OpenCert 2011

Pre-Proceedings of the
Fifth International Workshop on
Foundations and Techniques for
Open Source Software Certification

Montevideo, Uruguay, 15th November, 2011

Satellite Event of the 9th IEEE International Conference on
Software Engineering and Formal Methods (SEFM 2011)

Luís Soares Barbosa, Dimitrios Settas (editors)

High-Assurance Software Laboratory - HASLab
Universidade do Minho

HASLab:1:2011
November 2011

Preface

Over the past decade, the Free/Libre/Open Source Software (FLOSS) phenomenon has had a global impact on the way software systems and software-based services are developed, distributed and deployed. Widely acknowledged benefits of FLOSS include reliability, low development and maintenance costs, as well as rapid code turnover. Linux distributions, Apache and MySQL server, among many other examples, as a testimony to its success and resilience.

Such a success has brought with it an increasing interest to use FLOSS for complex and industrial-strength applications. However, state-of-the-art OSS, by the very nature of its open, unconventional, distributed development model, make software quality assessment, let alone full certification, particularly hard to achieve and raises important challenges both from the technical/methodological and the managerial points of view.

In such a context, the aim of the OpenCert series of workshops is to bring together researchers from the academia and the industry interested in the quality assessment of FLOSS and ultimately provide a platform for the establishment of a coherent certification process for FLOSS.

The 1st International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2007) was held on 31 March 2007 in Braga, Portugal, as a satellite event of ETAPS 2007. From then on OpenCert run regularly every year. In 2008, jointly with the *International Workshop on Foundations and Techniques bringing together Free/Libre Open Source Software and Formal Methods* (FLOSS-FM 2008), was held as a satellite event of OSS 2008, in Mlian, Italy. In 2009 it was organised again as a satellite event of ETAPS, in York, United Kingdom. Finally, in 2010, the 4th OpenCert workshop took place in Pisa, Italy, as a satellite event of SEFM. The post-proceedings were published in volume 33 of the *Electronic Communications of the EASST*. A special issue collecting the most significative papers of previous workshops is currently under preparation to appear in *Science of Computer Programming*.

This report contains the pre-proceedings of the *5th International Workshop on Foundations and Techniques for Open Source Software Certification* (OpenCert 2011) held on 15 November 2011, in Montevideo, Uruguay, as a satellite event of SEFM'2011, the *9th International Conference on Software Engineering and Formal Methods*. The report is edited by HASLab, the *High-Assurance Software Laboratory* a research centre of Universidade do Minho and INESC TEC. A post-proceedings volume, with revised versions of the accepted papers, will appear later in the *Electronic Communications of the EASST*.

This report includes a total of four full papers and two short contributions, which were selected among ten submissions, each of them reviewed by at least two Program Committee members. It also features the abstracts of two invited talks, one by Antonio Cerone, from UNU-IIST, and another by Ezequiel Bazan Eixarch and Carlos Luna, from Universidad Nacional de Rosario, Argentina, and Universidad de la República, Uruguay, respectively.

The organisers would like to express their gratitude to all members of the Program Committee

for their hard work and support. The result in your hand would not have been possible without their effort and commitment.

The organisers would also like to thank members of the SEFM'2011 Organising Committee, in particular to Alberto Pardo, SEFM Conference Chair, who was most helpful in all occasions, and the staff at the Universidad de la República, Montevideo, for their logistical, administrative and technical support.

Luís Soares Barbosa & Dimitrios Settas
November, 2011

Steering Committee

Bernhard Aichernig, Technical University of Graz, Austria
Antonio Cerone, UNU-IIST, United Nations University, Macau SAR China
Martin Michlmayr, University of Cambridge, UK
David von Oheimb, Siemens Corporate Technology, Germany
José Nuno Oliveira, HASLab / INESC TEC and Universidade do Minho, Portugal

Program Chairs

Luís Dimitrios Settas, UNU-IIST, United Nations University, Macau SAR China

Organization Chairs

Antonio Cerone, UNU-IIST, United Nations University, Macau SAR China
Siraj Shaikh, Coventry University, UK

Program Committee

Luís Barbosa, HASLab / INESC TEC and Universidade do Minho, Portugal
Andrea Capiluppi, University of East London, UK
Francisco Carvalho-Junior, Univ. Federal do Ceará, Brazil
Antonio Cerone, UNU-IIST, United Nations University, Macau SAR China
Ernesto Damiani, Università di Milano, Italy
Roberto Di Cosmo, Université Paris Diderot / INRIA, France
Rafael Dueire Lins, Univ. Federal de Pernambuco, Brazil
George Eleftherakis, CS Department, City College, Thessaloniki, Greece
Elsa Estevez, UNU-IIST, United Nations University, Macau SAR China
Fabrizio Fabbrini, ISTI-CNR, Italy
Andrei Formiga, Univ. Federal da Paraiba, Brazil
Dan Ghica, University of Birmingham, UK
Rene Rydhof Hansen, Aalborg University, Denmark
Mauro Jaskelioff, Universidad Nacional de Rosario, Argentina
Panagiotis Katsaros, Dept. of Informatics, Aristotle Univ. of Thessaloniki, Greece
Tim Kelly, York University, UK
Paddy Krishnan, Bond University, Australia
Paolo Milazzo, Dipartimento di Informatica, Universiti Pisa, Italy
Jose Miranda, MULTICERT, Portugal
John Noll, LERO, Ireland
Alexander K. Petrenko, ISP, Russian Academy of Science, Russian Federation
Alejandro Sanchez, Univ. Nacional de San Luis, Argentina
Dimitrios Settas, UNU-IIST, United Nations University, Macau SAR China
Sulayman K. Sowe, UNU-IAS, Japan

OpenCert 201 Programme

November, 15

09.15 Openning Session
09.30 Invited Talk - **Learning and activity patterns in OSS communities and their Impact on Software Quality** (by *Antonio Cerone*)
10.30 Coffee-break
11.00 **Session 1** (*chair: Antonio Cerone*)
11.00 **Process scenarios in Open Source Software certification**
by *Fabrizio Fabbrini, Mario Fusani and Eda Marchetti*
11.30 **The role of best practices in assessing software quality**
by *Miguel Regedor, Daniela da Cruz and Pedro Henriques*
12.15 Lunch
14.00 **Session 2** (*chair: Luís Barbosa*)
14.00 **Quality, success, communication and contribution in Open Source Software**
by *Sara Fernandes*
14.30 **Analysis of collaboration effectiveness and individuals' contribution in FLOSS communities**
by *Antonio Cerone, Simon Fong and Siraj A. Shaikh*
15.15 Coffee-break
15.30 Invited Talk - **A formal specification of the DNSSEC model** (by *Ezequiel Bazan Eixarch* and *Carlos Luna*)
16.30 **Session 3** (*chair: Siraj Shaikh*)
16.30 **Formal verification of a theory of packages**
by *Jaap Boender*
17.00 **Using antipatterns to improve the quality of Free/Libre/Open Source Software development**
by *Dimitrios Settas and Antonio Cerone*
20:00 *SEFM Welcome Reception*

Learning and activity patterns in OSS communities and their impact on software quality (Invited Talk)

Antonio Cerone[1]

[1] antonio@iist.unu.edu,
United Nations University, International Institute for Software Technology
Macau SAR, China

Abstract: OSS projects can be considered as learning and development environments in which heterogeneous communities get together to exchange knowledge through discussion and put it into practice through actual contributions to software development, revision and testing. OSS communities are open participatory ecosystems in which actors create not only source code but a large variety of resources that include the implicit and explicit definitions of learning processes and the establishment and maintenance of communication and support systems. Productivity, in terms of software development and release, is thus the final act of a very long and complex evolution and growth in the individual and collective knowledge and practise.

Individuals' participation in the OSS community evolves through time. It starts from an initial but often significantly long learning process in which communication is heavily used to capture, describe and understand contents, while no production activity is performed. At a later stage the role of communication gradually moves to the proposal of new contents, the defense of the proposed contents and the criticism to existing contents or contents proposed by others. At the same time, during this stage, production activity starts as a trial and error process with a consequent low quality in the resultant product and little or no immediately visible impact on the project productivity. Only during a mature stage the quality and level of code developed, reports and commits become important and communication is mainly used to support own productive contributions and contrast them to others' contributions.

This talk presents a framework to analyse learning and activity patterns that characterise participation of individuals in OSS communities. Then it defines notions of OSS quality and relate them to individual and community learning and activity patterns. Finally two new proposals to improve the OSS development process and to realise the OSS learning process in an educational context are presented and then related to OSS quality.

Keywords: Open Source Software, activity patterns, learning patterns, software quality.

Process scenarios in Open Source Software certification

Fabrizio Fabbrini[1], **Mario Fusani**[1] **and Eda Marchetti** [2]

[1] (fabrizio.fabbrini, mario.fusani)@isti.cnr.it,
Systems and Software Evaluation Centre, ISTI - CNR Pisa Italy
[2] eda.marchetti@isti.cnr.it, Software Engineering Laboratory, ISTI - CNR Pisa Italy

Abstract: Certification of Open Source Software (OSS) presents inherent trade-offs due to the necessity of precisely identifying both a product and an independent certification agent, and on the other of maintain the peculiar, valuable OSS characteristic of being available to an unlimited multiplicity of actors for trial, use and change. This is an intriguing challenge, usually solved by removing from the picture the certifying agent and providing an intrinsic certification by means of rigorous, reapplicable property demonstrations, adopting Formal Methods (FM) in expressing and verifying the code. As such approach, yet quite valuable and good-promising, has some restrictions (such as the limited set of provable product qualities), we propose to tackle the problem by analysing the various processes executed by different OSS stakeholders, including the process of an independent Certification Body. In the paper some kinds of representative scenarios in which such processes interleave are presented and discussed. The aim is to introduce a process-centered perspective for OSS that can stimulate research to further understand and mitigate the mentioned trade-offs.

Keywords: Open Source Software, Certification, Software Process

1 Introduction

Traditionally, software certification has been mainly associated with proprietary software or Closed Source Software (CSS) with the aim of increasing the confidence that a software-related product or service actually possesses its declared behavioral and/or structural attributes. Recently, the increasing adoption of Open Source Software (OSS) in new environments, such as public administrations, makes it even more urgent to evaluate the correctness and other software quality attributes, such as reliability and usability, of the software used. Indeed, intrinsic product variability, context criticality, compliance to standards and typical constraints of specific domains evidence that certification is still a key factor in adopting OSS software.

Commonly available solutions to this problem try to remove the activity of an independent certifying agent and provide intrinsic certification by means of rigorous demonstrations of software properties by adopting Formal Methods (FM) in expressing such properties and verifying the code against them. However as pointed out by [Wal04], one task is to certify properties of a software item with respect to defined specifications, and another task is to certify related properties of a system (hardware and software) of which the software item is a continuously evolving component. Sometimes the whole system is not available, and also when it is, it is not always easy just to express the ”global” properties of a software component (typically detectable as their impact into external system qualities), let alone to verify them.

Out of the FM, Model Checking techniques are typically adopted to demonstrate the characteristic of *correctness* (with sub-characteristics expressed in the form of provable/non-disprovable properties such as *liveness* and *safety*). Then Model Checking, now rather feasible and intensively automated, can be used for certification [Wal04] (an overview of some proposals is in Section 2). However these approaches, rigorous and good-promising (thanks to availability of many tools)as they may be, cannot be extensively adopted because the set of provable product qualities is still limited and depends on the available specifications. Moreover a recent survey [HSI10] on ongoing OSS projects shows that testing is still almost the only way adopted to check product properties, even if these properties are validated only for the test conditions.

From this considerations our proposal wants to tackle the problem by analyzing the various processes executed by different OSS stakeholders, including an independent Certification Body.

Starting from the assumption that OSS products have interesting, although unconventional, common process features that can be taken into account for certification, we introduce in this paper an approach to a sort of formalized view of the OSS certification activities. Motivated by recent trends in the Italian reality of the Public Administration, that fosters the use of OSS in automation of public offices, we want to investigate different related scenarios, evidencing the roles of the playing actors to find possibly standardizable yet feasible conditions that make an OSS eligible for certification.

We want to focus the reader's attention on the processes performed during the OSS life-cycle to stimulate research towards further understanding and mitigation of the trade-offs between the typical discipline requested by certification and the unconstrained nature of OSS that makes it so appealing and objectively valuable. In this process-centered perspective, we show how the long-dated certification concept, gained with conventional products, can be re-defined and applied to the more complex and varying OSS scenarios. We therefore highlight some of the OSS development process evidences that can be used in a certification process and show that, even inside different scenarios, these processes do have interesting, although unconventional, common features that can be taken into account for product certification.

The paper is organized as follows: In Section 2, a sample of related literature is commented in the light of our objectives and the typical characteristics of the OSS are summarised. In Section 3, the concept of certification is re-visited and new evidences are proposed. In Section 4, significant OSS certification scenarios are presented and commented to point out the role of the main actors, their expected actions and mutual relationships. In Section 5 an analysis of the proposed scenario is provided while Conclusions are drawn in Section 6.

2 Related work

From the vast literature of OSS and certification, we selected some works describing the current trends. As noticed, OSS certification is often related to the use of FM [KM08] in transforming requirements into code and code into requirements, as it is the case of safety and security related properties [CS08, SC09]. Model Checking is a further proposed solution, sometimes considered as the most effective technique for OSS analysis [CGR09]). Alternative proposals are those focused on agile methodologies for keeping consistency between a product and the evidences of the product(for instance [CGR09, MTT09]).

Even though effective, the so far mentioned solutions are limited to specific properties of the OSS software and can be dependent on the specifications.

The advantages of independent certification management is introduced in [PB08], which suggests also mechanisms for doing it (granting/revoking certificates and performing continuous certification in a vulnerable environment) and enforced in [KKS10], where an independent body (in the case, an association of developers and/or users) is expected to provide on-line services to OSS component integrators for which a set of tools, also including reverse engineering tools, search engines and analysers, are being produced.

However, so far the proposals for independent certification have been rather vague and certification against specific standard is reported as a drawback. In our view, certification can advantageously be performed at various levels and an independent entity, the Certification Body, can play different roles in its peculiar task of confidence and transfer it among stakeholders. The scenarios we propose try to figure out the various aspects and roles involved in a certification process and address issues for further research.

3 Peculiarities of the certification of OSS

The literature analysis and the experience in product (CSS) evaluation and process (SPICE) assessment of our Centre [ISO08b] help us to highlight some peculiarities of the OSS that should be taken into consideration for an OSS certification process, typically: availability of usually free or not expensive source; possibility of downloading the OSS from a public website; use of a development environment managed by a community of developers/testers/users, that eases rapid code change and re-use; possibility of measuring product characteristics such as correctness, reliability and maintainability.

From this picture, the main factors influencing a OSS certification process can be summarized as:

- **Stakeholders**: users and developers happen to work closely together and the boundaries among the roles become much more indistinct.
- **Requirements specifications**: In OSS, specifications are not any more controlled by a single organisation and continuously evolve according to the needs of individuals or companies. Often they can be collected from various sources such as developers forums or test cases.
- **Verification and testing**: OSS properties mostly get verified by testing in operational environment, both in case of software component selection, and during actual service. Testing before delivery is then only a fraction of the testing process. Regarding static verification, we noticed in Section 1, about FM techniques, the phenomenon of many proposals in literature and scarcely adopted in practice. Moreover, verification of process documents, which in CSS is one of the best sources of information for a Certification Body, is only marginal in OSS.
- **Independent development**: OSS can be totally or partially cloned by various developers, also concurrently. This may rise new configuration management problems.

- **Traceability of products**: OSS are characterized by high variability and evolution (versioning) of the same product, evidencing some difficulties in identifying the product from its releases.

- **Configuration management**: configuration management life-cycle processes are still there ([OMK08] and [HSI10]), and their actors are generally geographically distributed.

- **Process-related work products**: these are documents such as FAQs, annotations, lessons learnt, bug logs, and so on, which evidence the importance and the interest for OSS inside the community and the development effort behind the completion of the OSS itself. A Certification Body should learn to deal with these work products instead of the traditional CSS documentation.

From the above characteristics it becomes clear that the certification process is expected to continuously track evolving OSS requirements specifications and take evidences from the operational environment. Operational testing [L+96], that in CSS is extremely difficult to manage for cost and time constraints, becomes common practice ([MTT09], [HSI10]) and a precious contribution for improving the overall confidence in the OSS product.

