OSS-TMM: Guidelines for Improving the Testing Process of Open Source Software

Sandro Morasca, Davide Taibi, and Davide Tosi
Università degli Studi dell'Insubria
Dipartimento di Informatica e Comunicazione
[sandro.morasca, davide.taibi, davide.tosi]@uninsubria.it

ABSTRACT

Open Source Software (OSS) products do not usually follow the traditional software engineering development paradigms found in textbooks. Specifically, testing activities in OSS development may be quite different from those carried out in Closed Source Software (CSS) development, also due to the fact that OSS processes often seem to be less structured than CSS ones. Since testing and, in general, verification may require a good deal of resources in OSS, it is necessary to have ways for assessing and improving OSS testing processes.
In this paper, we provide a set of testing guidelines and testing issues that OSS developers may want to take into account to decide which testing techniques make most sense for their OSS products. This paper 1) provides a checklist that helps OSS developers identify the most useful testing techniques according to the main characteristics of their products under test and 2) outlines a proposal for a method that helps assess the maturity of OSS testing processes. The method is a proposal of a Maturity Model for testing processes (which we call OSS-TMM). To show its usefulness, we have applied the method to seven real-life projects. Specifically, we present in the paper how we applied the method to BusyBox, Apache Httpd and Eclipse Test & Performance Tools Platform to show how the checklist supports and guides the testing process of these OSS products.

Keywords: Open Source Software Quality, Trustworthiness, Testing Process, Survey, Testing Quality Assessment

1. INTRODUCTION

Open Source Software (OSS) is currently enjoying increasing popularity and diffusion in industrial environments. However, verification and testing of OSS systems have not received the amount of attention they often have in Closed Source Software (CSS) (Zhao & Elbaum, 2003; Tosi & Tahir, 2010). Even very well-known OSS projects, such as Apache Httpd or the GCC compiler, do not seem to have mature testing processes and test suites defined inside their development process. This is probably due to a few mutually related reasons, which we outline.
1. Some testing techniques that are well established for CSS are not directly applicable to OSS systems, so a good deal of effort and cost is required for designing new testing solutions that are created *ad-hoc* for OSS systems.
2. OSS system development as a whole hardly ever follows the classic software engineering paradigms found in textbooks, so the execution of testing activities for OSS is less structured than for CSS.
3. The planning and the monitoring of the testing process of an OSS system hardly ever follow the guidelines used for CSS systems, so it is necessary to redefine some of the methods that are at the basis of the testing process.
4. OSS project testing often relies on the contributions from end-users, more than CSS does. This may occur via the distribution of testing or beta releases, or by using the feedback from users on stable releases. This allows OSS developers to shorten release cycles. In a way, this was stated by Linus

Torvalds as "Given enough eyeballs, all bugs are shallow," which can be interpreted as referring to the importance of code review by peers, but also to the importance and power of testing by end-users contributing to OSS development with their feedback. The extent to which this occurs is larger than in CSS, and much larger than for products of other, more mature industrial sectors.

5. Often, OSS end-users independently test OSS products before using them. So, it is likely that the same kinds of tests are carried out over and over again by different users. This implies some wasted effort for end-users and lack of maturity of the testing process. In addition, this is against the very motivations of OSS, which implies sharing effort and reusing results, including those related to software verification and validation.

6. The lack of maturity of OSS testing processes is also indicated by the low coverage levels that are achieved by the test suites that are found along OSS products. Some results of the QualiPSo project show that the statement coverage levels of a number of well-known OSS projects do not exceed 25%. Clearly, this does not imply that at least 75% of source code statements have never been tested. Instead, this shows that a lot of the testing that is carried out goes undocumented, which is a symptom of low maturity.

