge resources and digital components; and (2) we present our SE-Advisor, an environment that can provide automated or semi-automated tool support in analyzing and/or assessing NFRs related to the evolvability of software ecosystems.

The remainder of the paper is organized as follows: Section 2 provides background relevant to evolving software ecosystems, NFRs, and ontologies. Section 3 discusses the motivation and objectives for our approach, including some typical use cases addressing quality assessment of NFRs. Section 4 introduces the implementation of our SE-Advisor. Section 5 illustrates its applicability in assessing various quality aspects. Section 6 presents related work, followed by conclusions and future work in Section 7.

2. Non-Functional Requirements and Evolving Software Ecosystems

2.1 Evolving Software Ecosystems

The term ecosystem was introduced in 1930 by Roy Clapham [5], who was asked if he could think of a suitable word to denote the physical and biological components of an environment that are considered in relation to each other as a unit. Some consider it as a basic unit in ecology, characterized by energy and matter flows between the different elements that compose it. From a software perspective, a *software ecosystem*, sometimes called a *digital ecosystem*, refers to software/digital components, which exist within a computer as a functioning unit. Digital components could be software components, applications, services, knowledge, business processes and models, training modules, contractual frameworks, or laws. A digital component is a useful idea, expressed by a language (formal or natural), digitized and transported within the ecosystem, which can be processed by humans or by computers.

From our perspective, a *software ecosystem* employs a broader set of components, such as software services, knowledge, process representations, and artifacts at different abstraction and semantic levels. The infrastructure of such a software ecosystem consists of a pervasive digital environment, which is mainly populated by digital components and supports the description, composition, evolution, integration, sharing, and distribution of knowledge. The social and business network topologies found in these ecosystems are often not hierarchical [6], making it difficult to identify and implement a single functional reference model, due to the heterogeneity of the components. Similar to biological ecosystems, software ecosystems are not static. They must evolve in order to adapt to changing requirements and environmental contexts, while still satisfying their expected functional and non-functional requirements.

2.2 Non-Functional Requirements

Unlike functional requirements, where a single analysis technique (e.g., use case modeling) is sufficient to identify essentially all requirements, the same analysis is not appropriate for all quality requirements. Quality, as defined by ISO 9000:2000 [7], is the "degree to which a set of inherent characteristics fulfills requirements", where a requirement is a "need or expectation that is stated, generally implied or obligatory". There has been a significant body of work on classifying requirements in order to establish links between qualities and requirements. All of these classifications have in common that they distinguish between some form of *functional* and *non-functional* requirements (e.g., they might refer to them as *behavioral* vs. *non-behavioral* requirements) [8]. In [9], a classification was introduced that defines functional requirements as those requirements that describe what a system should do, whereas all remaining requirements are regarded as non-functional. Davis [8] classifies non-functional requirements as qualities and uses Boehm's quality tree [1] as a sub-classification. The IEEE standard 830-1993 on Software Requirements Specifications [11] further classifies NFRs as external interface requirements, performance requirements, attributes and design constraints, where the attributes are a set of qualities such as reliability, availability, security, etc. The distinction between functional and non-functional requirements is further complicated by situations where the same requirement can be considered either as functional or non-functional. The classification might depend on the level of detail expressed by the requirements [12] or on the interpretation of these requirements (soft or hard). For example, the response time requirements in a user interface are typically soft (NFRs), whereas response time in real-time systems can be considered as a functional requirement. These examples show that the distinction of functional vs. non-functional requirements and their various sub-classifications are often fuzzy and imprecise.

In our research, we refer to NFRs as: *Qualities of a system, which can be classified as either behavioral or evolvability aspects. Behavioral* qualities include, among others, performance, usability, and reliability, all qualities that are observable typically at run time. *Evolvability,* the focus of our research, corresponds mainly to static aspects of the software product and the ecosystem in which the software was developed. Evolvability qualities include process maturity,

maintainability, extensibility, traceability, and quality of artifacts that are part of a software ecosystem.

2.3 Ontologies and Reasoning

Ontologies have been widely used in computer science to formally define domains of discourse. They can be described as a conceptualization of explicit information [3] that consist of properties, concepts (also often referred as classes), and relations between them. In contrast to databases, ontologies allow us to work with incomplete knowledge, due to their support for an open world assumption. Additionally, their formal representation allows for the use of ontological reasoning services and easy extensibility [3, 14].