Thus one of the most important role in the assessment part of certification is played by the process-related work products such as blogs, FAQs, annotations, bug logs and so on. That source of information can allow monitoring of product maturity, testing activity and properties implementations and is an essential element of the certification process. In particular, such information aimed to improve or decrease confidence about the product properties can be discovered by verification techniques such as dynamic testing, static analysis and Model Checking. To perform this process-related assessment, traditional document analysis cannot be used and new techniques must be devised, not excluding mining and natural language processing of raw text, as, for example, an evolution of the research proposed in [YSJS07].

Certification process could also be influenced by independent development: versioning of the same OSS and the different abilities of different developers are factors that can affect the final decision of a Certification Body.

These evidences, as well as any other pieces of information derivable by the current available OSI certification [OSI08] have to be monitored in a certification process. In particular, due to the dynamic nature of OSS products, the natural consequence is that certificates are associated to a certain evolution of the OSS, thus related to a specific time and version.

To collect the certified versions of the same OSS products, while the certification is valid, and to reduce the complexity of the communications among the stakeholders, the concept of a *virtual repository* can be introduced. Interactions with a virtual repository can happen only on a voluntary basis for example to get a (possibly dynamic) certificate. This way, registering an OSS project to a virtual certification repository would be appealing and would ease the adoption of the software itself.

The analysis of the influencing factors evidences that independent certification need new standards that do not disrupt the characteristic of free development while allowing an OSS to be eligible for certification. These new standards should recognize collaborative working environments and propose rules to properly collect process evidence and certified products.

In Table Table 1 some differences and similarities between CSS and OSS certification processes are shown. In this partial list, the only significant similarity seems to be CB accreditation.

4 The scenarios context

Lifecycle aspects for OSS certification have been extensively described in literature [Tay09].

In this section we provide an attempt at schematizing two different certification scenarios considering, as mentioned in the Introduction, the context of the Italian Public Administration (PA) environment. As we cannot validate yet our proposal, representing and analyzing possible certification scenarios seems useful to see how the consequences of the factors mentioned in Section 3 can work together. Thus even if specific, we think that these domestic situations, once analyzed, can have many points in common with other international environments and can be easily generalized or adapted to any other context.

In Italy, as in other countries, there is an increasing interest by the central Government in adopting OSS systems for the development of public projects, in particular in the Public Administration context, so to keep costs reasonable and accelerate the completion of administrative system projects. However, due to some criticalities of the context and the many standards and constraints specific to the public environment, only certified products are eligible to be integrated or used in the existing systems. Thus certification is still recognized as a key factor to persuade the various PA offices to adopt OSS solutions. This opens up two possible scenarios:

In the first one the PA itself, to assure that OSS is compliant with the standards and the required level of quality, commissions the certification to an external Certification Body (CB)).

In the second one, it is thinkable that developers can make available ready-to-use, certified OSS to the PA, under the payment of a (perhaps only symbolic) certification fee. In this case even if the developers have to face certification expenses from the CB, they can have incomes from the widespread use of the certified product, as well as from installation and maintenance activity.

We schematize the first scenario in section 4.1 where we consider the PA as Customer/User stakeholder and the certification ordered to an external Certification Body,

The second scenario is presented in section 4.2 where OSS Developer(s) would like to take the advantage pushing in the adoption of the OSS from the PA, by proposing OSS certified products on the market.

4.1 Certification scenario: PA triggering the certification process

In this section we provide details about a possible scenario in which the PA promotes the certification of an OSS product that needs to be integrated in its administrative systems. In particular in Figure 1 we schematize the main stakeholders of the scenario to outline their mutual relationships.

We suppose the PA uses a public OSS repository where developers provide OSS (OSS source artefacts in the figure) implementing different functionalities. The Certification Body (CB) has the role to certify, according to PA requests, conformity to applicable standards and quality constraints involved with the selected OSS and to maintain a common certified OSS repository. With

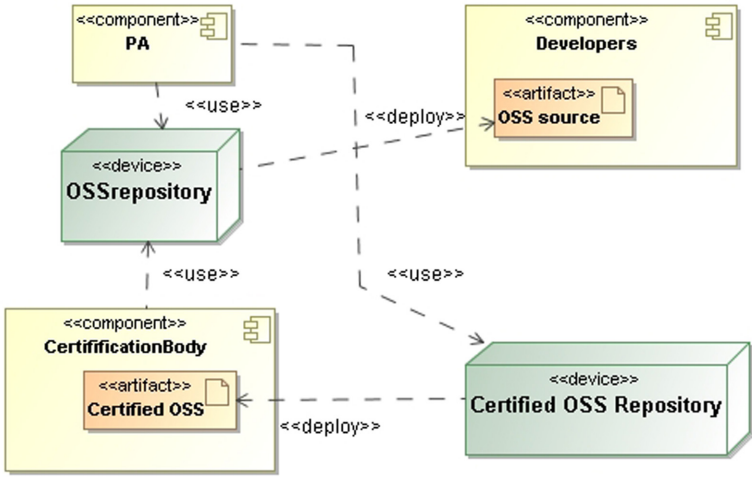

Figure 1: Simplified stakeholders scenario with PA

no expectations to be exhaustive, a possible interaction between the above mentioned stakeholders during the certification process can be schematized as in Figure 2.

Developers and PA, from different points of view, contribute to the requirement elicitation activity, which basically produces a list of constraints that can vary from operational systems specification to development environment constrains, performance and quality attributes and other more specific requests about the functionalities to be implemented. Following the OSS philosophy we consider not restrictive to represent the requirements elicitation as a free, open and not ruled activity, where exigencies coming from different actors and environments join together in common, publicly available (possibly textual) documents.

The various requirements can then be implemented in parallel into OSS product(s) or updated and refined by the PAs. As a common practice, the OSS repository will contain for each OSS product the source code and possibly the OSS storyboard, i.e. all the available source of information concerning the released OSS (logs, comments, description of functionalities and so on). PA can therefore select from the OSS repository the OSS product considered eligible for certification and commission to CB the analyses and the management necessary for the certification itself. CB, following the standard procedure, will perform requirement analysis, possibly recovering data defined during the requirements elicitation, and collect all the available OSS information (as described in section 3). Then the CB can continue the certification activities that, as said in Section3, may require further interaction with the Developers and with independent verification laboratories. The process, whose requirements are listed in Section 5, can produce three principal possible results:

1. CB declares the OSS product not eligible for certification;
2. the certification is successfully concluded, with the certified OSS transferred into the COSS repository and the storyboard opportunely updated;
3. CB identifies required modifications or bugs fixing needs and communicates this to PA

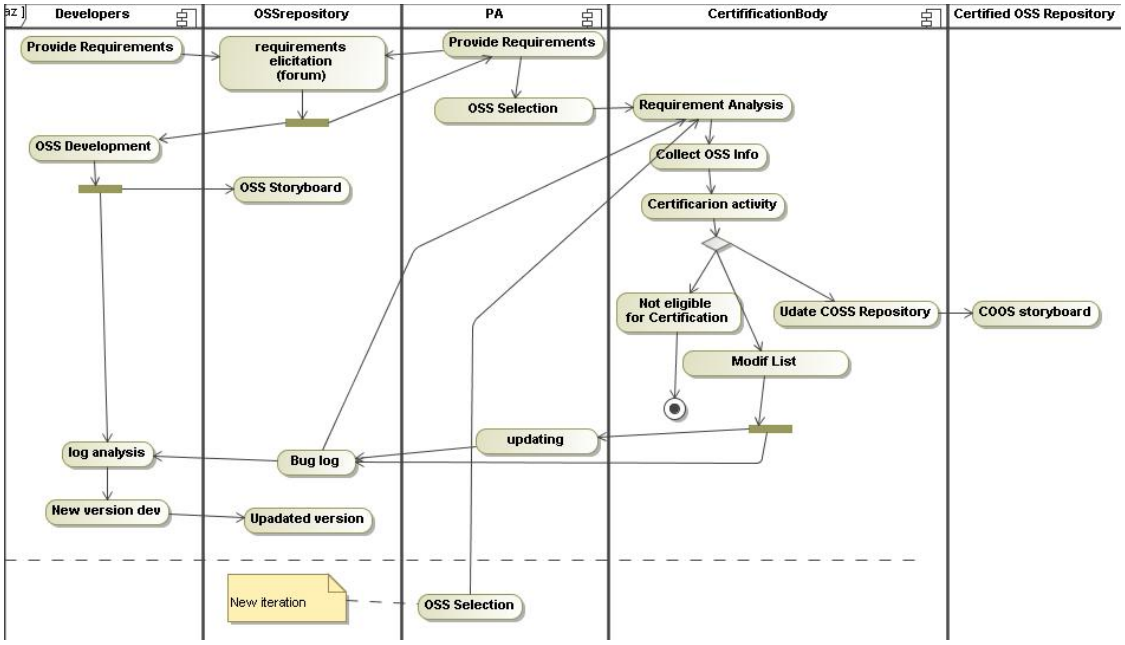

Figure 2: Simplified interaction scenario with PA

that has commissioned the certification. This in turn can update the OSS storyboard and possibly the bug log of the OSS repository so that developers can release an improved version of the required system and let the certification restart for a new iteration

Of course there are variants, still in the perspective of continuous certification ([PB08], [CGR09]): for example, in case of successfully concluded certification, Developers may go on independently updating the software, whose evolved version could draw the interest of the same PA or another public / private institution, which would trigger again the certification process.

From this simple scenario the following considerations can be drawn:

- The stakeholders act rather independently (see Section 3), even if they must synchronize on various occasions. The repository itself, a passive entity, acts as a common channel.
- CB, differently from all the other stakeholders, is bound to behavioral rules. As we already pointed out, it could (and should) be compliant to severe, even conventional standards, but this does not make its presence in an OSS scenario disturbing.

This scenario is a rather high-level one, and its representation hides many intermediate steps that would make the complete description quite a job (Requirements elicitation, Requirements analysis, Certification activity and so on) that can be omitted here.

4.2 Certification scenario: Developer triggering the certification process

In this section we provide details about a scenario in which the promoter for the certification of an OSS product is the Developer. Here the main role of PA is to express the user needs and inter-

est in terms of requirements, while the role of Developers is to implement certifiable products. Certification of OSS against the many standards and constraints specific to the public environment is considered a key factor to convince the various PA offices to adopt already developed OSS. As a side effect, the ability to provide accredited products for the PA could be a means for the developers to increase the number of clients and to make profits (for instance from software installation, maintenance or other related activities).

The stakeholders in this scenario are the same as in the previous one (Figure 1). Thus, again PA uses the public OSS repository for requirements elicitation and Developers are in charge of the OSS implementation (OSS source artefacts in the figure).

On behalf of the Developer, and not of PA as in the previous scenario, CB certifies the OSS according to PA standards and quality constraints, and maintains a common certified OSS repository. It is then possible to suppose that PA can download the certified OSS software from the certified repository upon payment of a special symbolic fee.

We schematize the interactions between Developers, PA and CB during the certification process as shown in Figure 3.

As in the previous scenario, PA contributes to the requirement elicitation activity, which mainly consists of a list of constraints, quality attributes and/or other more specific requests about the functionalities to be implemented. Then the various requirements can be implemented in parallel into OSS products for which the source code and possibly the OSS storyboard, similarly to the previous scenario, are made publicly available. At this step in the process only the developers who want to certify a version of OSS developed product order the certification analysis to the Certification Body.

Note that in this case, CB can use possible information about the development process provided by the Developer itself. Thus, factors that are hardly exploitable in OSS certification, such as, for instance, reference standards, life-cycle traditional standards, architectural requirements, or internal quality characteristics can now be used by CB for certification. As a consequence, the CB activity could be closer to that of a traditional CSS certification process.

Thus, CB, following the standard procedure, will perform a requirement analysis, integrating data about requirements elicitation (if any), available OSS information and additional information provided by Developer. Then CB can continue the certification process, as described in the previous scenario, directly interacting with Developer and possibly with independent verification laboratories.

Just as before, CB activity can produce three principal possible results: 1) CB declares the OSS product not eligible for certification; 2) the certification is successfully concluded, with the certified OSS transferred into the COSS repository and the storyboard opportunely updated; 3) CB identifies required modifications or bugs fixing and communicates this to the developer that has commissioned the certification. Only when the improved version of the required system is developed the certification process will restart for a new iteration.

Once committed in the certified OSS repository, the certified version can be used by Developer for its own marketing.

The stakeholders of this scenario act more independently than in the previous one, so minimizing the points of synchronization and better reflecting the typical features of OSS development. The OSS repository is still independently updated upon request of any developers who ask for certification. For exploitation purposes and also for covering the certification expenses, it is

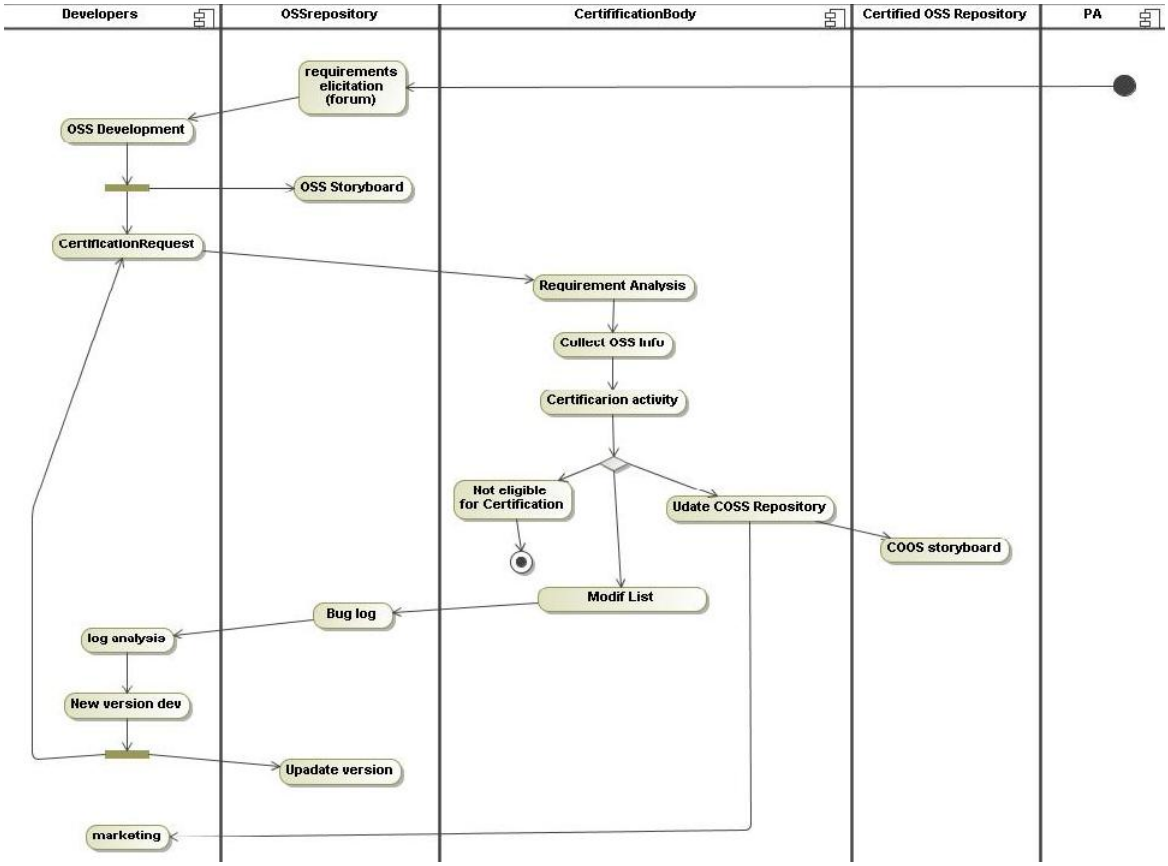

Figure 3: Simplified interaction scenario with Developer

plausible that developers ask each user of the Certified repository a symbolic fee.

The scenarios have been presented to a rather high abstraction level, purposely avoiding intermediate steps that would disturb viewing of the overall picture of the proposal.