Our experience in the context of OSS projects suggests that OSS communities do not usually view software testing as a primary software development activity. Also, most OSS projects do not fully integrate testing activities into their development process. In a survey, we asked 151 OSS stakeholders (developers, contributors, managers, end-users, etc.) to rate the importance of a number of factors that they take into account during the adoption of OSS components and products. The complete survey can be found in (QualiPSo 1, 2010). It is worth noting that about 90% of the interviewees were actually involved in OSS projects as developers, contributors, integrators, managers, etc., and only 10% were end-users. Interviewees answered on average that the factor "existence of benchmarks / test suites that witness for the quality of OSS" has low importance. This may actually be a result of the fact that benchmarks and test suites are hardly ever available for OSS, more than the fact that benchmarks and test suites might not be important. So, OSS stakeholders do not use benchmarks and test suites simply because they often do not exist. We also analyzed the web portals of 33 well-known OSS products (Tosi & Tahir, 2010) and we discovered that, in the web portals

- only 6% of the products provide the availability of a complete test suites
- 21% of the products provide performance benchmarks
- 3% show the usage of a testing framework to support testing activities
- 18% provide complete results about test suite executions
- 41% provide internal (or external) reports about the executions of benchmarks.

These somewhat discouraging data are in contrast with the trend followed by CSS products, for which software quality and consequently testing are considered very important during the development process. In CSS development, the testing process is much more rigorously defined and is considered a key factor for achieving high quality. In OSS development, bug detection and fixing depends much less on testing process rigor. OSS follows the "release early, release often" paradigm to have continual improvement and achieve high quality (Aberdour, 2007).

In (Morasca et al., 2009) we thoroughly discussed some of the inherent characteristics of OSS products and a set of testing techniques that may address these characteristics. In this paper, we extend our preliminary work as follows.

1. We provide a set of testing guidelines that OSS developers may follow to decide which testing techniques make most sense for their OSS products. We introduce a checklist to help OSS developers optimize and improve the testing practices of their products, by identifying what needs to be checked to cause as many failures as possible by means of specific testing activities and test suites, according to the main characteristics of their products under test.
2. We merge these contributions in a proposal for a method for assessing the maturity of OSS testing processes, OSS-TMM (Open Source Software Testing Maturity Model). OSS-TMM is the first step towards the definition of a new Maturity Model specifically focused on OSS testing, whose final goal is the quality improvement of OSS products by supporting the planning, monitoring, and execution of testing activities.

Unlike existing models (Burnstein et al., 1996; Herbsleb et al., 1997; El Emam, 1997), which have been defined with CSS characteristics in mind, OSS-TMM takes into account the specific issues that characterize OSS from CSS systems.

3. To provide evidence of the usefulness of the checklist, we have applied OSS-TMM step-by-step to three real-life OSS projects: BusyBox, Apache Httpd and TPTP, and we compared the assessed scores with the density of reported bugs to understand whether a correlation exists between the two indicators. Moreover, we applied OSS-TMM to four additional representative OSS products, to assess the maturity of their testing process.

The paper is structured as follows. Section 2 introduces the checklist and a set of testing guidelines for OSS products. Section 3 describes how to use the checklist and the guidelines for the assessment and the improvement of OSS testing processes. Section 4 details our experience with BusyBox, Apache Httpd, and TPTP, and reports on the maturity of four additional OSS projects. Section 5 discusses the state of the art in certifying software products. Section 6 draws our conclusions and future work.

2. TESTING GUIDELINES FOR OSS

We provide a set of testing guidelines and OSS-TMM to complement the testing mechanisms normally used in the OSS domain. Our goal is to help developers and contributors improve their testing process with relatively little effort. Specifically:

1. From a developers' point of view, OSS-TMM's goal is to simplify the internal process of testing OSS products by suggesting a rapid way for identifying a testing plan that best fits the characteristics of the OSS product. Also, OSS-TMM aims at simplifying the assessment or certification process of OSS products by comparing available testing activities and the activities suggested by our guidelines. OSS-TMM may also speed up the testing activities by guiding developers in selecting testing strategies depending on the characteristics of their OSS product and may increase the quality and the trustworthiness perception of the OSS by improving the testing activity.
2. From the end-user's point of view, OSS-TMM aims at simplifying and speeding up the selection of an OSS product by evaluating the maturity of the testing process as a possible indicator about the whole quality of the product.