Ontologies are an important part of knowledge modeling and sharing, by acting as a common language [3]. The Web Ontology Language (OWL) [15, 16] has long been standardized by the W3C and has paved the way for a machine-understandable Internet, in which Web resources can be automatically processed and reasoned about. More recently, ontologies have been introduced in engineering domains and are now widely accepted as an enhanced form of knowledge representation. Our research uses OWL-DL, a sub-language of OWL, that is based on Description Logics (DLs) [14]. DL-based knowledge systems allow us to enrich our platform with reasoning services such as Pellet or Fact++ [17, 18]. Reasoning systems within OWL-DL have proven to be sound, complete, terminating, and provide us with automatic inferences. By incorporating semantics and data in a uniform ontological model, an expressive representation of knowledge is formed. Ontologies have been used in the software engineering domain to model general terms and relations found within the domain and to perform logical consistency checks. For a more detailed discussion on OWL ontologies, DL, and reasoning, we refer the reader to [14, 16, 46].

3. Analysis and Quality Assessment

One of the major challenges during the analysis and assessment of NFRs is the distribution of relevant information across digital software ecosystems. NFRs such as a system's evolvability or maintainability depend frequently on the ability to correlate distributed components or knowledge resources. For example, the prediction of faults in a component might require accessing bug tracking information, the source code of the component, and data from the version control system [18, 31]. In order to facilitate the analysis of NFRs, knowledge from distributed components and resources needs to be integrated and semantically modeled. Given such a common representation, automated or semi-automated knowledge exploration and analysis are needed to deal with these large knowledge resources.

In what follows, we introduce two use cases describing evolvability related NFRs in software ecosystems. Use case #1 addresses the need to assess quality properties of artifacts at different scope and abstraction levels (Figure 1). Use case #2 focuses on supporting and improving a system's evolvability. These use cases illustrate how our SE-Advisor can provide (semi-) automated support in assessing the evolvability of software artifacts.

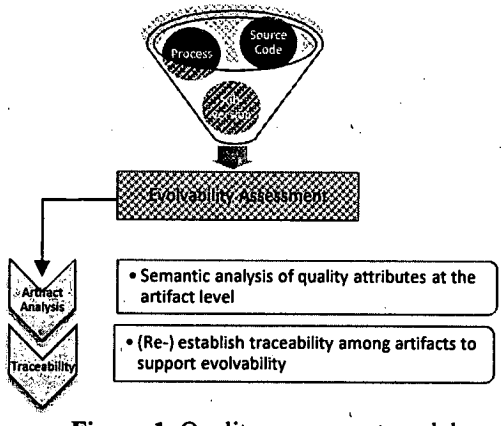

Figure 1. Quality assessment model

Use case #1: *Quality assessment at the artifact level.* As part of software ecosystems with mature software development processes, a variety of software artifacts are created and stored in large repositories. Such repositories contain explicit and implicit knowledge that can be mined to provide additional insights and provide continuous guidance during software development and to plan evolutionary aspects of software projects. Bug reporting systems are an example of such repositories, allowing users to submit, describe, track, comment on, and classify bug reports and feature requests. These systems typically contain a large number of bug reports that are manually evaluated to assess their quality and content. Either automated or at least semi-automated tool support is needed to improve the assessment process.

Use case #2: *Integrate and trace knowledge resources.* Programmers tend to spend a large amount of manual effort on synthesizing and integrating information from various sources in order to establish their connections, i.e., the traceability links [19, 20, 21]. Various software and software evolution artifacts, like requirements, design documents, and bug reports contain a large amount of information in the form of descriptions and text written in natural language. These documents,

combined with source code, represent two of the main software artifact types utilized in software evolution [19, 20]. Existing source code/document traceability research [19, 21] mainly focuses on connecting documents and source code using Information Retrieval (IR) techniques. However, these IR approaches often ignore structural and semantic information that can be found in both documents and source code, limiting therefore their precision and applicability. Source code analysis and text mining techniques have to be combined with a richer semantic representation to establish and maintain traceability links among artifacts, in order to improve the evolvability of these artifacts and the software ecosystem.