5 Comparison and discussion

By describing in this paper only two typical OSS certification scenarios we intended to provide a first tentative to formalize the necessary steps of a the certification process, considering respectively two different triggering situations: one from the PA side and the other from the Developer's. Even if, in the considered scenarios, the overall process shows no development but service only, even a slightly deeper inspection can reveal that process can be composed of various, typically interacting, processes, whose characteristics can be inspired to a process reference model such as in [ISO08a]. In particular: the CB process structure includes lifecycle processes such as management (common to all stakeholders), requirements analysis, verification and testing; Developer process include a more complete set of lifecycle processes, among which coding; PA process may include requirements elicitation (present also in the other stakeholders), prod-

uct acquisition and supplier monitoring. All these processes synchronize with each other, also through the OSS repository, a passive entity that must have its access rules. It is a very high level synchronization with rather weak coupling features, even less compelling in the second scenario, so no deadlock conditions may occur. So, the overall certification process can be expected to be feasible and repeatable, provided it can rely on consolidate best practices. Moreover, it has to be tailorable and able to be monitored even in dynamically evolving situations. Indeed, due to the intrinsic characteristics of OSS development, certification has to face the typical cactus-like versioning of the same product. As a a consequence, an important requirement emerging from both scenarios is that the certification policy should clearly establish the properties to be certified and the validity limits of the certificate itself.

From both scenarios, the need for of an independent Certification Body, able to assess the quality level and the standard compliance of the source code and possibly of other work products with respect to the PA constraints, emerged clearly. As already noticed, the CB is the only stakeholder that has to comply to strict behavioral rules, such as requirements expressed in [ISO04]. The main difference in the situations considered, is represented by the information that CB can use for its activities: In the first scenario (Section 4.1) available data are minimal and mainly represented by the process-related work products; In the second one (Section 4.2) CB could exploit also evidences about the development process provided by the developer itself.

Summarizing, CB should be responsible for the following technical activities, that it might also directly/indirectly perform.

- Assessment and verification of properties declared in a certification scope. Specifically for the scenario presented in Section 4.1, when little or no reference-model exists, such as quality / performance / functional requirements or expected attributes, the role of CB is more exploration than verification: in this case no certificate may be issued, as is not in case the verifications fail (see Figures 2).
- Witnessed or monitored testing. In particular for the first scenario testing is executed by actors different from CB, such as Developers or Users. If testing procedures and reports are standardized and automated, then the monitoring part takes less effort by the CB. This could be true also for the second scenario (Section 4.2), but the strict collaboration from CB and developer could assure more suitable testing info and make the certification process easier.
- Independent testing (e.g., executed by an accredited Independent Laboratory, possibly conformant to ISO/IEC 17025. This is possible in both the scenarios considered.
- Code analysis and inspection (possibly executed by reviewers independent of Developers)
- Model Checking (based on formal models as a source) and Software Model Checking (based on code as a source).

Regarding testing, we already observed that in OSS most of this activity is focused on operational testing (Section 3), that typically can be considered as software validation.

Finally the two scenarios are different also for the purposes of the certification process. In the first scenario (Secton 4.1) certification is a guarantee for the PA that the standards and constraints

specific to the public environment are not compromised or invalidated by the adoption of a OSS software. In the second scenario developers can exploit their certified ability in implementing accredited products for the PA for several beneficial side effects:

- Developers could increase the possible clients because their certified products can be adopted as a ready-to-use and cheap solution in the numerous PA(s) having the same necessities and constraints.
- The certified skill of the developers could be a precious advertisment within the PA communities, and in general for any other OSS community, to increase possible orders for different product development.
- Developers could ask for a symbolic fee for each download from the OSS repository of the Certified OSS. This could refund the developer of the expenses sustained for the certification.
- From the adoption of the OSS certified product, developers could exploit the possibility of opening a new business market due to product diffusion, installation and maintenance activities.

6 Conclusions

In this paper we examined the frequently discussed issue of Open Source Software (OSS) certification. As our Centre has been active in software product verification and process independent assessment since mid 1980's (both as an applied research activity and a service provided to Public Administration and privates), but never in OSS environment, we believe that this perspective could extend the Centre's scope of activity towards a promising business environment.

From the experience gained so far with the Centre, we think we know something about the basic nature of certification for traditional Closed Source (CSS) software, and we expect that the certification concept should be, in some parts, revised to adapt it to the nature of OSS.

In this paper we wanted to observe the process aspects of OSS certification, because of our experience in process engineering and because the process concept includes more knowledge and practice besides the set of techniques adopted.

Generally speaking, a process uses its resources, including technology in the aspects of: appropriateness for the purpose, ability of modeling real situations, ability of providing methods of operation, partial/full automation of such methods by means of appropriated tools (to be integrated into the process), related human factors (role specialization, knowledge, training, skill, motivation, again to be integrated into the same process) and ability of successfully deploying the technology in real projects. Out of these aspects, we addressed here some issues regarding key-roles behavior.

We first re-discussed some aspects of the certification in the light of OSS in terms of reference, standards, techniques, practices, role of stakeholders, examining what factors can positively or negatively impact on the main certification goals (Section 3), then projected the overall certification process into the single processes executed by some significant stakeholders (Developer, Certification Body and Customer in the particular case of Public Administration) (Section 4).

Table 1: Comparison of CSS and OSS certification process

Certification process properties, practices and techniques	**CSS certification process**	**OSS certification process**
Certification process compliant to standards (like ISO/IEC 17000)	Achieved through formal CB accreditation	Achieved through formal CB accreditation
Analysis and verification of life-cycle process documents, different from source code, such as: Requirements, architecture, verification, testing, design reviews,	Quite feasible and opportune for certification	Usually not available
Analysis and verification of life-cycle process documents different from source code such as: field information, forums, blogs,	Usually not provided and if so, unimportant with respect to traditional lifecycle work products	Mostly available and useful. Needs of some general rules and standards to
Analysis and verification of source code	Typically indirect: an assessment of developers' code verification work	Direct code verification using inspection tools and reverse engineering
Close relationship with developers	Feasible and useful	Quite loose relationships available, if any: virtual repository can be an indirect relationship
Continuous certification (multiple versions)	Difficult and expensive	Often necessary
Use of virtual public repository	Often impossible	Quite opportune

We made these processes, together with a passive process ”OSS repository”, interact in a couple of possible significant scenarios, representing them as activity diagrams, and analyzed some pros and cons of the scenarios, trying to summarize a set of requirements for the ”OSS certification process”.

We also pointed out how the process executed by an independent Certification Body, bound by accreditation duty to follow some rigorous standards, can help to mitigate the trade-off between the intrinsic properties of the OSS and the strict rules imposed by certification.

We think this can be another, perhaps untried-before way to have a higher-level view of the possible activities playing around OSS certification, that might be useful for search for more OSS process certification features.

We do not claim to be exhaustive in this formulation, nor particularly innovative. The aim of this paper is to introduce a process-centered perspective for OSS that can help to understand possible different scenarios and to stimulate related research issues.

Bibliography

[BCF+10] I. Biscoglio, A. Coco, M. Fusani, S. Gnesi, G. Trentanni. An Approach to Ambiguity Analysis in Safety-related Standards. In *Proc. of QUATIC 2010 (7th International Conference on the Quality of Information and Communications Technology)*. September 2010.

[Bou10] A. Boulanger. Open-source versus proprietary software: Is one more reliable and secure than the other? *IBM Systems Journal* 44(2):239–248, 2010.

[CGR09] C. Comar, F. Gasperoni, J. Ruiz. OPEN-DO: AN OPEN-SOURCE INITIATIVE FOR THE DEVELOPMENT OF SAFETY-CRITICAL SOFTWARE. 2009.

[CS08] A. Cerone, S. A. Shaikh. Incorporating Formal Methods in the Open Source Software Development Process. In *Proc. of the Third International Workshop on Foundations and Techniques for Open Source Software Certification*. September 2008.

[FFL06] F. Fabbrini, M. Fusani, G. Lami. Basic Concepts of Software Certification. In *Proc. of 1st International Workshop on Software Certification (CERTSOFT'06)*. Pp. 4–16. McMaster University, 2006.

[Fus09] M. Fusani. Examining Software Engineering Requirements in Safety-Related Standards. In *Proc. of DeSSerT (Dependable Systems, Services and Technologies)*. April, 2009.

[HSI10] Z. Hashmi, S. Shaikh, N. Ikram. Methodologies and Tools for OSS: Current State of the Practice. In *Proc. of the Third International Workshop on Foundations and Techniques for Open Source Software Certification*. September 2010.

[ISO96] ISO/IEC. ISO/IEC Guide 2:1996, Standardization and related activities General vocabulary. ISO/IEC, 1996.

[ISO04] ISO/IEC. ISO/IEC 17000: 2004, ISO/IEC 17000:2004, Conformity assessment - Vocabulary and general principles. ISO/IEC, 2004.

[ISO08a] ISO/IEC. ISO/IEC 12207 - Information Technology: Software life cycle processes. ISO/IEC, 2008.

[ISO08b] ISO/IEC. ISO/IEC 15504-5:2006 – Information technology – Process Assessment – Part 5: An exemplar Process Assessment Model. ISO/IEC, 2008.

[ISO08c] ISO/IEC. ISO/IEC TR 15504-6:2008 Information technology – Process assessment – Part 6: An exemplar system life cycle process assessment model. ISO/IEC, 2008.

[KKS10] G. G. Kakarontzas, P. Katsaros, I. Stamelos. Component Certification as a Prerequisite forWidespread OSS Reuse. In *Proc. of the Third International Workshop on Foundations and Techniques for Open Source Software Certification*. September 2010.

[KM08] A. Khoroshilov, V. Mutilin. Formal Methods for Open Source Components Certification. In *Proc. of the Third International Workshop on Foundations and Techniques for Open Source Software Certification*. September 2008.

[L+96] M. R. Lyu et al. Handbook of software reliability engineering. McGraw-Hill New York et al., 1996.

[MTT09] S. Morasca, D. Taibi, D. Tosi. Towards certifying the testing process of Open-Source Software: New challenges or old methodologies?. In *Proc. of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*. Pp. 25–30. IEEE Computer Society, 2009.

[OM09a] A. Ocampo, J. Muench. Rationale modeling for software process evolution. *Software Process. Improvement and Practice* 14(2):85–105, 2009.

[OM09b] A. Ocampo, J. Muench. *Rationale modeling for software process evolution*. Volume 14(2). 2009.

[OMK08] T. Otte, R. Moreton, H. D. Knoell. Applied Quality Assurance Methods under the Open Source Development Model. In *Proc. of the 32nd Annual IEEE International Computer Software and Applications Conference*. Pp. 1247–1252. COMPSAC, 2008.

[OSI08] OSI. OSI Certified Open Source Software. OpenSource.org, 2008.

[PB08] S. Pickin, P. T. Breuer. Open Source Certification. In *Proc. of the Third International Workshop on Foundations and Techniques for Open Source Software Certification*. September 2008.

[SC09] S. A. Shaikh, A. Cerone. Towards a metric for Open Source Software Quality. In *Proc. of the Third International Workshop on Foundations and Techniques for Open Source Software Certification*. September 2009.

[Tay09] R. Taylor. Understanding how OSS Development Models can influence assessment methods. In *Proc. of the Third International Workshop on Foundations and Techniques for Open Source Software Certification*. March 2009.

[Tri02] L. Tripp. Software Certification Debate: Benefits of Certification. *IEEE Computer*, pp. 31–33, June 2002.

[Uni] Unified Modeling Language. http://www.uml.org/.

[Voa00] J. Voas. Developing a Usage-Based Software Certification Process. *IEEE Computer* 33:32–37, 2000.

[Wal03] K. C. Wallnau. Volume III: A Technology for Predictable Assembly from Certifiable Components. Technical report, SEI - CMU, April 2003.

[Wal04] K. C. Wallnau. Software Component Certification: 10 Useful Distinctions. Technical report, SEI - CMU, September 2004.

[YSJS07] K. Youngjoong, P. Sooyong, S. Jungyun, C. Soonhwang. Using classification techniques for informal requirements in the requirements analysis-supporting system. *Inf. Softw. Technol.* 49:1128–1140, November 2007.

The role of best practices in assessing software quality

Miguel Regedor[1], **Daniela da Cruz**[2] **and Pedro Henriques**[3]

[1] miguelregedor@gmail.com
[2] danieladacruz@di.uminho.pt
[3] prh@di.uminho.pt
Dep. de Informática / CCTC
Universidade do Minho
Braga, Portugal

Abstract: Thousands of open source software (OOS) projects are available for collaboration in platforms like Github or Sourceforge. However, like traditional software, OOS projects have different quality levels. The developer, or the end-user, need to know the quality of a given project before starting the collaboration or its usage—they might of course to trust in the package before taking a decision. In the context of OSS, trustability is a much more sensible concern; mainly end-users usually prefer to pay for proprietary software, to feel more confident in the package quality. OSS projects can be assessed like traditional software packages using the well known software metrics. In this paper we want to go further and propose a finer grain process to do such quality analysis, precisely tuned for this unique development environment. As it is known, along the last years, open source communities have created their own standards and *best practices*. Nevertheless, the classic software metrics do not take into account the *best practices* established by the community. We feel that it could be worthwhile to consider this peculiarity as a complementary source of assessment data. Taking Ruby OSS community and projects as framework, this paper discusses the role of *best practices* in measuring software quality.

Keywords: software metrics, static code analysis, open-source, program comprehension

1 Introduction

Nowadays, Open Source Software (OSS) is well disseminated. Thousands of OSS packages can be found online, and free to download, in Open Source Project Hosting Websites (OSPHW) like SourceForge[1], Google Code[2], or GitHub [3]. Those websites, usually in conjunction with a Version Control System (VCS), make it easy for developers, all around the globe, to collaborate in Open Source Software Projects (OSSP), and also act as a way to make software available to users.

[1] http://sourceforge.net/.
[2] http://code.google.com/.
[3] https://github.com/.

According to NetCraft[4], the market share for top servers across the million busiest sites was 66.82% for the open source web server, Apache, much higher than the 16.87% for Microsoft web servers in May 2010. Even governments started noticing open source, during the last few years, and in some case adopted it[Hah02]. The broad acceptance of OSS means that now OSS is not only used by computer specialists.

John Powell[5] has declared that measuring the savings that people are making in license fees, the open-source industry is worth 60 billion dollars. Matt Asay[6] shares the view that from the customers perspective open source can be now considered the largest software industry in the world. The full review can be found at CNET News[7].

Usually large industries have a strict organization model, that is not the way open source communities operates. Open Source communities work in a kind of *bazaar style*. [RE00] compares the traditional software development process to built cathedrals, few specialized individuals working in isolation. While open source development seemed to resemble a great babbling *bazaar*. But OSS is not developed, all the time, in *bazaar style* and each community can have particular habits. Currently, big open source projects can have companies supporting them. However, most projects are not that big and sometimes it is hard to distinguish the project developers from the project customers/users, because of that bug reports and wanted features can get indistinguishable too. The specification of an open source software project evolves in an organic way [CM07].

Can software that is developed in such chaotic way be trusted as a high quality product? The shock is that in fact the *bazaar style* seemed to work [HS02]. Some big projects, for instance Linux distributions such as Ubuntu[8], are the proof of it. However, how can the quality of this software be measured?

The most basic meaning of software quality is commonly recognized as lack of ”bugs”, and the meeting of the functional requirements. But quality is not simply based on that [GKS+07]. The quality of a software system depends, among other things, on update frequency, quantity of documentation, test coverage, number and type of its dependencies and good programming practices. By analysing those parameters a user can make a better choice when picking software for a specific task [MA07].

When a user/developer finds a new OSSP, for example in GitHub, the things that will most influence the time needed to have a better understanding of the project, to use, or collaborate in it, are the quality of the documentation and the source code readability. Although the OSPHWs provide plenty of useful information about the hosted projects, currently, they do not give a quick answer to the following questions: Does this project have good documentation? Does the code follow standards? How similar is it to other projects?

An OSSP is built up from hundreds, sometimes thousands, of files. It can be coded in many different computer languages. To analyze manually a software project is a very hard and time consuming task, and not all users have the ability to answer the previous questions by looking at the source code [CAH03].

[4] http://news.netcraft.com/archives/2010/05/14/may_2010_web_server_survey.html/, accessed on 2010/12/21.
[5] John Powell is CEO, President, and Co-founder, Alfresco Software Inc.
[6] Matt Asay is chief operating officer at Canonical, the company behind the Ubuntu Linux operating system.
[7] http://news.cnet.com/8301-13505_3-9944923-16.html/ accessed on 2010/12/21.
[8] http://www.ubuntu.com. Ubuntu is a free & open source operating system.

However, open source communities are constantly creating and improving their working methodologies. And even without noticing, communities create rules and best practices. By following those *best practices*, software projects increase their maintainability level.