We have identified a comprehensive set of issues that characterize OSS systems and, correspondingly, a set of testing techniques that best fit the characteristics of OSS products (Morasca et al., 2009). In this section, these two sets are summarized and presented in a checklist that can be used by developers and end-user to identify the peculiarities of their OSS products, discover the level of compliance of the target OSS product with the typical OSS characteristics, and define the best testing practices for the target product. While some of them are not individually unique to OSS development, their ensemble characterizes OSS testing.

Checklist and Guidelines

Here, we report on the checklist, which is one of the improvements provided in this work. Table 1 shows the checklist. The checklist refers to the five main issues (I1 to I5) discussed in detail in our previous paper (Morasca et al., 2009).

The first issue (I1) addresses the full visibility of the logic and structure of the code. The second issue (I2) addresses all the aspects related to system analysis and product design activities, such as the definition of system requirements, risks, and quality assurance methodologies. The third issue (I3) addresses the way OSS products are developed and takes into account the concepts of collaborative and distributed development, virtual communities, and the idea of "unstructured companies" (Raymond, 2001). Issue I4 addresses the rapid growth that characterizes OSS systems (Mockus et al., 2000) and the creativity that characterizes the communities around OSS products (O'Reilly, 1999). Issue I5 addresses the way OSS products and projects are documented, and how these documents are disseminated.

The checklist itemizes a set of questions for each issue (see the left column of Table 1). Each question stresses a specific sub-issue that may (or may not) characterize the OSS project under concern.

	Y	P	N	Guidelines
I1 – Code Visibility				
I1.1 Is the source code available via Versioning Systems?				IF Y: regression testing is required for versioned projects to avoid the risk of introducing new bugs from a release to another; white-box unit and integration testing is required for projects structured in components, units of code, packages to test each unit of code in isolation or in integration and provide evidence that the functionality of each unit is implemented as specified. (IEEE Std 1008, 1986; Leung & White, 1990; Pezzè & Young, 2007)
I1.2 Is the project structured in folders: source, binary, libraries, docs?				
I1.3 Is information about releases (date, number, change-log) visible?				
I1.4 Is information about code revision (author, number, description) visible?				
I1.5 Are security issues meaningful for the product (private data, sensible data)?				IF Y: formal testing is required for functions that manipulate private / sensible data, penetration tests, dependencies tests, risk-based security tests (Howard, 2006; Dannenberg & Ernst, 1982; Arkin et al., 2005)
I1.6 Are the scripts of the test cases open source?				
I1.7 Does the project integrate third-party components and is it released under several OSI licenses[1] ?				IF Y: compatibility checks among the different licenses adopted by the project are needed to satisfy all the licenses dependencies and to avoid legal implications (Tuunanen, 2009; Fossology, 2010; OSLC, 2010)
I1.8 Are log files about system executions available?		-		IF Y: dynamic analysis techniques are applicable to test at run-time the behavior of the system against the expected one (Ernst et al., 2001; Orso, 2010)
I2 - System Analysis and Product Design Activities				
I2.1 Is a project plan/roadmap available?		-		IF Y: the available system analysis should be exploited to design oracles and test cases for black box testing (category partition, catalogs, hw/sw requirements testing, acceptance testing) (Ostrand & Balcer, 1988; Pezzè & Young, 2007); IF N: evolutionary testing (R.A. Santelices et al., 2008)
I2.2 Is a risk analysis available?		-		
I2.3 Is a requirements analysis available?		-		
I2.4 Is a goal analysis available?		-		
I2.5 Are system designs available (in UML or other notations)?				IF Y: the available models should be used as starting point for model-based testing to derive test cases from the abstract representation of the system (Pretschner et al., 2005)
I2.6 Are standard protocols or patterns identified?		-		IF Y: conformance testing should be applied to verify the protocol behavior and to check the adherence of the source code to the identified standards (Bernhard, 1994)
I2.7 Are coding standards and conventions identified?		-		IF Y: style check and inspection should be used to verify the adherence to the identified conventions (PMD, 2010; Fagan, 1986)
I2.8 Are performance requirements meaningful (real-time, normal constraints)?				IF Y: performance testing (load testing, stress testing, endurance testing) (IEEE Std 829, 2008; Weyuker & Vokolos, 2000; Pezzè & Young, 2007)
I2.9 Does the system follow a specific architectural style (e.g., SOA, peer-to-peer)?		-		IF Y: select the testing techniques specialized for the chosen architectural style (e.g., service-oriented architectures prefer on-line testing) (Mao et al., 2007)
I2.10 Is the system developed with a GUI?		-		IF Y: use capture&replay tools to automate the execution of GUI tests. Consider also usability aspects, and evaluate the product by testing it on users (Jacareto, 2010, Nielsen, 1999)