Before describing in detail how we address these use cases in our SE-Advisor, we introduce the system architecture and describe some of the major implementation aspects.

4. System Architecture Overview

The SE-Advisor is a continuation of our previous work [2, 10] that we extended to support the assessment of non-functional quality requirements. Knowledge integration and sharing, two key aspects of our environment, are performed through a unified ontological representation. For a more detailed discussion of our ontologies, their design, and integration, we refer the reader to [2, 10]. The ontological knowledge base (KB) is hosted as a central repository on a server, with persistent storage being provided by a RDBMS.

Given the distributed environment in which our SE-Advisor is used, we adopted a client-server architecture in which clients communicate over a network with a knowledge/advice server (Figure 2). The communication between the clients and server is realized as a Representational State Transfer (REST) web service. REST is a software architecture for stateless client-server communication and is used as a simple and intuitive method to realize well-defined CRUD (create, read, update, delete) operations through the standard HTTP protocol. The exchange format used is XML, which allows human-readable and therefore easily analyzable and maintainable communication. In order to serialize objects to XML, the XStream [22] library is used. XStream takes any POJO (Plain Old Java Object) and produces an XML representation from it. Circular references are automatically resolved and clean XML is produced. Our server application is deployed on a Tomcat server and secured through HTTPS.

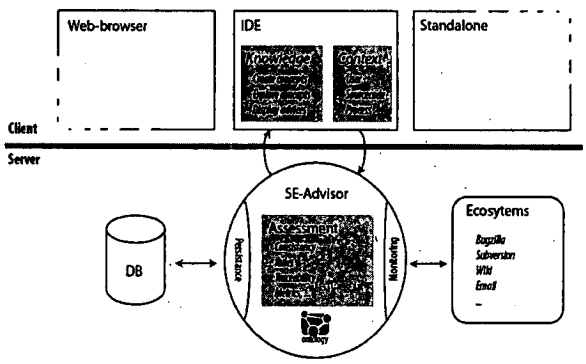

Figure 2. SE-Advisor system architecture

The server is responsible for analyzing an ecosystem and assessing the collected information in order to populate the software engineering ontology (see Section 5). Monitoring is performed through system hooks or is time-triggered (i.e., nightly scheduled). Updates are propagated to the ontological KB, whereby information is pre-processed, aggregated, and assessed. Pre-processing includes the standardization of date/time types or the elimination of non-valid characters. Aggregation calculates certain metrics (e.g., LOC) or clusters text to paragraphs (e.g., in wikis and emails). Finally, during the assessment step, we generate and extract additional semantic rich information. A more detailed discussion of the analysis process is provided in the next section. In order to build and modify our ontology, we used the Protégé API. Protégé is an ontology editor and knowledge acquisition system that provides a rich API to perform many useful operations on ontologies. Database persistence is realized through Jena (a Java framework for building semantic web applications), which is also one of the building blocks of Protégé.

5. Assessing Non-Functional Quality Requirements

In this section, we discuss how the SE-Advisor supports the two use cases introduced in section 3.

5.1 Artifact Evaluation - Quality of Bug Reports (Use Case #1)

The process of evaluating, classifying, and assigning bugs to programmers is a difficult and time-consuming task, which greatly depends on the quality of the bug reports themselves. It has been shown [23, 24] that the quality of reports originating from bug trackers or ticketing systems can vary significantly. Therefore, bug quality is an excellent example for quality assessment at the artifact level (*use case #1*).

Our approach applies both IR and Natural Language Processing (NLP) techniques for mining bug repositories. Natural language properties influencing the report quality are automatically identified and applied as part of a classification task. As a result, our automated approach supports important aspects in evaluating the evolvability of a large digital ecosystem as well as in the assessment of quality and maturity of bug trackers.

For the automated assessment, we introduce a new set of quality guidelines used for the evaluation of free form bug descriptions typically associated with bug reports. The attributes themselves are derived from results observed in [23, 24] and from general guidelines for good report qualities, such the ones discussed in [25]:

Reproducibility. The bug report description includes steps to reproduce a bug or the context under which a problem occurred.