With that in mind, a system capable of analyzing and measuring a given OSSP, producing detailed quantitative and qualitative reports about it, would enable users to make better choices and, of course, developers to further improve the package.

This paper discusses the concept of Quality when addressing an OSSP, and how to measure it (Section 2) using classic approaches. After that, the notion of *best practices* is introduced and the impact of taking their use into account when assessing an OSSPs is explored. To make this proposal clearer and stronger, Ruby[9] community is taken as a starting target (Section 3). At last but not least, to support our proposal a case-study is shown, in Section 4: seven Ruby OSSP are measured and compared.

2 Assessing Open Source Software

The simplest operation in science and the lowest level of measurement is classification [Kan02].

By assessing OSS we mean to sort OSS projects into an ordinal scale[10] This can be achieved by defining a ranking system[11] and by placing OSS projects into quality categories with respect to certain quality attributes. First we need to find a way of quantifying those OSS quality attributes.

In software, quality is an abstract concept. It is commonly recognized as lack of ”bugs”, and the meeting of the functional requirements. However, quality can be perceived and interpreted differently based on the actual context, objectives and interests of each project. Many software development companies do monitor costumer satisfaction as a quality index, for instance, IBM ranks their software products in levels of CUPRIMDSO [Kan02]:

- Capability/Functionality (refers to the software meeting its functional requirements)
- Usability (refers to the required effort to learn, and operate the software)
- Performance/Efficiency (refers to the software performance and resource consumption)
- Reliability (refers to software fault tolerance and recoverability)
- Instalability/Portability (refers to the required effort to install or transfer the software to another environment)
- Maintainability (refers to the required effort to modify the software)
- Documentation/Information (refers to the coverage and accessibility of the software documentation)
- Service (refers to the company monitoring and service)

[9] Ruby is an open source programming language. Ruby community is relatively young but still very focused on following best practices.

[10] Ordinal scale refers to the measurement operations through which the subjects can be compared in order.

[11] Raking system example: to classify a quality attribute, for instance the project documentation, according to its quality with five, four, three, two or one star.

- Overall (refers to an overall classification based on the other attributes)

Almost every big software company have similar quality attributes. ISO/IEC 9126 provides a framework for the evaluation of software quality (The goal is to achieve quality in use, in other words, quality from the user perspective) [Bev99] IISO/IEC 912 defines six software quality attributes:

- Functionality (refers to the software meeting of the functional requirements)
- Reliability (refers to software fault tolerance and recoverability)
- Usability (refers to the required effort to learn, and operate the software)
- Efficiency (refers to the software performance and resource consumption)
- Maintainability (refers to the required effort to modify the software)
- Portability (refers to the required effort to transfer the software to another environment)

Quality attributes have interrelationships. They can be conflictive[12] or support[13] one another. For example, the higher the functional complexity of the software, the harder it becomes to achieve maintainability [Kan02].

Because of the OSP bazaar style and continuous development process, it is intuitive that the maintainability and documentation attributes have a big influence on the overall quality and continuous progress of an OSP (maintainability and documentation have support relationships with usability, reliability and availability attributes, but might be conflictive with functionality and performance attributes).

Failure to meet functionality often leads to late changes and increased costs in the development process. The software industry and researchers have been mostly interested on testing methodologies that focus on functional requirements and pay little attention to non-functional requirements [CP09].

There are several challenges and difficulties, in assessing non-functional quality attributes for software projects. For example, security is a non-functional requirement that needs to be addressed in every software project. Therefore a badly-written software may be functional, but subject to buffer overflow attacks. Another example is the amount of codebase comments, if the code does not have any comments it will not affect the functional requirements, but it is obvious that it will decrease readability and maintainability [GKS+07].

2.1 Classic Software Metrics

To classify OSS with regards to a certain quality attribute, we need to find which factors influence it. Then we need a way to measure that attribute. If we need to measure we need metrics.

Fortunately, there are around two thousand documented software metrics, but there is few information on how those metrics relate to each other. Most of them simply have different names but give similar information [FN99]. The major challenge is to discover how important

[12] Conflictive, negative influence, if one attribute is high it makes the other one low.
[13] Support, positive influence, if one attribute is high it makes the other one high too.

the information given by those metrics is, if the calculation effort pays off, how to interpret their values and find correlations[14] to assess the quality attributes of an OSP.

2.1.1 Lines of Code

A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements in the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements [CDS86].

2.1.2 Cyclomatic Complexity

The measurement of cyclomatic complexity [McC76] was designed to indicate a program's testability and maintainability. It is the classical graph theory cyclomatic number, indicating the number of regions in a graph. As applied to software, it directly measures the number of linearly independent paths through a program source code.

2.1.3 Fan-In and Fan-Out

Fan-in and fan-out are perhaps the most common design structure metrics, which are based on the ideas of coupling [YC79]:

- *Fan-in* is a count of the modules that call a given module
- *Fan-out* is a count of modules that are called by a given module

In general, modules with a large fan-in are relatively small and simple, and are usually located at the lower layers of the design structure. In contrast, modules that are large and complex are likely to have a small fan-in. There is also the theory that high fan-outs represent a high number of method calls and thus are undesirable, while high fan-ins represent a high level of reuse [WLCR07].

2.1.4 Object-Oriented Metrics

Classes and methods are the basic constructs of OO technology. The amount of function provided by an OO software can be estimated based on the number of identified classes and methods or their variants. Therefore, it is natural that the basic OO metrics are related to classes, methods and their size.

The pertinent question therefore is what should the optimum value be for OO metrics. There may not be one correct answer, but based on his experience in OO software development, Lorenz proposed eleven metrics as OO design metrics called rules of thumb [LK94].

- *Average Method Size (LOC)*: Should be less than 8 LOC for Smalltalk and 24 LOC for C++

[14] Correlation is probably the most widely used statistical method to assess relationships among observational data [Kan02].

- *Average Number of Methods per Class*: Should be less than 20. Bigger averages indicate too much responsibility in too few classes.
- *Average Number of Instance Variables per Class*: Should be less than 6. More instance variables indicate that one class is doing more than it should.
- *Class Hierarchy Nesting Level (Depth of Inheritance Tree, DIT)*: Should be less than 6, starting from the framework classes or the root class.
- *Number of Class/Class Relationships in Each Subsystem*: Should be relatively high. This item relates to high cohesion of classes in the same subsystem. If one or more classes in a subsystem don't interact with many of the other classes, they might be better placed in another subsystem.
- *Average Number of Comment Lines (per Method)*: Should be greater than 1.

3 Best Practices in OSSP development

Open source communities have a tendency to create coding standards. It is a natural and evolutive process. Standards are not rules but instead best practices that are spread through the community and everybody does it that way. Furthermore, best practices discourage:

- Poor performance (due to bad patterns)
- Poor error checking (defensive programming)
- Inconsistent exception handling / Maintainability (long-term quality)

When a developer follows the standards and best practices, the project maintainability is increased. Consequently, project new comers will find it easier to understand the project codebase [Dro02].

However, there is little work done concerned with measuring coding standards by automatic analyzing source code. A plausible explanation for that is the fact that best practices are not a set of immutable rules, they are a continuous evolution and improvement of development methodologies. Communities are constantly creating rules and best practices, even without noticing it. It is not possible to write down a list of best practices without some ambiguities. Nevertheless, it is still possible, to use metrics on the source code and, by analyzing their values, to find hints to help answering weather some methodological approaches were taken into account during the project development process.

At first glance, best practices metrics are for classic metrics as natural as medicine is for science. But, it is not the case. In fact, classic metrics, on their own, do not give much information about a project. In many cases, best practices can be the key to understand what should be the optimum value for a classic metric, for instance, how many lines of code should a ruby method have?

Of course, those questions are subjective. However by analyzing renowned projects, developers opinions and so on, it is possible to find the best practice and that gives a plausible answer to the optimum value.

We believe that, actually, best practices can give a meaning to metrics.

3.1 Best Practices in RoR Projects

Ruby is a dynamic, object oriented, open source programming language created by Yukihiro Matsumoto and public released in 1995. It has an elegant syntax that is natural to read and easy to write. Ruby has drawn devoted coders worldwide. In 2006, Ruby achieved mass acceptance. Moreover, the web framework Ruby on Rails is considered the biggest responsible for Ruby popularity (tens of thousands of Rails applications are online).

Ruby and Ruby on Rails community members are, in general, addicted to best practices. However, in reality, many of those best practices are studied development methodologies. For instance, the majority of Ruby on Rails book authors speak about automated tests, written using specific DSLs, like Cucumber or Rspec. It is also common to associate Ruby on Rails with Behaviour Driven Development (BDD) and Agile methodologies.

Because of all this, the ruby community has great potential to be a starting point to understand the role of best practices, its benefits and how to measure it. In fact, there is already some work done.

The web site Rails Best Practices[15], works in similar way to a web forum and its objective is to engage developers to discuss which practices should be considered best practices to follow, when building a RoR web application. The community involved with this web site is committed to build a gem[16] that produces a report about a given project.

3.2 Ruby Best Practices Examples

But what is is a best practice after all? Best practices can be related to code formatting:

- *Use two spaces to indent code and no tabs*, it is a matter of taste but every worthy ruby developer do it that way.
- *Remove trailing whitespace*, trailing whitespace makes noises in version control systems.

Can be related to syntax:

- *Avoid return where not required*.
- *Suppress superfluous parentheses*, when calling methods, but keep them when calling functions(when you use the return value in the same line).

Can be related to naming:

- *Use snake_case for methods*.

[15] http://www.rails-bestpractices.com/ is a web site created by Richard Huang, it was inspired by Wen-Tien Chang talk given at Kungfu RailsConf 2009 in Shanghai. Slides can be found here http://www.slideshare.net/ihower/rails-best-practices.

[16] Ruby Libraries are called gems. Ruby gems can be easily managed using rubygems (rubygems is for Ruby as aptitude is for Debian or cpan for perl).

- *Other method naming conventions*: Use map over collect, find over detect, find_all over select, size over length.

And can also be specific to a framework, rails best practices:

- *Law of Demeter*, A model should only talk to its immediate association.
- *Move code into controller*, according to MVC architecture, there should not be logic codes in view.
- *Isolate seed data*, do not insert seed data during migrations, a rake task[17] can be used instead.
- *Do not use default route*, When using a RESTful design. The default RoR routes can cause a security problems.
- *Replace Complex Creation with Factory Method*, Sometimes you will build a complex model with params, current_user and other logics in controller, but it makes your controller too big, you should move them into model with a factory method.

4 Assessing Ruby on Rails Projects

After deciding that some *procedure* is a *best practice*, it would be handy to find a way to automatically verify whether that practice is being followed by the developers of a given project. With that in mind, an open source ruby gem was created (by the authors of Rails best practices web site) with the objective of automatically producing a report that shows where, in the source code, a project is failing to obey to consensual practices. At the moment of writing, this gem can check for 28 kinds of best practices (from the 70 described in that web site).

However, one of the first things that we have noticed when we have applied this gem to OSS projects, is that the biggest and most renown projects have much more errors than the smaller and unknown projects. This nonsense has a simple interpretation. Small projects (like the majority of RoR projects found in github) are simple software packages, often developed by a single user. These applications are so simple that many times the code is almost entirely created by RoR code generators. Usually, when code is not written by humans, it has few mistakes concerning those recommendations.

4.1 First Study

Having taken the above into account, we decided to run the rails best practices gem on similar RoR (Ruby on Rails) projects. Seven *time tracking* or *project management* open source systems were chosen. After running the gem and counting not best practices (NBPs)[18] occurrences, the following results were obtained:

[17] Rakefiles work in similar way to Makefiles but are written in ruby. It is a simple way to write code to automate repetitive tasks.

[18] In fact, Rails best practices gem does not find best practices in the source code. It does the opposite, it discovers when the code is not written according to a best practice, in other words, it identifies bad practices (similar to the detection of code smells). We decided to name those occurrences NBP.

Rails Best Practices Results							
Best Practice	**A**	**B**	**C**	**D**	**F**	**G**	**H**
Add model virtual attribute	-	2	7	-	-	5	4
Always add db index	-	-	-	43	-	-	51
Isolate seed data	-	-	-	-	-	79	17
Law of demeter	20	38	45	6	30	164	85
Move code into controller	-	-	-	-	2	-	4
Move code into model	-	26	-	7	1	3	19
Move model logic into model	-	-	76	11	11	98	100
Move finder to named_scope	-	4	9	2	4	25	-
Needless deep nesting	-	-	-	1	-	-	-
Not use default root	-	1	1	-	1	1	1
Notes use query attribute	-	2	-	-	-	-	-
Overuse route customizations	-	-	2	4	-	2	2
Remove trailing whitespace	68	57	126	110	330	316	100
Use factory method	-	15	9	5	1	8	19
Replace instance var with local var	13	-	70	239	142	31	100
Use before_filter	-	7	9	8	8	19	23
Wrong email content_type	-	3	-	-	-	-	-
Use query attribute	-	-	11	5	8	29	6
Use say with time in migrations	-	-	24	-	10	23	56
Use scopes access	-	-	-	-	-	-	04
User model association	-	-	12	9	-	1	21
Keep finders on their own model	8	4	1	-	11	-	-
Total	109	156	402	450	559	834	864

A: Rubytime , *B:* Notes , *C:* Tracks , *D:* Handy Ant , *F:* Retrospectiva , *G:* Redmine , *H:* Clockingit
Figures shown represent the number of times a project do not follow a best practice; is expected that *smaller the number, better the project*.

Table 1: Results obtained by running the *best practices analyzer gem* on the 7 Open Source Projects chosen (data produced on April, 2011).

Rubytime seems to have the best results and Clockingit the worst. The fact is that very good user reviews can be found about Rubytime. However, Tracks obtained an unexpected high score, since it has been very sparsely maintained (old code has higher probability of not following the current best practices). As explained before, those values are not really measuring if a project follows best practices but instead measuring when it fails. This should also be taken into consideration.

The most evident problem here is that best practices are not being weighted and neither the size of the project considered. For instance, if the developers have the habit of leaving trailing white spaces, the occurrences of this will obviously be related to the size of the project. On the other hand, it is a best practice to remove the default route generated by rails, independently of the project size this is true or false, there is no way to leave the route two times. So, if developers do not take into account those two best practices, when the project grows, the number of trailing spaces will increase and the results will show more NBPs, but the other one will always be only one NBP. Because of that we can get twisted results.

To avoid this, the projects were sized. The size attribute is based on the quantity of models and controllers in the project. After that, we divided the values previously obtained by the project size. By doing that, a new set of results emerge.

Rails Best Practices Results							
Best Practice	**A**	**B**	**C**	**D**	**F**	**G**	**H**
Total	109	156	402	450	559	834	864
Total Without Trailing Whitespace	41	99	276	340	229	518	764
Project Size	12	11	11	29	26	58	31
Total / Project Size	9	14	37	16	23	15	28
Total Without Trailing Whitespace / Project Size	3	9	25	12	9	9	25

A: Rubytime. , *B:* Notes , *C:* Tracks , *D:* Handy Ant , *F:* Retrospectiva , *G:* Redmine , *H:* Clockingit

Table 2: Results obtained by running the *best practices analyzer gem* on the 7 Open Source Projects chosen, after normalization (data produced on April, 2011).

Those results are much more likely to be helpful in terms of understanding if a project is or is not following best practices. The numbers reflect both the community reviews and our own estimates much more.

4.2 Second Study

After the first study reported above, we felt that it was time to make a bigger one; we should repeat the experiment over a larger sample. In addition, there was the need to define an objective quality metric to compare the metrics results with. As a second target for this new phase, it was decide to find an objective quality rate (a reputation ranking) for each project in the sample, to be possible to compare with the results computed for the best practices metrics.

For the second study, we selected 40 Ruby on Rails projects hosted in github and decided to consider the number of followers[19] and forks[20], that each project has on github, as a *project reputation* metric.

The objective was to prove that a negative correlation exists, between the NBPs of a project and its followers and forks.

The previous study has shown us the need to apply different weights to each NBP. By diving the NBPs by the project size, in the first study, seemed like we got better results. However, not all NBPs depend on the project size. Therefore, we altered the rails best practices gem to make it possible to know how much project files were analyzed by each rails best practice checker.