[1] See www.opensource.org/licenses/ for an exhaustive list of licenses approved by the Open Source Initiative (OSI)

I2.11 Does the product use external libraries/plugins?		-		IF Y: check compliance among the target product version and the integrated modules through versioning compatibility checks and installation testing activities (Agruss, 2000)
I3 - Development Process				
I3.1 Is a specific development process used (e.g., waterfall, continuous building)?		-		IF Y: verify whether the testing process is in line with the chosen development process to optimize the whole process and the use of available resources
I3.2 Are developers/contributors structured in teams?		-		IF I3.2 N and I3.3 Y: use the sand box as testing environment;
I3.3 Does the system provide a sand box environment?		-		
I3.4 Is a specific IDE used/recommended?		-		IF Y: make the most of the testing potentialities offered by the chosen IDE by exploiting external plugins or components that can support testing;
I3.5 Is a testing platform used/recommended?		-		IF N: select a testing framework (Open, 2010) to support and automate the testing process;
I3.6 Is a bug tracking system available?		-		IF Y: fault-based testing techniques to demonstrate the absence of a set of pre-specified and reported faults (Morell, 1990)
I4 - System Growth and Community Creativity				
I4.1 Is the number of code changes per release > 500 ?		-		Integration and regression testing activities should be improved proportionally to the size of the community and the vitality of the project. The same should happen for the automation of testing, the sharing of testing knowledge to increase the reusability of the test suites, the documentation of test-strategy/tests-results, and the monitoring of testing activities. Also exploit *online built-in testing methodologies* to automatically collect input-output and interaction data. This facilitates the discover of functional and non-functional misbehavior (Mao et al., 2007)
I4.2 Is the number of developers/contributors > 100 ? (small community < 10; medium < 100; big > 100)		-		
I4.3 Does the system have different releases/system updates?		-		
I4.4 Is the number of open bugs/fixed bugs/... available?		-		
I4.5 Is the frequency of changes/updates/bug time solving recognizable? (to evaluate whether the project is active)		-		
I5 - Documentation and Dissemination				
I5.1 Is a system-level documentation available?		-		IF Y: exploit the documentation to simplify acceptance/system testing, usability testing, installation testing (IEEE Std 829, 2008)
I5.2 Is a library-level documentation available?		-		
I5.3 Is a feature-level documentation available?		-		
I5.4 Is a user manual available?		-		
I5.5 Are bugs reports available?		-		
I5.6 Is code documentation available (javadoc, etc.)?				
I5.7 Are docs disseminated via unstructured channels (forums, mailing lists)?		-		
I5.8 Are Installation requirements documented?				
I5.9 Are test-plans/test-designs/test-results not available?				IF Y: provide testing documentation through "test management tools" (such as TestLink, qaManager, etc.) that automate/simplify the generation of reports (Morasca et al., 2010; Open, 2010)

Table 1: Checklist and Guidelines for testing OSS products

The evaluator assesses the availability of each sub-issue by ticking the box "Y" if the sub-issue fully characterizes the system under analysis, "P" if the sub-issue partially characterizes the system under analysis, and "N" otherwise. Some sub-issues may not be applicable to the target system: if a sub-issue is not applicable, the testing guideline associated must be discarded. When the process is completed and all the checklist entries have been checked, the evaluator simply should take into account the testing techniques, which have a real impact on the system, by following the testing guidelines reported in Table 1 column <<*Guidelines*>>. Each testing technique is supplied with references to the literature. Questions and guidelines are formulated to limit subjectivity.