Observability. The bug report contains a clearly observed behavior, whether positive or negative. Evidence of the problem such as screenshots, stack traces, or code samples is provided.

Certainty. The level of speculation is embedded in a bug description. A high certainty indicates a clear understanding of the problem and often implies that the reporter can provide suggestions on how to solve it.

Focus. The bug description does not contain any off-topic discussions, complaints, or personal statements. Only one bug is described per report.

In order to validate our findings, we performed an assessment of the ArgoUML [26] bug repository. ArgoUML is a popular and mature open source UML drawing tool. We extracted a dataset consisting of 4,839 bug reports, which included 3,731 reported defects and 1,108 feature and enhancement requests. An evaluation of our automatic classification approach with 178 manually classified bug reports showed a relatively high precision (86%) for our automated approach in assessing and classifying the bugs correctly.

The extracted quality of bug reports and other bug tracker properties (such as links between bugs defined through *depends-on* or *resolved-by* properties) were mapped onto our SE ontology. We then performed ontology alignment [13] between our bug tracker sub-ontology and source code ontology (described in Section 5.2) by comparing the revision numbers, bug IDs, class, method, and namespace identifiers extracted through NLP. It must be noted at this point that this process is not complete (in the case of ArgoUML, only about 25% of bugs could be matched with source code or vice versa), but any new information gained through this alignment is valuable.

Given such knowledge about bug quality and its link to source code and other artifacts, we can now improve the assessment of a software system by taking into consideration the impact of bad quality across artifacts. This allows for a more precise evaluation of software systems in terms of evolvability. For a more detailed discussion on our approach of evaluating the quality of bug reports, we refer the reader to [27, 28].

5.2 Traceability (Use Case #2)

In a complex application domain like software engineering, knowledge has to be continually integrated from different sources (like source code repositories, documentation, test case results), at different abstraction levels (from single variables to complete system architectures), and across different relations (static, dynamic, etc.). Consequently, one of the major challenges for programmers is the need to comprehend and/or maintain this multitude of often disconnected artifacts. This lack of traceability among software artifacts is a result of several factors including: (1) artifacts written in different languages (natural language vs. programming language); (2) artifacts describing a software system at different abstraction levels (design vs. implementation); (3) applied organizational processes that do not enforce maintenance of existing traceability links. In what follows, we provide an example of (re-)establishing traceability links among a subset of these existing artifacts, namely software documents (requirements and design) and source code.

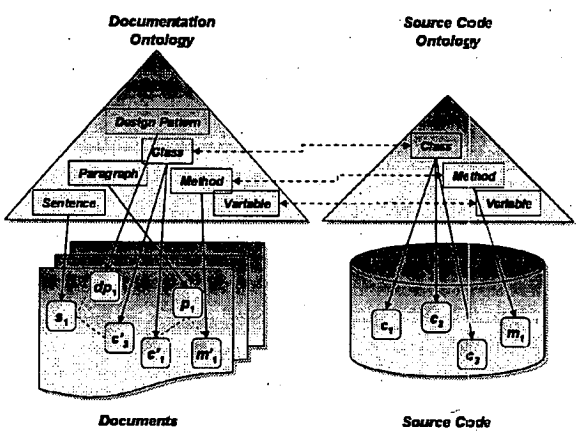

Figure 4: Linking instances from source code and documentation

Software documentation and source code are two of the major artifacts typically used as part of software

maintenance. Having source code and documents represented in an ontological form allows us to link instances from source code and documentation artifacts using existing approaches from the field of ontology alignment [13, 16].

Similar to our bug tracker sub-ontology, our approach focuses mainly on matching instances through source code analysis and text mining [45]. Our text mining system [2, 10] can take the results of the source code analysis as an input when detecting named entities. This allows us to connect instances from the source code and document ontologies. For example, our source code analysis tool may identify *c1* and *c2* as classes and this information can be used by the text mining system to identify named entities – *c'1* and *c'2* and their associated information in the documents (Figure 4). As a result, source code entities *c1* and *c2* are now linked to their occurrences in the document (*c'1* and *c'2*), as well as to other information related to these two entities in the document, such as design patterns, architectures, etc. The coherence of identified concepts (e.g., a design pattern and a namespace identifier appearing together in a document) is still a matter of ongoing work. For now, we consider concepts to be related in case they appear within the same paragraph. In the future, we would like to take into consideration further semantics of text (e.g., headings).