Basically, after collecting the GitHub URLs for each project, we followed the next steps:

- *Retrieve GitHub information*, in this step we get the followers and forks(and more info that might be used in further analyses).
- *Download the project repository*.
- *Run rails best practices gems*, at this point, we get the non weighted NBPs and files given by each one of the 29 checkers.
- *Calculate the Weighted Global NBPs*, the evaluation algorithm consists in dividing the value returned by each NBP checker by the number of files checked and, then sum it.

[19] Number of users that want to receive notifications about the project.
[20] Number of people that forked the project. This means that either they want to contribute to the project or create a derived project

Next, an excerpt of the obtained table is shown:

Rails Best Practices Results											
Projects	**Forks**	**Watchers**	C1	C1 F.	**W. C1**	C2	C2 F.	**W. C1**	...	T. NBPs	**W. T. NBPs**
Rails Admin	30	2478	0	141	**0**	0	37	**0**	...	50	**739**
Rubytime	12	82	24	161	**149**	0	134	**0**	...	146	**1334**
Redmine	30	1781	49	996	**49**	1	362	**2**	...	884	**1402**
BrowserCMS	30	784	11	234	**47**	0	216	**0**	...	268	**1510**
Tracks	17	87	46	842	**54**	15	271	**55**	...	569	**2810**
...	...	...	...	...	...	...	...	...	...	...	...

C(x): The rails best practices gem has 29 checkers(when this study was carried), each one tries to find occurrences of a different nbp in the project.
C(x) Files: The number of files in the project, where it tried to find nbps (for instance, some checkers may only be concerned with html files, some other checker nbps my only occur in model files, etc)
W. C(x): Weighted C(x) = C(X) / C(x)Files * 1000 (A really small number is added to each variable to avoid divisions by zero).

Table 3: Results obtained by running the *best practices analyzer gem* on the 40 Open Source Projects chosen, from GitHub (data produced on April, 2011). The full table can be found at www.bestpracticesstudy.gorgeouscode.com

4.3 Results

After building a table containing the results for the 40 projects, we easily found correlations between columns. We discovered that the average correlation index, for the weighted C(x) columns, is -0.2. Only three of the weighted C(x) columns do not have negative correlation. This is quite good, considering the fact that there is an explanation for it. Those three checkers (without negative correlation) aimed at finding NBPs that almost non of the projects were committing, so there is no correlation.

The most important results are in the next table:

Correlations		
	Total NBPs	**Total Weighted NBPs**
Forks	0.14	-0.53
Watchers	0.07	-0.40

Table 4: The full table can be found at www.bestpracticesstudy.gorgeouscode.com

These correlation indexes show that if we just count the nbps there is no relation between them and the number of forks and watchers. Nevertheless, the Weighted NBPs have a quite perceptible negative correlation both with watchers and forks.

Observing that Table, it is possible to notice that the forks correlation is bigger. We believe that if it happens, it is because forking a project shows intensions of digging into the code and, of course, it easier to understand others code when it follows good practices.

As future work, we are considering more correlations with other variables that are already available, but we haven't used yet. The most relevant ones: the number of commiters, starting date of the project, last commit data, and total number of commits. We believe that those vari-

ables can strongly be related with the forks, watchers and of course, in the end, the quality of the project.

5 Conclusion

Nowadays, thousands of open-source software packages can be found and freely downloaded online. github is a web-based hosting service for projects that use the Git revision control system. It hosts more than 1 million open-source projects.

There is little work done concerning the measurement of coding standards by automatic analyzing source code. We strongly believe that some research and development should be done in this direction. Along the paper we gave arguments in order to make evident that it is worthwhile to detect on the source code that the author follows the best practices recommended by the respective community.

In this particular context, Ruby Community, there is already some work done. The reports generated by the existent source code analyzers, can spot the occurrences of bad smells but this is not enough. There is the need to interpret those results to end up with a high level quality statement. By comparing some projects, it was possible to start understanding how to interpret those values. For instance, the *size of the project* should be taken into consideration (it is intuitive that a project with 10 lines of code and 10 errors is worse than a project with 1000 and 20 errors). From the study we carried out and, described in the paper, we also have learned that each best practice has a different importance level — 1 error that affects security or performance is, for sure, worse than 10 errors related to indentation; or 10 errors related to naming conventions are worse than the indentation mistakes).

We do believe that by analyzing a massive amount of open source projects, it is possible to create a new set of metrics capable of quantify the standards followed by a given project, judge the impact of the metrics evaluated and consequently assess its level of maintainability.

The future work is to develop a system capable of automatically produce quality reports about a given OSSP combining traditional SW metrics with best practices analysis.

This system will enable users to make better choices about what software to use and help developers to improve their software.

Bibliography

[Bev99] N. Bevan. Quality in use: meeting user needs for quality. *Journal of Systems and Software* 49(1):89–96, 1999.

[CAH03] *Defining open source software project success*. 2003.

[CDS86] S. Conte, H. Dunsmore, Y. Shen. *Software engineering metrics and models*. 1986.

[CM07] A. Capiluppi, M. Michlmayr. From the Cathedral to the Bazaar: An Empirical Study of the Lifecycle of Volunteer Community Projects. *INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING -PUBLICATIONS- IFIP* 234/2007:31–44, 2007.

[CP09] L. Chung, J. do Prado Leite. On non-functional requirements in software engineering. *Conceptual Modeling: Foundations and Applications*, pp. 363–379, 2009.

[Dro02] R. Dromey. A model for software product quality. *Software Engineering, IEEE Transactions on* 21(2):146–162, 2002.

[FN99] N. Fenton, M. Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software* 47(2-3):149–157, 1999.

[GKS+07] *Software quality assessment of open source software*. Athens University of Economics and Business, Patission 76, Athens, Greece, 2007.

[Hah02] R. Hahn. *Government policy toward open source software*. Brookings Institution Press, Washington, DC, USA, 2002.

[HS02] *High quality and open source software practices*. 2002.

[Kan02] S. Kan. *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

[LK94] M. Lorenz, J. Kidd. *Object-oriented software metrics: a practical guide*. 1994.

[MA07] A. Marchenko, P. Abrahamsson. Predicting software defect density: a case study on automated static code analysis. *Agile Processes in Software Engineering and Extreme Programming* 4536/2007:137–140, 2007.

[McC76] T. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, pp. 308–320, 1976.

[RE00] E. Raymond, T. Enterprises. The Cathedral and the Bazaar. *KNOWLEDGE, TECHNOLOGY AND POLICY* 12:1–35, 2000.

[WLCR07] Y. Wang, Q. Li, P. Chen, C. Ren. Dynamic fan-in and fan-out metrics for program comprehension. *Journal of Shanghai University (English Edition)* 11(5):474–479, 2007.

[YC79] E. Yourdon, L. Constantine. Structured design. Fundamentals of a discipline of computer program and systems design. 1979.

Quality, success, communication and contribution in Open Source Software

Sara Fernandes0

[1]sarasantos.fernandes@gmail.com
HASLab / INESC TEC , University of Minho

Abstract: Free and open source software projects are often perceived to be of high quality. To a great extent the success of open source software seems to be due to an implicit but effective connection between communication and contribution in its development process. In this paper, we present a snapshot of the state the art on quality and success of Open Source Software (OSS) based on a review of the literature. For each of these concepts, we describe various measures considered in the literature and a number of methods by which they are obtained. Contributions to an Open Source Software (OSS) project are made through communication among developers and users. We elaborate on the concrete notions of communication and contribution in Open Source Software (OSS) and their links.

Keywords: Open Source Software, Quality, Success, Communication, Contribution

1 Introduction

To study the properties of Open Source Software (OSS), we need to consider this phenomenon as a process, both technical and social, not just its end product. The development of open source software is inseparable from the communities that engage in its development. This suggests that a suitable model to study the properties of OSS may be based on the models for Information Systems (IS). Silver et al. defined IS as follows: “Information systems are implemented within an organization for the purpose of improving the effectiveness and efficiency of that organization. Capabilities of the information system and characteristics of the organization, its work systems, its people, and its development and implementation methodologies together determine the extent to which that purpose is achieved”.

Open source software is very often developed in a public, collaborative manner. The qualifier “collaborative” designates a style of software development whose focus is on public availability and communication, mainly via the internet. The collaborative aspect of OSS development is similar to the one used on an IS.

An IS is any combination of information technology and people’s activities using that technology to support operations, management, and decision making. This combination is collaborative and somehow close to the one used on OSS development.

Regarding OSS development as an IS allows us to apply the existing models of information systems success (DeLone and McLean Model) to predict potential success measures for OSS projects. However, OSS projects and IS also differ in significant ways. This makes a number of the predictive measures for the success of IS unsuitable for OSS, while others are difficult to apply in OSS environments.

Many of the predictive measures for the success of IS are based on a vision of system development in an organization that is at the core of an IS. This however does not take into account the unique characteristics of the OSS development environment, embodied in OSS communities. Free open source projects are carried out by volunteers that are members of a OSS community. Due to the free commitment to contribute to the development of any OSS product, the contributions of the volunteers aren't always reliable since volunteers may stop carrying out their duties for a long period of time or, in some cases, never return to them. Also the distributed characteristic of the OSS project makes it impossible to gather all the volunteers at the same time[Mic04].

In this paper, we propose to apply a version of DeLone's model, modified to accommodate what distinguishes OSS projects from IS, to predict the success of OSS projects.

Despite of the similarities and differences between OSS projects and IS, OSS projects can be seen as organizations. It is necessary to think of an OSS community as an organization, to make it possible to apply the measures of success of IS in the model of DeLone and McLean to OSS projects, taking into consideration the fact that there can be more categories/dimensions than the ones presented in the original model. OSS can be seen as an organization but it emerges organically bottom up, as opposed to the top down created-by-design structures of traditional organization where information systems have been studied. Because they are organic and self-organizing, and rely on motivated volunteers, some measures and factors that are relevant for their categorization are different than those for the case of traditional IS.

Open Source Software (OSS) projects are often thought of having the unique characteristic of being developed by volunteers and are perceived to have higher quality. Much attention has been paid to the implications of development by motivated volunteers. In this paper we clarify the concepts of quality, success, contribution, and identify a number of characteristics of communication in OSS projects. We also identify measures that can be applied to assess the quality and success of an OSS product and the distribution of contribution and communication on the development of high quality and successful OSS projects and products. Our goal is to consider different measures of these four properties and to establish relationships between them.

The remainder of this paper is organized as follow. In Section 1 we present the research context. In Section 2, we briefly review the literature on relevant measures for software quality and success in OSS products. Section 3, reviews the literature on communication and contribution in OSS projects, and describes the properties used in the literature to measure communication and contribution. In Section 4, we discuss possible links between the concepts presented in Sections 2 and 3, resorting to the DeLone and McLean Model of IS Success. Based only on the literature, this comparison suggests additional measures that may be incorporated in the development of successful and quality OSS. We conclude in Section 5, by suggesting directions for future research.

2 Research Context

Open Source Software (OSS) plays an increasingly important role in our society. Consequently, it is necessary for OSS to have and maintain a suitable level of quality to justify third-party confidence on its products. Both of these factors (quality and success) are influenced by the fact that OSS allows users and developers to have access to the source code that is developed under

some sort of ”open source” license. There are different forms of licenses[UPO11] for OSS that grant substantially different rights to their users. However, we will not discuss the OSS licensees in this paper. In order to have OSS (and/or Free Software)[Per04] that is not only successful but also has quality and impact in the society, it is necessary to understand not only how success and quality can be measured, but also the physical distribution of contributors/developers, the distribution of the contributions made by the developers and users, and certain characteristics of communication that takes place in the OSS development process.

One of the most quoted papers on the phenomena of OSS, is ”The Cathedral and the Bazaar” [Ray99] by Eric Raymond. The paper claims that the OSS process is marked by volunteerism, collaboration, and a lack of project structure. It recalls that the development progress depends on the developers free time and because of it, there is no release schedule. Often, there is only an outline of features. And in the limit, the end user ensures quality. Obviously, many OSS projects involve not only volunteers; big companies put their paid employees to work on OSS as well. Companies such as IBM and Sun Microsystems have entire divisions devoted solely to Open Source Projects. Also, many serious OSS projects do have release schedules, testing, etc., often adhered to even more strictly than in many non-OSS projects. Although this model characterizes the nature of an OSS community and OSS project development, the increasing role of OSS in our society requires more. It is necessary to take into consideration that OSS is a form of system development and to find ways that measure quality, success, contribution and also the impact of communication between developers and users.

Since OSS development can be defined as a form of information system development, the study of the success measures of an IS becomes relevant.

Our starting point is the fact of OSS being somehow more like IS than traditional software development. Due to the looser structure of OSS projects, the roles of people and the OSS community are very important. An OSS project can be seen as an organization, that emerges organically bottom up, as opposed to the top down created-by-design structures of traditional organizations where in information systems have been studied. It can also be seen as an open democratic society, were the role of the individuals and the role of the community are very significant in contrast to the pre-established structures in the traditional software development projects. The similarity of OSS projects to IS suggests that we may use the success measures proposed for IS to predict the success of OSS projects.

One of the most cited models of success in IS, shown in Figure 1, is the one developed by DeLone and McLean in [DM92], [CAH03].

The model in Figure 1, provides a schema for clarifying a multitude of measures into six categories, or dimensions, suggesting a model of temporal and causal interdependencies between these categories. DeLone and McLean conclude their paper with the comment that the model in Figure 1 ‘clearly needs further development and validation before it could serve as a basis for the selection of appropriate IS success measures”.

The model can be interpreted as follows:

Systems quality and *information quality* singularly and jointly affect both *use* and *user satisfaction*. Additionally, the amount of *use* can affect the degree of *user satisfaction* positively or negatively – as well as the other way around. *Use* and *user satisfaction* are direct antecedents of *individual impact*; and lastly this impact on individual performance will eventually have some *organizational impact* (DeLone and McLean, 1992: 83-87).

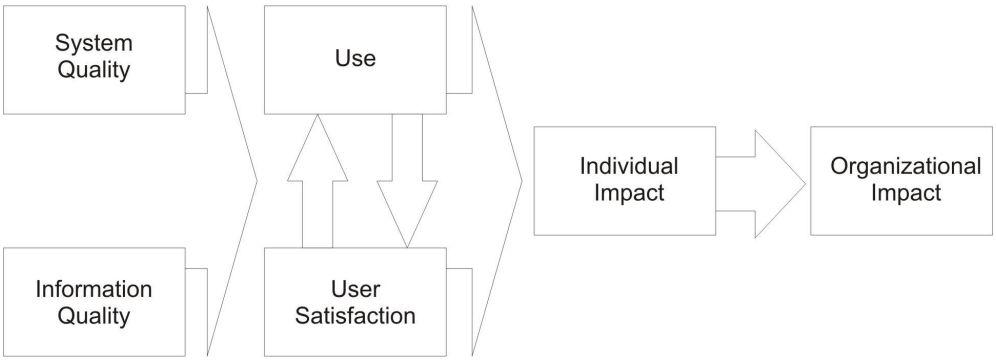

Figure 1: DeLone and McLean's Model of IS Success

But IS is any combination of information technology and people's activities. In traditional software projects, the structures and goals, budgets, schedules, etc., are set more strictly before hand. Thus, aside from the technical competence of the people involved, the "people" involved do not matter that much. The structures do not leave much room for their individuality to change the process or the outcome. In OSS projects, all structures are much looser. Therefore, the characteristics of the individuals, their personal motivations, personalities, goals, etc., play a much more important role in the development process. Actually, we believe that the emphasis of IS on people and their interactions with the software is their contact point with OSS projects. OSS is free, is open source, is built by volunteers and it allows heterogeneous teams. On the other hand traditional software must be paid, its source is closed, is built by the company employees and has a stable development team. Also, and as in IS, an OSS community needs to coordinate development: for example, there is the need for management to know wether each volunteer knows what to do and all this requires decision making.

Considering an OSS project as an IS with particular characteristics, we believe that the DeLone and McLean model fits our purpose to predict the success of OSS projects.

3 Measures for Quality and Success in OSS

Abandoned projects may find an explanation in the volunteers that find more interesting projects but also the successful projects face important problems related to quality due to the not always reliable contribution of volunteers [Mic04]. Also, it must be taken into consideration that OSS communities seek newly and this fact also reflects on the web hosting services. Note that it is possible to find abandoned projects on SourceForge that are active on GitHub or other web hosting.