In Sections 3 and 4, we will present how to use the checklist and how to apply it to real-life OSS projects.

3. OSS-TMM DESCRIPTION

In this section, we draw a step-by-step method (OSS-TMM) that developers and end-users can follow to assess the maturity level of a testing process.

Process Assessment

The approach we propose is compliant with the ISO/IEC14598 standard (ISO/IEC14598, 2001), which gives guidance and requirements for evaluating software processes. OSS-TMM is based on five main steps (S1 – S5), as described next.

S1: Take the checklist of Table 1, sequentially scan each entry of the checklist, and inspect the target OSS product with reference to the issues reported in the checklist. Answer each question of the checklist and identify the testing guidelines that best fit the characteristics of your product by means of the testing guidelines provided in column <<*Guidelines*>> of Table 1.

S2: Define the Best Testing Practices (BTP) on the basis of the results of step S1. By "best testing practices," we mean the most mature process that can be theoretically achieved with reference to the inherent characteristics of the product. BTP will be a list of testing techniques that should be applied to the product under assessment to optimize its testing process. Of course, the definition of the BTP is not fully objective due to the huge number of testing techniques and practices that are potentially useful. Our intent is to suggest a survey of representative testing activities and technologies, and not a rigid model, due to the rapid evolution of the field, especially in the OSS world.

S3: Isolate and briefly analyze the currently Available Testing Practices (ATP) of the product in order to list the properties and the already used testing techniques. To do this, developers can for example ask the community whether (and which) testing activities have been performed on the product. End-users can for example surf the web portal of the product to collect data about testing activities.

S4: Verify the intersection degree between the activities of the testing process model derived in step S2 (BTP), with the ones analyzed in step S3 (ATP), and estimate the maturity level (ML) of the testing process referring to the four maturity levels identified in the next subsection.

S5: End-users can use the maturity level as an indicator that contributes to assessing the quality of the OSS product they are evaluating. Developers can use the maturity level to evaluate whether their ATP needs improvement. If this is the case, they can improve the testing process by following the recommendations and guidelines provided in the BTP.

Maturity Levels

In compliance with existing certification and maturity models, we identified four maturity levels (ML) that reflect the evolution of the testing process from one that is unstructured and undefined (ML 1) to one that is well planned, monitored, and optimized (ML 4). Refer to Section 5 for more details about other certification and maturity models.
Unlike in CMM and TMM (Herbsleb et al., 1997; Burnstein et al., 1996), our levels are not defined and structured as sets of predetermined maturity characteristics and goals, but they depend on the specific characteristics of the product under evaluation. Burnstein et al. define five maturity levels of a testing process starting from Level 1, in which the testing process is initial and not distinguishable from debugging, to Level 5 in which the testing process has a set of defined testing policies, a test life cycle, a test planning process, a test group, a test process improvement group, a set of test-related metrics, appropriate tools and equipments, controlling and tracking mechanisms, and finally a product quality control. Such a model is unsuitable in the OSS scenario where the testing process strongly depends on the inherent issues of the target product, where resources are limited for implementing such a testing process with several groups that work on testing, where developers are unstructured and are difficult

to group in teams, where each developer/tester has the fully visibility of the code instead of a part of the product as often happen in closed-source projects, where system analysis and product design are usually not well planned activities thus making hard the definition of preplanned testing plans. In our experiments, we quickly applied Burnstein's Testing Maturity Model TMM to the 33 OSS products of our experiments, and we discovered that the vast majority of products fall into the first and second level. Therefore, TMM does not seem to be useful for differentiating the testing maturity levels of OSS products.