Our approach addresses an important issue in supporting and enhancing the NFR quality aspects of software ecosystems. Ontologies capture structural and semantic information conveyed in artifacts and therefore, allow the recovery and re-establishment of traceability links between software implementation and documentation at the semantic level. These identified instances are linked to source code entities and as a result, allows users to discover relations between source code and its design information at different abstraction levels. Furthermore, our documentation ontology identifies a large number of design level concept instances such as design patterns and architectural styles. Applying text mining technologies to the software domain allows us to utilize semantic information automatically extracted from software documents to support traceability tasks. Using state-of-the-art ontology reasoners such as Pellet [17], implicit relations between discovered concept instances can be inferred. As an example, in the case of ArgoUML, we are able to link the classes of the `org.argouml.model` package to the MVC (Model-View-Control pattern), as the two concepts are mentioned on the same page of the development wiki (http://argouml.tigris.org/wiki/). The resulting linked ontologies provide the ability to perform cross boundary queries between programming and natural language. This example illustrates how our SE-Advisor can support the traceability among knowledge resources in order to support their evolution (use case #2).

6. Related Work

Typically, NFRs are gathered and documented during system analysis to guide decisions during system design and implementation. Existing work on requirements classification has focused on establishing links between qualities and requirements [11, 12]. Other work has focused on making NFRs an integrated part of the development [35] or certification process (e.g., DO-178B [36]) of newly developed software. We focus on the quality assessment of existing software ecosystems.

There are many application examples for OWL as a knowledge representation in the software engineering domain, such as model-driven software development [37], a CMMI-SW model representation and reasoning for classifying organizational maturity levels [38], component reuse [39], and open source bug tracking [40]. However, there is only limited research in modeling software evolution using ontologies. Ruiz et al. [41] present a semi-formal ontology for managing software maintenance projects. They consider both the static and dynamic aspects such as the workflow in software maintenance processes. Kitchenham et al. [42] designed a UML-based ontology for software maintenance to identify and model factors that affect the results of empirical studies. Dias et al. [43] extended the work of Kitchenham by applying a first order logic to formalize knowledge involved in software maintenance. González-Pérez and Henderson-Sellers present a comprehensive ontology for software development [44] that includes a process sub-ontology that models process related concepts such as techniques, tasks, and workflows. Despite considerable research on ontologies representing functional requirements, little work exists in using ontologies to represent non-functional requirements. In [35], a NFR ontology was introduced that can be used to structure and express constraints as part of the quality of service specification. More recently, the collaborative nature of software engineering has been addressed by introducing Wiki systems into the SE process. Semantic Wiki extensions like Semantic MediaWiki [33] or IkeWiki [34] add formal structuring and querying extensions based on RDF/OWL metadata. This work can be seen as complementary to our approach, in that they can support the creation and visualization of the developed KB. However, by themselves, they do not address the main concern

covered by our approach, i.e, the assessment of NFRs in software ecosystems.

7. Conclusions and Future Work

Software systems in general must satisfy non-functional requirements often referred to as quality properties, such as security, safety, and evolvability. The ongoing globalization of software development through collaborative workspaces has introduced new types of software ecosystems. System knowledge tends to be dispersed over a wide range of artifacts, involving different representational formats.

In this research, we introduced an extension of our semantic software engineering environment (SE-Advisor). We presented an ontological approach to provide a semantic rich model for digital components found in many software ecosystems and applied it to an open source software project. We also discussed how this ontological representation can support the analysis and assessment of the evolvability of artifacts found in software ecosystems.

As part of our future work, we plan to analyze additional artifacts and further validate our approach and its ability to assess the evolvability of these software ecosystems' artifacts. We also plan to expand our work in other NFR areas, such as security and performance assessment.

Acknowledgement:
This research was partially funded by DRDC Valcartier (contract no. W7701-081745/001/QCV).

8. References

[1] Boehm B. et al. Quantitative Evaluation of Software Quality. Proceedings of the 2nd IEEE International Conference on Software Engineering. 592-605, 1976.