Many quality problems have been identified in interviews performed in free software communities [VMT07]. Analysis of web hosting for OSS, such as GitHub or SourceForge also provide useful insight. SourceForge is one of the most popular hosting sites for free software with over 313,000 projects. It is not only a good resource to find well maintained projects, but also to find out a large number of abandoned projects and low quality code.

The most common problems identified are the *unsupported code*, *configuration management*,

security updates and *users* that do not know how to report bugs.

- *unsupported code* affects quality. Note that it is impossible to know how to handle code that has previously been contributed but it is now unmaintained;
- *configuration management* affects quality depending on the number of contributions. It is difficult for the lead developer to test all contributions;
- *security updates* affects quality since, although some bugs are fixed there isn't any schedule to make the fixes available;
- *users* who sometimes do not know how to report a bug and often fail to include enough information in bug reports;

All these problems need to be faced and it is necessary to come up with a range of measures that can attest the quality of an OSS product.

Some say that the quality of OSS can be measured by the number of users and the code quality. Actually, in the book "The Cathedral and the Bazaar" [Ray99], Raymond suggests that the level of quality is partly due to the high degree of peer review and user involvement often found in free software projects.

Code quality has been studied extensively in software engineering. Literature provides many possible measures of the quality of software. This includes understandability, completeness, conciseness, portability, maintainability, testability, usability, reliability, structuredness and efficiency, etc (Boehm et al. 1976; Gorton and Liu 2002). Code quality measures seem particularly applicable on OSS, since source code is available by definition. Some authors have already studied this dimension, Stamelos et al (2002) believe that OSS is, in general, of good quality. Mishra (2002) offers an analytic model that suggests factors contributing to OSS quality.

Based only on the literature review, we present what we consider the most relevant measures for OSS success[CAH03]: *User*, *Product*, *Development Process*, *Developers*, *Use*, *Recognition* and *Influence*.

- *user* who have the possibility to change the software to satisfy her needs. Moreover, some users feel involved in the development process through mailing lists or bug reports or other forms of communication with the development team. However, sometimes inconsistency is observed in open source software mainly due to "short feedback loops between users and core developers" [RCA11], [PYM+06]. Users feel that frequent beta versions of the software frustrates some of their expectations. However, the satisfaction of users / clients increases while they feel involved in the development of new releases.
- *product*, whose key success issues are the degree of meeting requirements of design, the code structure organized, clear and maintainable documentation. These aspects are also related to the code quality. A final product with quality has a higher probability to achieve success. Also code quality may be measured by its portability and compatibility with other systems and programs, as mentioned previously. Moreover, if an OSS product is available from more then one source, then the probability of reaching a bigger number of users increase;

- *process* which is complex and requires a high level of activity in the form of fixing bugs, writing, updates, etc. Developers must always work having in mind the goals of the project. Also, after a beta release, it is important to keep track of all bugs and fix them in a reasonable time. Moreover, the level of activity of a product is also a measure of the time the group has been active;

- *developers*, that are the main contributors of OSS. The OSS community of developers has the particular aspect of sharing their developers among several projects. This give each project more opportunities to succeed due to the different background and different expertise behind the contributions; just like users, developers also need to feel satisfaction for innovating. This also gives them the opportunity to enjoy the process of development;

- *use*. Ultimately, success can be measured by the effective use of a certain OSS product. The usage can be measured by the number of users, besides the developers, that use an OSS product and the number of downloads it gets;

- *recognition*, by the peers is also a way to measure success. If other projects or organizations pay attention and recognize the importance of a product then it is more likely that they will refer to it at some point;

- *influence*. Since OSS code is free, the degree of code re-use in other projects can be considered as a measure of success.

4 Communication and Contribution in OSS

It is clear that the OSS phenomena is mainly driven by the developers communities. Although OSS projects differ amongst themselves, for example, in their size, also the communities differ in levels participation in coding and communication. Actually, communication and contribution also depend on having suitable strategies. These strategies have a number of dependencies that can be measured.

We believe that communication and contribution play an important role in OSS development.

Researches who have studied the OSS phenomena and OSS projects have largely focused their attentions on contributions of code and the overall contribution itself. However, in addition to contribution, we believe that also communication plays a relevant role during OSS development. Communication in OSS also contributes to knowledge, sharing and collective innovation.

As mentioned on Section 2, OSS proponents have argued that the strengths of the OSS phenomenon are associated with the bazaar model of full participation from large numbers of developers and users [Ray99]. This argument is based on the rationale that a highly participative community may lead to richer discussions, better flow of ideas, efficient code development, faster bug finding and fixing and, hence, faster and efficient project growth [A.C98], [Ray99], [VMT07]. However, empirical studies have found that the numbers of contributors / developers by themselves are not a sufficient factor to sustain the bazaar view of OSS projects. A.Cox [A.C98], a senior developer of Linux, argues that “the bazaar model may turn into a “town council” where equally participative community produces only noise and hence no output” damaging

communication. He further claims that “such a bazaar - town council model may slow down the evolution of a project...”.

These different points of view raise the importance of communication as well as contribution in OSS development. The impact of non- existing communication and chaotic contribution may destroy a functional democratic OSS community and create an anarchic one compromising OSS success and quality. Existing literature suggests measures for communication and contribution. As far as communication is concerned *stable releases*, *participation*, *interactivity* and *release strategy* are considered measures.

- *Software stable releases* are considered a communication measure even if some authors hold that an OSS project never leaves the Beta state. Stable releases may be considered a measure of communication since new releases typically bring in new users, i.e., the number of downloads should increase compared to previous releases. Also, the release notes improve communication between developers and users, which helps to boost the testing function shared by users;
- *Participation* in developing, testing and use is also an obvious measure;
- *Communication interactivity* is expected to have a positive impact on OSS development. It is believed that a good, interactive communication between developers leads to successful and quality products;
- *Release strategy*, under the typical OSS slogan “release often and release fast”. The advantage of using this mechanism is to provide the necessary feedback to understand if the approach followed in a project is the best.

For contribution, the existing literature suggests the following measures:

- *Development time*, i.e., the length of time between the inception date of the project and the release date of the identified stable release;
- *Code contribution*, taking into account that there are different ways to contribute to an OSS project, and that this should somehow be recorded;
- *Number of developers*, although there is no agreement if the number of developers is a measure of real contribution and successful communication in an OSS project, we will consider it as a measure;
- *Project size*, which may determine the effectiveness of an OSS project. The smaller the project the easier it is to control the constant changes that occur during its development.

5 Quality, Success, Communication and Contribution on OSS

We began by using the information system model of DeLone and McLean[DM92]. Through this model, we present the relationship between quality and success. However, this model is not clear enough to further explore the effects of contribution and communication and their impact on quality and success.

Although some authors, as in "The cathedral and The Bazaar" emphasize the importance of the number of contributors, one cannot analyze only the numbers of users and developers to say that an OSS project has quality and is successful. There is a lack of empirical data to sustain this statement. On the other hand, appearance of these statements in the literature suggests a strong relationship between quality and success in OSS projects.

But, how to link Quality, Success, Contribution and Communication? Do we need another model or is it enough to extend an existing one?

It turns out that, having measures for the four concepts defined, makes it possible to determine a relationship between quality, contribution, communication and success by extending the DeLone and McLean model to include contributors and communication. In section 2, quality and success were presented together because many believe that quality by itself entails a successful product. This is true but also the communication and contribution aspects must be considered. An OSS can be successful, have relatively good quality but, if it lacks the communication part then it will not be as successful as wished. Actually, success is not only measured by the number of contributors that take part in the development team. It is necessary for the developers to be motivated and develop OSS collaboratively.

Also in section 3, communication and contribution were presented together since, it is our belief that it is impossible to have quality contribution without communication between contributors, and in the limit, users.

Communication during OSS development has a fundamental importance and so, we believe the model proposed by DeLone and McLean[DM92] can be extended to deal explicitly with communication and contribution:

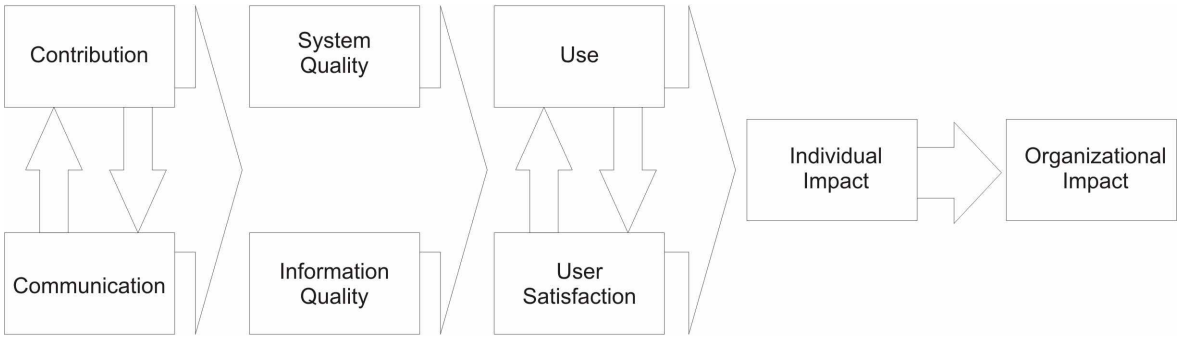

Figure 2: Extended Information System Model

Figure 2 presents an extended version of the DeLone and McLean model which highlights the relevance of communication and contribution together for OSS development. Figure 2 aims to clarify the importance of communication to the contributors in order to develop quality software and achieve success. The 8 dimensions in Figure 2 are interrelated, as it happens in the DeLone and McLean model, wherein the 6 categories of success are proposed as interrelated rather than independent dimensions.

6 Conclusion and Future Work

Open source software projects demonstrate a number of similarities with traditional information systems. However some of their characteristics significantly distinguish them from typical information systems as traditionally understood. The similarities between OSS projects and information systems motivate researching the possibilities of extending IS theories to OSS. This paper makes one such attempt to apply and extend the IS success model of DeLone and McLean to evaluate the success of OSS projects. We presented the concepts and measures for quality, success, contribution, and the characteristics of communication, defining a set of measures that can be used in the analysis of quality and success of OSS projects, augmented with measures for communication and contribution. We also presented an extended version of the DeLone and McLean model in order to make clear the relevance of contribution and communication to achieve quality and success in OSS projects.

DeLone and McLean conclude their paper commenting on the model in Figure 1 as follows: *"clearly it needs further development and validation before it could serve as a basis for the selection of appropriate Information System success measures"*. Our extended model for OSS projects clearly needs further development and validation before it can serve as a basis for the selection of the appropriate parameters in OSS projects. Nevertheless, we believe that the prospect of having a model to predict the success and impact of OSS projects has interesting practical implications. For instance, one can use such a model to assess the potential success of an ongoing OSS project, and subsequently adjust the structure, organization, participants, contributions, etc., that affect the parameters of the model deliberately, in order to increase the success measure of the project, before it is too late. As future work, we intend to further develop and validate this model.

Bibliography

[A.C98] A.Cox. Cathedrals, Bazaars and the Town Council. 1998. http://slashdot.org/.

[CAH03] K. Crowston, H. Annabi, J. Howison. Defining Open Source Software Project Success. In *in Proceedings of the 24th International Conference on Information Systems (ICIS 2003*. Pp. 327–340. 2003.

[DM92] W. Delone, E. McLean. Information Systems Success: The Quest for the Dependent Variable. *Information Systems Research* 3(1):60–95, 1992.

[Mic04] M. Michlmayr. Managing Volunteer Activity in Free Software Projects. In *Proceedings of the 2004 USENIX Annual Technical Conference, FREENIX Track*. Pp. 93–102. Boston, USA, 2004.

[Per04] Perry Donham. Ten Rules for Evaluating Open Source Software. 2004. http://collaborative.ws.

[PYM+06] A. Porter, C. Yilmaz, A. M. Memon, A. S. Krishna, D. C. Schmidt, A. Gokhale. Techniques and processes for improving the quality and performance of open-source software. *Software Process: Improvement and Practice* 11(6):163–176, May 2006.

[Ray99] E. S. Raymond. *The Cathedral and the Bazaar*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1999.

[RCA11] A. Raza, L. Capretz, F. Ahmed. Users' perception of open source usability: an empirical study. *Engineering with Computers*, pp. 1–13, May 2011.
doi:10.1007/s00366-011-0222-1
http://dx.doi.org/10.1007/s00366-011-0222-1

[UPO11] UPOSU. User's perception of open source usability: an empirical study. 2011. http://eng.uwo.ca.

[VMT07] P. Vir, S. Ming, F. Y. Tan. An Empirical Investigation of Code Contribution, Communication Participation, and Release Strategy in Open Source Software Development: A Conditional Hazard Model Approach. 2007.

Analysis of collaboration effectiveness and individuals' contribution in FLOSS communities

Antonio Cerone[1], **Simon Fong**[2] **and Siraj A. Shaikh**[3]

[1] antonio@iist.unu.edu,
United Nations University, International Institute for Software Technology
[2] ccfong@umac.mo
Department of Computer and Information Science, University of Macau
[3] siraj.shaikh@gmail.com
Department of Computing, Coventry University

Abstract: Free/Libre Open Source Software (FLOSS) development has proven itself over the years to be able to deliver high-quality software products. However, it is not clear how quality emerges from the large amount of loosely organised activities of a FLOSS community. This makes it difficult to apply traditional quality metrics and certification processes to FLOSS products.

This paper investigates possible indicators of collaboration effectiveness and quality of individuals' contribution that could be extracted from the data available in repositories of FLOSS projects. The ultimate purpose of this effort is to develop quantitative metrics for these indicators and merge such metrics into a global metric for FLOSS software quality to be used in a certification process.

Keywords: Open Source Software, software quality, collaboration models, trust models, certification.

1 Introduction

Although the high-quality of a number of Free/Libre Open Source Software (FLOSS) products has been accepted as a fact, it is still unclear how such high-quality emerges from the "bazaar-style" activitis of a FLOSS community. Large communities of users have been growing around popular high-quality FLOSS products such as Linux, Ubuntu, Apache, MySQL and Moodle, among the others. The widespread use of FLOSS products not only involves personal users, who install Linux/Ubuntu and MySQL on their machines, but also small and medium enterprises, who use Apache servers and FLOSS tools in their production activities or even incorporate FLOSS components in their software products, and academic and teaching institutions, who use FLOSS products in their research and educational activities, including Learning Management Systems (LMS) such as Moodle. More recently, large software companies have been launching FLOSS projects with the aim to get revenues by adopting a freemium business model, in which the basic product or service is provided free of charges, while a premium is charged for the provision of support services and/or advanced features and functionality.

The only issue that still limits the diffusion of FLOSS products is the lack of certification process. This is actually a major limitation, since most software systems that we encounter in our

daily life include safety-critical or security components, which require the approval of a certification authority in order to be used. However, the lack of accurate information on how quality emerges from the large amount of loosely organised activities of a FLOSS community makes it difficult to apply traditional quality metrics and certification processes to FLOSS products. For instance, if we consider McCall's production revision quality factors [MRW77], can we claim that a FLOSS product lacks *maintenability* because there are no defined coding standards and guidelines to which programming has adhered? The philosophy of freedom and absence of hierarchical organisation typical of FLOSS communities results in collaborative production environments in which there is no space for *prescriptive* standards and strict guidelines. Communication and collaboration are the drivers of such production environments and naturally determine the evolution of programming practices within teams of contributors and across the FLOSS community, even beyond a specific FLOSS project. In such a context a *descriptive* approach that analyses the FLOSS community of practise and its activities is likely to define better indicators of the quality of the software product than a *prescriptive* approach that tries to check whether these activities follow prescribed standards and guidelines.

Ih this paper, we carry out a preliminary discussion towards a methodology to analyse community activities in FLOSS projects and extract from the data collected pieces of semantic information to be used as indicators of the quality of the software product. In particular, we focus on two aspects of the FLOSS community of practice: collaboration effectiveness and quality of individuals' contribution.

2 Collaboration Effectiveness

Collaboration within FLOSS communities is enabled by the usage of tools, such as versioning systems, mailing lists, reporting systems, etc. These tools also serve as repositories which can be data mined to understand the identities of the individuals involved in a communication, the topics of their communication, the amount of information exchanged in each direction, as well as the amount of contribution in terms of code commits, bug fixing, reports and documentation produced and email postings. Such a large amount of data can be selectively collected and then analysed not only by using inferential statistics to identify activity patterns but also by using ontology engineering formalisms that support the extraction of semantic information. In the area of Empirical Software Engineering, cyber-archeology [SII07] has been applied to these repositories to learn and better understand the patterns of contribution of FLOSS developers in the projects concerned [SC10].