[2] W. J. Meng, J. Rilling, Y. Zhang, R. Witte, and P. Charland. An Ontological Software Comprehension Process Model. In: 3rd Int. Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006), Genoa, Italy, 2006.

[3] Gruber, T. R. A Translation Approach to Portable Ontology Specifications, Knowledge Acquisition, 5(2):199-220, 1993.

[4] Wongthongtham, P., Chang, E., Dillon, T.S. & Sommerville, I. Software Engineering Ontologies and their Implementation. The IASTED, , Austria, 2005.

[5] Willis A J. The ecosystem: an evolving concept viewed historically, Functional Ecology 11:2, pp 268-271,1997.

[6] A. L. Barabási, H. Jeonga, Z. Néda, E. Ravasz, A. Schubert and T. Vicsek. Evolution of the social network of scientific collaborations, 2002.

[7] ISO/IEC 9126-1. Software engineering – Product quality – Part 1: Quality model. International Organization for Standardization, 2001.

[8] Davis, A.. Software Requirements: Objects, Functions and States. Prentice Hall, 1992

[9] Gilb, T. Towards the Engineering of Requirements. Requirements Engineering 2, 3 165-169, 1997.

[10] J. Rilling, R. Witte, P. Schugerl, and P. Charland. Beyond Information Silos — An Omnipresent Approach to Software Evolution. Int. Journal of Semantic Computing (IJSC), Special Issue on Ambient Semantic Computing, May, 2009.

[11] IEEE. IEEE Recommended Practice for Software Requirements Specifications. IEEE Standard 830, 1993.

[12] Kotonya, G., I. Sommerville. Requirements Engineering: Processes and Techniques. John Wiley & Sons, 1998.

[13] N. F. Noy and H. Stuckenschmidt. Ontology Alignment: An annotated Bibliography. In Semantic Interoperability and Integration, Schloss Dagstuhl, Germany, 2005.

[14] F. Baader et al. The Description Logic Handbook, Cambridge Univ. Press, 2003.

[15] Web Ontology Language, www.w3.org/2004/OWL/ (last accessed 1/2009).

[16] OWL Web Ontology Language Reference, W3C Recommendation, www.w3.org/TR/owl-ref (last accessed 1/2009).

[17] Pellet the Open Source OWL Reasoner, clarkparsia.com/pellet (last accessed 2/2009).

[18] FaCT Plus Plus OWL-DL Reasoner, code.google.com/p/factplusplus (last accessed 2/2009).

[19] J. Ratzinger, T. Sigmund, P. Vorburger, H. Gall. Mining Software Evolution to Predict Refactoring, Proc. of the Int. Symposium on Empirical Software Engineering and Measurement (ESEM 2007), IEEE CS, p. 10, 2007.

[20] P. Arkley, P. Mason, and S. Riddle. Position Paper: Enabling Traceability, Proc. of 1st Int. Workshop on Traceability in Emerging Forms of Software Engineering, Edinburgh, Scotland, pp. 61–65, 2002.

[21] A. Marcus, J. I. Maletic. Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing, In Proceedings of 25th International Conference on Software Engineering, 2002.

[22] The XStream XML Serialization Library, xstream.codehaus.org (last accessed 2/2009).

[23] N. Bettenburg, S. Just, A. Schroter, C. Weiss, R. Premraj and T. Zimmermann. Quality of Bug Reports in

Eclipse, Proc. of the 2007 OOPSLA Workshop on Eclipse Techn, eXchange, Montreal, Canada, 2007.

[24] N. Bettenburg, S. Just, A. Schroter, C. Weiss, R. Premraj and T. Zimmermann. What Makes a Good Bug Report? (Revision 1.1), 2007.

[25] Simon Tatham. How to Report Bugs Effectively. www.chiark.greenend.org.uk/~sgtatham/bugs.html (last accessed 5/2008).

[26] ArgoUML Open Source Modeling Tool, argouml.tigris.org (last accessed 2/2009).

[27] Schugerl, P. and Rilling, J. and Charland, P. Mining Bug Repositories – A Quality Assessment, In Proc. Of the International Conference on Innovation in Software Engineering (ISE 2008), Vienna, Austria, 2008.