In this previous work [SC10] data collection has involved communications mainly in terms of participants, quantity and sometimes topics but neglected the objective collection of actual communication contents. At most, content data has been collected through questionnaires and surveys or through written reports by researcher who joined the community as observers, thus providing subjective rather than objective data. With reference to existing cognitive-based collaboration models [NM02], data mining methods can be used to extract content information from email communication and posting with the support of appropriate ontologies aiming to identify patterns, progress, evolution and achievements in the collaboration process occurring within groups of participants. This requires the analysis of incremental data and the construction

and analysis of graph data from the overall social networking aspect of the FLOSS community.

Cognitive-based collaboration theory [NM02] aims to consider many different factors underlying the mechanisms that connect community member understandings to community effectiveness in production. There is no single model that represents all of these factors, but separate models that address different factors. Only some of these models are relevant to a self-organising non-hierarchical community as is a FLOSS community.

The individual-team interplay model [NM02] is a cyclic model in which individuals *perform a task*, *notice need for interaction*, *prepare for interaction*, *perform the interaction* (posting, in the FLOSS context) and *go back to the task*, possibly *delivering the product of the task* (committing, in the FLOSS context) and then starting a new task.

The Cognition-Behaviour-Product model [NM02] emphasises the nature of the relationship between individual and team understandings, individual and team behaviours and individual and team production. In a FLOSS context, this model has the important role to explain how task quality and understanding affect each other and is essential in measuring individual task performance and collaboration effectiveness.

An important factor in collaboration is trust. Fong et al. [CFKSA, CFda], have developed a *filtering trust network model* to study about fusing elusive information and deriving trust factors in a social network, by taking Facebook as a case.

The above models can be used to define metrics for information interaction, task performance, product quality, peer trust [NM02, CFda]. In addition to metrics definition, some contextual inferring mechanisms are needed as a data pre-processing step for extracting the semantics and essences from the empirical data using machine learning techniques. Techniques previously applied to the analysis of public moods based on Internet comments [WFDia] can be also applied to implement such mechanisms.

3 Quality of Contributions

Quality of an individual's contribution to a FLOSS project can be measured in terms of three parameters: engagement, productivity and reputation.

Shaikh and Cerone [SC09] have identified some factors that are unique to the FLOSS development process and influence the entire software development process and, consequently, the quality of the final software product. In their work, Shaikh and Cerone also define an initial framework in which such factors can be related to each other and to the quality. In particular, they distinguish three main notions of quality in the context of FLOSS development

quality by access which aims to measure the degrees of availability, accessibility and readability of source code in relation to the media and tools used to directly access source code and all supporting materials such as the documentation, review reports, testing outcomes, as well as the format and structural organisation of both source code and supporting materials.

quality by development which aims to measure the efficiency of all development and communication processes involved in the production, evolution and release of source code, its execution, testing and review, as well as bug reporting and fixing;

quality by design which corresponds to the traditional notion of software quality [Pre00]: the end quality is judged by the design and implementation of the actual software and the code that underlies it.

These three notions of quality can be used as a basis for characterising the quality of engagement of an individual in the community. Every activity of an individual can be classified under an appropriate category of quality and marked to contribute to the final software product accordingly. Bug reporting, testing and reviews enhance quality by development, the media and format used to externalise such contributions affect quality by access, whereas evidence of planning and design, and validation of software code contribute to quality by design.

Number of commits and communication provide indicators of the level of contribution of an individual in the community and have been statistically analysed to determine patterns of contribution and their implications for the quality of code [SC10]. The number of commits describes how much the individual delivers in terms of product and is therefore an indicator of the individual productivity. Although there is no guarantee on the quality of the product delivered, number of commits can be considered by itself an important parameter in evaluating the quality of the individual as a contributor. Moreover, by integrating data on contribution in terms of commits, bug reports, bug fixing and on the approval and inclusion of the resultant artifacts in a release by the project leader team we can define a more accurate measure of the quality of the individual's productivity.

Communications among members of a FLOSS community can be analysed to extract information about the reputation an individual has achieved within the community. Text mining of communications can be used to identify keywords and phrases that may indicate whether an individual is asking or providing support and whether an answer or suggestion is taken on board or refuted by others. In addition, the filtering trust network model [CFKSA, CFda] discussed in Section 2 can highlight trust factors that contribute to build an individual's reputation.

The reputation of an individual depends not just on the participation to a specific FLOSS project, but on the global activities of that individual in the FLOSS world. Therefore reputation information of a given individual have to be collected over all FLOSS projects listing that individual as a participants and be integrated with personal information including the individual's background, publication in the FLOSS field and participation in related social networks and discussion fora. Finally, the level of engagement of a individual within a project is visible to the entire project community and, therefore, implicitly affects that individual's reputation.

Separate metrics can be defined to characterise engagement, productivity and reputation of individuals. However, it is still unclear how to combine such metrics into a global metric that quantify the quality of an individual's contribution to a specific FLOSS project. One of the challenges is represented by possible interrelations between the three metrics; for instance, we have pointed out above that engagement affects reputation.

4 Conclusion and Future Work

We have discussed possible indicators of collaboration effectiveness and quality of individuals' contribution which can be extracted from the data available in repositories of FLOSS projects. Such indicators can be used to define metrics that characterise collaboration effectiveness in

terms of information interaction, task performance, product quality and peer trust, and quality of individuals' contribution in terms of engagement, productivity and reputation.

In our future work we intend to merge such metrics into a global metric for FLOSS software quality to be used in a certification process.

Bibliography

[CFKSA] W. Chen, S. Fong, G. Kim. Multi-Collaborative Filtering Trust Network Model for Web 2.0 Recommender. In *2011 SIAM International Conference on Data Mining (SDM 2011)*. 20–30 April 2011, Mesa, Arizona, USA. To be published.

[CFda] W. Chen, S. Fong. Social Network Collaborative Filtering Framework and Online Trust Factors: a Case Study on Facebook. In *The 5th International Conference on Digital Information Management (ICDIM 2010)*. Pp. 266–273. July 2010, Thunder Bay, Canada.

[MRW77] J. A. McCall, P. K. Richards, G. F. Walters. Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality. 1977. http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA049014&Location=U2&doc=GetTRDoc.pdf

[NM02] D. Noble, Michael. Cognitive-Based Metrics to Evaluate Collaboration Effectiveness. In *Analysis of the Military Effectiveness of Future C2 Concepts and Systems*. RTO-MP-117, NATO Consultation, Command and Control Agency, The Hague, The Netherlands, 23-25 April 2002. http://www.rto.nato.int/Pubs/rdp.asp?RDP=RTO-MP-117

[Pre00] S. R. Pressman. *Software Engineering - A Practitioner's Approach*. McGraw-Hill International, London, 2000.

[SC09] S. A. Shaikh, A. Cerone. Towards a metric for Open Source Software Quality. *Electronic Communications of the EASST* Volume 20: Foundations and Techniques for Open Source Certification 2009, 2009.

[SC10] S. K. Sowe, A. Cerone. Integrating Data from Multiple Repositories to Analyze Patterns of Contribution in FOSS Projects. In *Proceedings of OpenCert 2010*. 2010. Vol. 33 of Electronic Communications of the EASST.

[SII07] S. K. Sowe, G. S. Ioannis, M. S. Ioannis (eds.). *Emerging Free and Open Source Software Practices*. IGI Global, 2007.

[WFDia] C. I. Weng, S. Fong, S. Deb. An Analytical model for Evaluating Public Moods Based on Internet Comments. In *National Conference on Data Mining (NCDM 2011)*. 19–2- February 2011, Pune, India.

A Formal Specification of the DNSSEC Model (Invited Talk)

Ezequiel Bazan Eixarch[1] **Carlos Luna**[1]

[1] ezequielbazan@gmail.com
FCEIA, Universidad Nacional de Rosario, Rosario, Argentina
[2] cluna@fing.edu.uy
InCo, Facultad de Ingeniera,Universidad de la Repblica, Montevideo, Uruguay

Abstract: Many applications are based in the use of Domain Names, making desirable to have trusted data into the Domain Name System (DNS). For this reason the Internet Engineer Task Force (IETF) has developed security extensions for DNS (DNSSEC). The aim of this paper is to provide a formal specification of the DNSSEC model, that allows to reason over the security properties of the chain of trust generated along the DNSSEC tree. This specification, which has been formalized in Coq, gives an abstract representation of the state and the security- related events, making possible to analyze important security goals, such as the effectiveness over cache poisoning attacks.

Keywords: Security properties, DNS, formal modelling.

Formal verification of a theory of packages (short paper)

Jaap Boender[1]*

[1] boender@pps.jussieu.fr
Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126 CNRS, F-75205 Paris, France

Abstract: Over the years, open source distributions have become increasingly large and complex—as an example, the latest Debian distribution contains almost 30 000 packages.

Consequently, the tools that deal with these distribution have also become more and more complex. Furthermore, to deal with increasing distribution sizes optimisation has become more important as well.

To make sure that correctness is not sacrificed for complexity and optimisation, it is important to verify the underlying assumptions formally.

In this paper, we present an example of such a verification: a formalisation in Coq of a theory of packages and their interdependencies.

Keywords: verification, theorem proving, open source distributions

1 Introduction

During the last decade, open source distributions have become more and more popular, and more and more complex. Where the first release of Debian only had 128 packages, the latest is well on its way to 30 000.

In order to deal with these complexities, the MANCOOSI project has developed new tools and methods to help distribution editors gain insight into the structure of their distributions, to more easily discover errors and to help with administrative procedures (such as migrating packages from unstable to stable distributions).

Since distributions are so large and complex, it is very important that these tools be as fast as possible, without sacrificing correctness. We have therefore used the Coq theorem prover to formally verify some of the assumptions used for optimising the MANCOOSI tools.

In the rest of this article, we will present a brief overview of the formalisation and explain the design decisions taken.

2 Formalisation

In this section, we will discuss the formalisation of the theory of packages as described in [MBD+06], [ADBZ09] and [DB10]. We shall not repeat the definitions presented there, only

* Partially supported by the European Community's 7th Framework Programme (FP7/2007-2013), grant agreement n°214898, "Mancoosi" project. Work developed at IRILL.

stating them informally, so that we can concentrate on the formalisation itself.

The Coq sources of the formalisation are available for download at
http://www.pps.jussieu.fr/~boender/package-theory.tar.gz.

2.1 Definitions

The formalisation has been done in the Coq proof assistant, which comes with an extensive library. The first choice to be made is of how to represent packages and repositories. In the original definition, packages are represented as a pair (u, v), where u is the name and $v \in \mathbb{N}$ the version. We have decided to treat packages as atomic entities, since we do not use the version number at all in our theories. If necessary, this could be changed easily, since the theory is set up in a very modular way.

A repository is a set of packages. There are at least three ways in Coq to represent sets: the `ListSet` library, which uses linked lists, the `Ensemble` library, which uses characteristic functions, and the `MSet` library. We have chosen the latter, because it is very modular, flexible, and comes with a large set of theorems. We do not need the complexity of the `Ensemble` library, since package repositories are finite.

An example of the modularity of the `MSet` library is the definition of conflicts. A conflict is a pair of packages, which can be expressed in the `MSet` library as a `PairOrderedType` with the package type as its substrate. In this way, we can use the entire `MSet` library for sets of conflicts without any extra work.

2.2 Dependencies

All definitions pertaining to dependencies have been put in a separate module. This is not only for ease of reference, but it also allows us to use the dependency module as a parameter later on. This is especially interesting when we start formalising the dependency cone (the transitive closure of the dependency function, i.e. all packages that could possibly be needed to install another package); by making this independent of the dependency function, we can easily specify different cones: the normal dependency cone, but also the cone that includes only conjunctive dependencies (dependencies that can only be satisfied by a single package), or even the reverse dependency cone (the cone of packages which depend on a given package).

The dependency function itself is a function `Dependencies` : `Package.t` $\rightarrow$ `listPackageSet.t`. The dependencies of a package are specified as a list of alternatives; every alternative is a set of packages, one of which has to be installed for the package itself to be installed.

We have added a 'filter' to the formalisation of dependencies, a function `dep_filter` : `PackageSet.t` $\rightarrow$ `bool`. This function allows us to consider only select dependencies, for example conjunctive ones (a dependency is conjunctive iff its set is a singleton).

This can come in useful for optimisations, for example, if q is a conjunctive dependency of p, it will always be part of an installation set of q. Using this lemma (proven in the formalisation) makes checking installability faster, since some 80 percent of dependencies is conjunctive.

2.3 The dependency cone

The dependency cone is defined as the transitive closure of the aforementioned dependency function. It is formalised as follows:

```
Function cone (P: {x: PackageSet.t | x [<=] R})
  {measure (fun x => cardinal R - cardinal (proj1_sig P)) P}: PackageSet.t :=
  if equal (inter R (dependencies (proj1_sig P))) (proj1_sig P)
  then (proj1_sig P)
  else
    cone (exist (fun v => v [<=] R) (inter R (dependencies (proj1_sig P)))
    (fun a => inter_subset_1 (s:=R) (s':=dependencies (proj1_sig P)) (a:=a))).
```

This definition basically states that the cone of a set P is the fixed point of the repeated application of the dependency function.

Note that the argument of `cone` is a dependent type: P has to be a subset of a repository R (as stated in the type) because this is necessary to prove termination; the `measure` keyword states that $|R| - |\texttt{dependencies}\ P|$ strictly decreases. Coq now generates the necessary proof obligations automatically.

This does complicate the definition: we need to use `proj1_sig` everywhere to extract the package set x from the dependent type. Additionally, for the recursive application of the `cone` function, we have to provide a proof that the recursive argument is also a subset of R; for this we use the `inter_subset_1` theorem which states that $\forall_{s,s'} s' \subseteq s \rightarrow (s \cap s') \subseteq s$.

2.4 Triangle conflicts

In [DB10], an algorithm was presented that finds strong conflicts, i.e. pairs of packages that cannot be installed together under any circumstances. This algorithm is much faster than the naive method of just checking every possible pair, by using a theorem that states that for two packages p and q not to be installable together, there must be an explicit conflict (c, c') such that p depends on c and q depends on c'. The algorithm can then traverse the dependency tree backwards from every explicit conflict, which limits the number of candidates to check.

An optimisation to this algorithm uses the concept of *triangle conflicts*. Here is its definition:

Definition 1 A conflict (c_1, c_2) is a triangle conflict if and only if there exists a package p such that:

- there is a $d \in \texttt{Dependencies}(p)$ such that $c_1 \in d$ and $c_2 \in d$;
- there is no other p' such that p' depends on either c_1 or c_2.

An example of this situation can be found in figure 1; `bravo` and `charlie` are in conflict, and there are no packages that depend on them except for `alpha`.

We have proven that triangle conflicts can be discarded when checking for strong conflicts. Informally, the reason for this is that since the only way to get to the two packages in conflict (c_1 and c_2) is through p; in order to install p, we can choose either c_1 or c_2. Since there are no other packages that depend on c_1 or c_2, which one we choose does not matter.

In Coq, we express this by the following theorem:

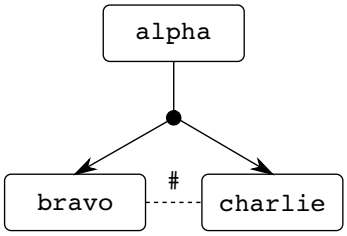

Figure 1: Example of a triangle conflict

```
Lemma triangles_ok:
  forall (R: PackageSet.t) (C: ConflictSet.t) (a: PackageSet.t | a [<=] R),
  (For_all (fun a => installable (NoSupDependencies R) R C a) (proj1_sig a)) ->
  only_triangles R C (fun _ => true) ->
  ~ConflictSet.Exists (fun c => let (c1, c2) := c in
    In c1 (proj1_sig a) \/ In c2 (proj1_sig a)) C ->
  co_installable (NoSupDependencies R) R C (proj1_sig a).
```

This expresses that given a repository R, a set of conflicts C, and a set of packages a, if all packages in a are installable separately, if the only conflicts are triangle conflicts, and if there is no direct conflict between any members of a, then all packages from a must be installable together.