[28] Schugerl, P. and Rilling, J. and Charland, P. Enriching SE Ontologies with Bug Report Quality, 4th Int. Workshop on Semantic Web Enabled Software Engineering (SWESE 2008), Germany, 2008.

[29] P. Arkley, P. Mason, and S. Riddle, "Position Paper: Enabling Traceability," Proc. of 1st Int. Workshop on Traceability in Emerging Forms of Software Engineering, Edinburgh, (September 2002), pp. 61–65.

[30] A. Marcus, J. I. Maletic, "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing". In Proc. of 25th International Conference on Software Engineering, 2002.

[31] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," *IEEE Trans. Software Eng.*, vol. 31, pp. 429-445, 2005.

[32] Agostino, Claudio, Damiani, Ernesto and Frati, Fulvio. FOCSE: An OWA-based Evaluation Framework for OS Adoption in Critical Environments. Open Source Development, Adoption and Innovation.: Springer Boston, 2007.

[33] S. Schaffert. IkeWiki: A Semantic Wiki for Collaborative Knowledge Management. In WE-TICE, pages 388–396, IEEE Computer Society, 2006.

[34] M. Krötzsch, D. Vrandečić, and M. Völkel. Semantic MediaWiki. In I. Cruz, S. Decker, D. Allemang, C. Preist,D. Schwabe, P. Mika, M. Uschold, and L. Aroyo, editors, The Semantic Web – ISWC 2006, volume 4273 of LNCS, pages 935–942, Springer, 2006.

[35] Dobson, G.; Hall, S.; Kotonya, G. (2007).A Domain-Independent Ontology for Non-Functional Requirements, e-Business Engineering, 2007. ICEBE 2007. IEEE International Conference on e-Business, Volume , Issue , Page(s):563 – 566, 2007.

[36] RTCA Inc., RTCA/DO-178B: Software considerations in airborne systems and equipment certification, 1992.

[37] Ontological Driven Architectures and Potential Uses of the Semantic Web in Systems and SE, www.w3.org/2001/sw/BestPractices/SE/ODA/, (accessed Feb, 2009).

[38] Soydan, G. H. and Kokar, M., "An OWL Ontology for Representing the CMMI-SW Model", W. on Semantic Web Enabled Software Engineering (SWESE), 2006.

[39] Happel, H.-J., Korthaus, A., Seedorf, S., and P.Tomczyk, "KOntoR: An Ontology-enabled Approach to Software Reuse", 18th Int. Conference on Software Engineering and Knowledge Eng. (SEKE), July, 2006.

[40] Ankolekar, A., "Supporting Online Problem –Solving Communities with the Semantic Web", Ph.D.Thesis, Carnegie Mellon University, Pittsburgh, PA, 2005.

[41] Ruiz, F., Vizcaíno, A., Piattini, M., and García F., "An Ontology for the Management of Software Maintenance Projects", International Journal of Software Engineering and Knowledge Engineering, 14(3), 323-349, 2004.

[42] Kitchenham, B., Travassos, G.H., Mayrhauser, A.V., Niessink, F., Schneidewind, N.F., Sing-er, J., Takada S., Vehvilainen R., and Yang H., "Towards an Ontology of Software Maintenance", Journal of Software Maintenance and Practice, 11(6), 365-389, 1999.

[43] Dias, M. G. B., Anquetil, N., and Oliveira, K. M.: Organizing the Knowledge Used in Software Maintenance, J. of Universal C.S., pp. 641-658, 2003.

[44] C. Gonzalez-Perez, B. Henderson-Sellers,"Modelling software development methodologies: A conceptual foundation", Journal of Systems and Software 80(11): 1778-1796, 2007.[45] René Witte, Qiangqiang Li, Yonggang Zhang, and Juergen Rilling. Text Mining and Software Engineering: An Integrated Source Code and Document Analysis Approach. IET Software Journal, Volume 2, Issue 1, February 2008, pp.3–16. Special Section on Natural Language in Software Engineering.

[46] Haarslev, V. and Möller, R., RACER System Description, In Proc. of Int. Joint Conference on Automated Reasoning (IJCAR'2001), Springer-Verlag, 701-705, 2001.