The proof sketch is as follows:

- By induction on a.
  1. If $a = \emptyset$, then all packages in a are trivially installable together.
  2. If $a = a_1 \cup \{x\}$, then:
     - By the induction hypothesis, all packages from a_1 are installable together (take A_1 as the installation set). Also, x is a member of a, and therefore it is installable by the original hypothesis (with A_x as the installation set).
     - Now, we take the union of A_1 and A_x, where we remove c_1 or c_2 if both are present. This set is an installation set as well (it contains all packages needed and no conflicts).
- Hence, all packages from a are installable together.

Using triangle conflicts can result in a speed-up in checking a distribution for strong conflicts. Debian, for example, only has 60 triangle conflicts in its latest version, but one of these is the `debconf` package with two of its dependencies. The `debconf` package is needed by 11 000 other packages, which means that it generates 121 million candidates for strong conflicts. Given that the number of candidates after removal of triangle conflicts is only around 10 million, and that every candidate has to be checked using a SAT solver, it is easy to see that the removal of `debconf` alone results in a sizeable speed-up.

3 Conclusion and discussion

We have presented a short overview of the formalisation of the theory of packages as used in the MANCOOSI project.

This formalisation is not yet complete. For example, the proof presented in [Boe11] has not yet been formalised, mostly due to the lack of an appropriate graph theory library in Coq.

The ultimate goal of this formalisation is to be able to formally verify the actual algorithms and tools used in the MANCOOSI project, of which the computation of strong conflicts is one (relatively simple) example. Since modern distributions are very large, the use of optimisations is necessary, but we should be careful that these optimisations do not result in a loss of correctness. Formal verification can be of use in that regard.

Most of the algorithms in MANCOOSI rely on a SAT solver to check for installability. Formally verifying a modern SAT solver would be very difficult, but it might be possible to treat the SAT solver as a black box and only verify its result.

Acknowledgements: The author would like to thank Roberto Di Cosmo and Pierre Letouzey for their very useful advice and encouragement.

References

[ADBZ09] P. Abate, R. Di Cosmo, J. Boender, S. Zacchiroli. Strong dependencies between software components. In *ESEM '09: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. Pp. 89–99. IEEE Computer Society, Washington, DC, USA, 2009. doi:http://dx.doi.org/10.1109/ESEM.2009.5316017

[Boe11] J. Boender. Efficient Computation of Dominance in Component Systems. In *Proceedings of SEFM*. 2011.

[DB10] R. Di Cosmo, J. Boender. Using strong conflicts to detect quality issues in component-based complex systems. In *ISEC '10: Proceedings of the 3rd India software engineering conference*. Pp. 163–172. ACM, New York, NY, USA, 2010. doi:http://doi.acm.org/10.1145/1730874.1730905

[MBD+06] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, R. Treinen. Managing the Complexity of Large Free and Open Source Package-Bas ed Software Distributions. In *ASE*. Pp. 199–208. 2006.

Using antipatterns to improve the quality of FLOSS development (short paper)

Dimitrios Settas and Antonio Cerone

settdimi@iist.unu.edu cerone@iist.unu.edu
United Nations University
International Institute for Software Technology
Macau SAR, China

Abstract: Antipatterns have been mostly reported in closed source software environments. With the advent of Free/Libre Open Source Software (FLOSS), researchers have started analyzing popular FLOSS projects, seeking vitality indicators and success patterns. However, an impressively high percentage of FLOSS projects are unsuccessful. Moreover, even in the successful cases of FLOSS there can be found tracks of failed attempts, dead-ends, forks, abandonments etc. FLOSS antipatterns can help developers to improve their code and improve the communication and collaboration within the FLOSS community. Moreover, they can be used as quality indicators in the certification of FLOSS products.

Keywords: FLOSS development, antipatterns, certification, ontology

1 Introduction

An antipattern is a new form of pattern that has two solutions [BMMM98]. The first is a problematic solution with negative consequences and the other is a refactored solution, which describes how to change the antipattern into a healthy solution. The second solution is what makes antipatterns beneficial. The difference is in the context: An antipattern is a pattern with inappropriate context and is particularly useful in the case of knowledge representation, because it captures experience and provides information on commonly occurring solutions to problems that generate negative consequences [LN06]. The process that is followed by a pattern to change its solution into a better one is called refactoring. This solution has an improved structure that provides more benefits than the original solution and refactors the system toward minimised consequences.

FLOSS anti-patterns are not yet explored to the same extent as in closed source. In addition, because FLOSS and closed source software produce code using very different development processes, FLOSS antipatterns are quite different in nature from their closed source counterparts. There exist different categories of antipatterns. According to the literature [BMMM98, LN06] closed source software antipatterns exist at a development, architectural and managerial level. FLOSS antipatterns mostly exist at a community level and describe social and managerial issues regarding communication, interaction and coordination among developers that participate in FLOSS projects. However, closed source software development antipatterns are also applicable in FLOSS projects and can greatly affect the quality of both closed and open source software projects.

This short paper describes how antipatterns can be used in FLOSS projects by defining the sources of these antipatterns and the different user roles of FLOSS antipatterns. While antipatterns cannot be used as a formal certification approach, different kinds of antipatterns (i.e. development, community level) can be used (1) within the FLOSS development process to directly overcome problems that may affect the certification of the FLOSS product and (2) as part of the certification process to define indicators of the quality of the development process and the resultant FLOSS product. Development antipatterns may help developers overcome commonly occuring coding problems. For example, the "Spaghetti Code" antipattern [BMMM98] can be used to describe code that has a complex and tangled control structure, especially one with several exceptions, threads, or other "unstructured" branching constructs. Spaghetti code can be caused by several factors, including inexperienced programmers and a complex program which has been continuously modified over a long life cycle. A solution proposed to resolve the antipattern is using a formal and predictable style of coding such as that of Structured Programming. Community antipatterns can help developers overcome problematic FLOSS practices, such as participation and motivation problems, which are crucial to FLOSS development. For example, "The Big Show" antipattern [Nea11] describes the scenario in which companies will work for several months on a software project behind closed doors before announcing it to the public. This behaviour negatively impacts the ability of a FLOSS community to grow outside corporate walls. The project members are assigned to "secret" projects and interact less with the community, and the end result is a big code drop which has not had public peer review and was not listed on any roadmap before its announcement, resulting in people outside the company feeling like second class citizens.

Since encountered coding problems and problematic community practices hightlighted by such antipatterns affect the quality of the FLOSS product, once an antipattern has been identified, it may be incorporated in the certification process as a negative quality indicator. One problem is that identifying antipatterns is essentially a qualitative process, in which symptoms are associated to antipatterns either directly or indirectly by means of existing causal relationships between antipatterns. Such a qualitative nature of antipattern identification is not adequate to the the accuracy required by a formal certification process. In order to overcome this problem, metrics for technical quality, based on the ISO/IEC 9126 standard, that have been successfully applied to proprietary software as well as to FLOSS [CV08] could be customised to detect specific antipatterns that describe coding problems such as the Spaghetti Code antipattern.

The main source of FLOSS antipatterns is the Web, as no further FLOSS community antipatterns have been published elsewhere at the moment. These antipatterns do not use official antipattern templates proposed by Brown *et al.* [BMMM98] or Laplante and Neil [LN06]. They are documented using a short textual description in order to quickly describe them and allow users to memorise them easily. An open issue that has not been resolved for community antipatterns is the lack of formalisation of this knowledge, which, in fact, would be essential to enable the use of software tools to support this technology as well as to quantify the process of their identification. Furthermore, as far as the authors are aware there is no single knowledge base that documents these antipatterns and allow their use by software tools.

2 FLOSS Community Antipatterns

FLOSS community antipatterns are very relevant to the certification of FLOSS projects as they describe ways to build stronger project communities with effective communication, collaboration and management practices.

Tobrien [Tob] has recently proposed six OSS community antipatterns, which are caracterised by six "personalities" that have emerged in the last years among FLOSS communities: Rule maker, Open Source Politician, Attack Dog, Non-contributing Pontificator, Back Room Dealer and (Apache Way) Ambassadors. For each of these antipattern, Tobrien describes the causes, symptoms and consequences as well as the refactored solution that can make the antipattern beneficial to an FLOSS project. For example, although it is important to have a few rulmakers within the community, when their growth in numbers of individuals and their attempts to advocate and enforce standards appears to hinder productive community activities, it may be necessary to split the community up into smaller, more focused teams, each governed by an active code contributor, who is expected to have a natural tendency to avoid needless bureaucracy [Tob].

Josh Berkus [Ber11] has also listed ten ways on how to destroy a FLOSS community. These antipatterns are likely to arise when companies are involved or even are leading FLOSS projects. For example, a company leading a FLOSS project may permanently prevent anybody outside the company from having commit access, respond evasively to queries and/or choose employees who write no code as committers on the project.

Leung [Leu08] has identified 22 FLOSS antipatterns and has provided a very short description of each one. Most importantly he identified the lack of management itself as an antipattern emphasising the need for managing FLOSS projects.

Dave Neary [Nea11] has developed a Community Management Wiki in which he maintains 18 FLOSS community antipatterns. These antipatterns are described in more detail and include the symptoms of each antipattern together with their refactored solution. For example, the "anarchy" antipattern highlights the problems associated with the "Free Software is all about absolute Freedom" way of thinking. This kind of thinking leads to a kind of anarchistic community where a substantial proportion of members affirm a total right to freedom of speech, freedom of expression, and other extreme libertarian principles. The result is a community where no indiscression can easily be corrected, because any attempt to do so is perceived as a limit to the absolute freedom of speech.

These 56 antipatterns contain valuable knowledge that can be used from the users of a FLOSS community in order to identify problematic practices or dysfunctional processes that have been previously documented as antipatterns. However, memorising such a big number of antipatterns is problematic and requires the use of software tools that support the technology of antipatterns.

3 Antipatterns Users within the FLOSS Development Process

In addition to the 56 community level antipatterns considered in Section 2 different kinds of traditional antipatterns that have been documented in the literature [BMMM98, LN06] and the Web [Malb, Mala, WCa, WCb] can be used at either development level by FLOSS developers (active developers or contributors) or at managerial level (by project leaders or the company

which leads the community). Therefore, defining the FLOSS user types that could benefit from this technology is very important. These are:

Developers who are the biggest part of the community by contributing and by reporting bugs. They can benefit from both development antipatterns that have been already documented in the literature and antipatterns that are specific to FLOSS community.

Users of the software, who are also part of the community and download software or exchange messages in mailing lists. They mainly benefit from community level antipatterns.

Companies FLOSS might be a crucial element in a product or service provided by a company and this company will have to be active or even lead the community [Saa]. Companies benefit from both community level antipatterns [Ber11] and management antipatterns that have been already documented in the literature.

Leaders or Managers of open source communities can benefit from the existence of project management antipatterns that exist in the literature and the Web.

Learners Students and free lerners that participate in FLOSS projects in order to learn from more advanced developers or wish to experience FLOSS development. They can benefit from all kinds of antipatterns.

Moreover, all these FLOSS user types would greatly benefit from working in an environment that provides tools that support antipattern technology. The antipattern ontology [SMSB11] has been developed using the Web Ontology Language (OWL) and define antipatterns in terms of concepts and relationships. It has been implemented in the antipattern ontology Webprotege installation [SMSB11], which allows users to participate in the enrichment of the content of the ontology or edit the existing antipatterns according to their user rights. As a first step towards the use of tool-supported antipattern technology within FLOSS communities, the OWL antipattern ontology has been enriched with data from 13 FLOSS community antipatterns that exist on the Web.

4 Using Antipatterns to support the FLOSS Certification Process

Community level antipatterns are strongly related to the notion of *quality by development* defined by Shaikh and Cerone [SC09] as specific to the FLOSS development process. Quality by development aims to measure the efficiency of all development and communication processes involved in the production, evolution and release of source code, its execution, testing and review, as well as bug reporting and fixing; [SC09]. The idea of *quality by development*, therefore, is an attempt to measure the efficiency of such processes and the interaction between them.

Shaikh and Cerone identify factors that characterise the inherent quality aspects of these processes, but also observe that it is the overall management of the project to play a more central role, with communication and coordination being the two key aspects of this. Management quality aspects that cannot be fully captured by Shaikh and Cerone's quality model, may instead be described in terms of antipatterns. "Make the project depend as much as possible on difficult

tools", "provide no documentation", "employ large amounts of legalese", "governance obfuscation" and "don't answer queries" are examples of community level antipatterns that have been described by Josh Berkus [Ber11].

Symptoms of these antipatterns may be detected through the analysis of communications and the activities of the FLOSS project. Collaboration in FLOSS projects is highly mediated by the usage of tools, such as versioning systems, mailing lists, reporting systems, etc. These tools serve as repositories which can be data mined; data can be selectively collected and then analysed not only by using inferential statistics to identify activity patterns [SC10] but also by using ontology engineering formalisms that support the extraction of semantic information. Appropriate ontologies aiming to identify symptoms of antipatterns as well as the solutions applied to solve such antipatterns [SMSB11], together with weights for the severity of the antipatterns and the effectiveness of the applied solution, can enable the extraction of quantitative information to be used as a measure for quality by development.

5 Conclusion and Future Work

We have presented the potential benefits of antipatterns usage within the FLOSS development process to allow various types of users to overcome problems that may affect the certification of the FLOSS product. The enrichment of the OWL antipattern ontology with data from 13 FLOSS community antipatterns provides a way of realising these potential benefits. There are clear advantages to the certification of FLOSS software that come from improving the quality of the developed software both at a development level but also by overcoming FLOSS community problems. The strong social aspect of the collaborative development of the antipattern ontology will ultimately increase the communication of FLOSS user roles and will provide a platform in which FLOSS users can discuss their problems and possible solutions using antipatterns.

We have also suggested how to relate antipatterns, especially community level antipatterns, and their applied solutions to FLOSS quality by development as part of a FLOSS certification process. Development within a FLOSS projects is not dictated by prescriptive rules, but naturally emerges as the product of community activities. It is therefore an important parameter that strongly affect the quality of the FLOSS product. Defining a metric for quality by development, which is part of our future work, would be essential in the creation of a FLOSS certification process.

Bibliography

[Ber11] J. Berkus. How to destroy your community. 2011. http://lwn.net/Articles/370157/

[BMMM98] W. Brown, R. Malveau, H. McCormick, T. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley Computer Publishing, 1998.

[CV08] J. P. Correia, J. Visser. Certification of Technical Quality of Software Products. In *Proceedings of the OpenCert and FLOSS-FM 2008 joint Workshop*. UNU-IIST Research Report 398, pp. 35–51. 2008.

[Leu08] T. Leung. OReilly OSCON Open Source Convention. 2008.

[LN06] P. Laplante, C. Neil. *Antipatterns: Identification, Refactoring and Management*. Taylor and Francis, 2006.

[Mala] N. Malik. Software Project Management Antipattern Blog, Pardon my dust. http://blogs.msdn.com/nickmalik/archive/2006/01/19/PMAntipattern-Pardon-My-Dust.aspx

[Malb] N. Malik. Software Project Management Antipattern Blog, Project Managers who write specs. http://blogs.msdn.com/nickmalik/archive/2006/01/03/508964.aspx

[Nea11] D. Neary. OSS community management Wiki. 2011. http://communitymgt.wikia.com/wiki/Category:Anti-patterns,

[Saa] M. Saastamoinen. Managing OSS As an Integrated Part of Business (OSSI), The linux foundation. https://fossbazaar.org/content/managing-oss-integrated-part-business-ossi-final-report

[SC09] S. A. Shaikh, A. Cerone. Towards a metrics for Open Source Software Quality. In *Proc of OpenCert 2009*. 2009. Vol. 20 of ECEASST.

[SC10] S. K. Sowe, A. Cerone. Integrating Data from Multiple Repositories to Analyze Patterns of Contribution in FOSS Projects. In *Proc of OpenCert 2010*. 2010. Vol. 33 of ECEASST.

[SMSB11] D. L. Settas, G. Meditskos, I. G. Stamelos, N. Bassiliades. Detecting antipatterns using a Web-based collaborative antipattern ontology knowledge base. In *Proc. of ONTOSE 2011*. Volume 83, pp. 478–488. Springer, 2011.

[Tob] Tobrien. 6 Open Source Community Anti-patterns (or Less Talk. More Do.). http://www.discursive.com/blog/4355

[WCa] Wiki-Community. Pattern Community Antipattern Catalogue. http://c2.com/cgi/wiki?AntiPatternsCatalog

[WCb] Wiki-Community. Wikipedia Antipatterns Community Catalogue. http://en.wikipedia.org/wiki/Anti-pattern