Hence, we identified four maturity levels with less stringent requirements than TMM, which are dynamically computed for each product by applying OSS-TMM. The fours maturity levels are defined as follows.

ML1: The activities performed by the ATP cover 25% of the activities suggested by BTP at the most. As a formula:
$|ATP \cap BTP| < 25\%|BTP|$

ML2: The activities performed by the ATP cover 25% to 50% of the activities suggested by the BTP. As a formula:
$25\%|BTP| \leq |ATP \cap BTP| < 50\%|BTP|$

ML3: The activities performed by the ATP cover 50% to 75% of the activities suggested by the BTP. As a formula:
$50\%|BTP| \leq |ATP \cap BTP| < 75\%|BTP|$

ML4: The activities performed by the ATP cover at least 75% of the activities suggested by the BTP. As a formula:
$75\%|BTP| \leq |ATP \cap BTP| \leq 100\%|BTP|$

We acknowledge that our proposal has a few subjective aspects, which will need to be minimized in the future. However, this is what happened to many successful proposals (e.g., in a different software subdiscipline, Function Points (Garmus & Herron, 2001)), whose initial proposal entailed a high degree of subjectivity that was reduced over the years.

The four maturity levels have a range of values that can be refined over time to normalize the values based on the results obtained from a more extensive industrial evaluations of OSS-TMM. For example, a unit of Siemens AG is experimenting OSS-TMM by assessing some internally used OSS products. Unfortunately, their results are not publicly available.

4. OSS-TMM EVALUATION

Here, we show how OSS-TMM can be used by applying it to BusyBox, Apache and TPTP. We show these three applications because of their popularity, maturity, and complexity. Also, these applications cover the most popular languages used to develop OSS, i.e., C, C/C++, and Java [http://langpop.com]. The experimentation has been conducted by applying the OSS-TMM checklist both internally (i.e., in our labs) and also externally (i.e., thanks to the help of contributors of each project). In this way, we try to confirm the results we obtained internally.

BusyBox Case Study

BusyBox [www.busybox.net] is an OSS project, developed in C, which has some of the typical properties of OSS projects. BusyBox combines tiny versions of common UNIX utilities into a single small executable, providing minimalist replacements for the utilities usually found in Linux environments. BusyBox is a fairly large project with 177000 lines of code (LOC) and a medium size community of developers that contributes to its development (30 developers).
We applied OSS-TMM to BusyBox from the developer's point of view with the goals of:
Q1 - demonstrating the simple applicability of OSS-TMM to derive the ML of the product;
Q2 - verifying whether the suggestions provided by our model can actually be used to improve the testing process of BusyBox.

To answer the first research question *Q1*, we are interested in verifying that the computation of the maturity level can be accomplished by following the checklist suggestions (i.e., steps S1, S2, S3 and S4 are supported by the checklist) with a limited effort. To answer the second research question *Q2*, we fully implemented the BTP suggested by OSS-TMM to verify whether a well-designed testing process can actually improve the quality of BusyBox.

We sequentially applied all the steps of OSS-TMM. Thus, we analyzed BusyBox by scanning and answering each entry of the checklist, we identified the BTP in relation to

the actual characteristics of BusyBox, we estimated the maturity level of the product, and finally we redefined the BusyBox ATP by following the suggestions provided by the identified BTP. For space reason, we are not able to report the complete checklist of BusyBox. Please, refer to (OSS-TMM, 2010) to download the BusyBox's checklist.

Step S1: Analysis of Issues

In Table 2, we summarize all of BusyBox's characteristics derived from the analysis of the project through the checklist.

Step S2: BTP derivation

The data collected during the previous step suggest that BusyBox is characterized by a high degree of visibility. The browsing of the source code is facilitated by the availability of a subversion system (SVN). Source files are packaged in 28 main directories and information is provided for each directory about the number of revisions, authors of the revisions, age of the latest revisions, and log entries. This facilitates the applicability of all the white-box testing techniques and a clear identification of the units that compose the entire system. The high level of modularity and the low level of interoperability among the features of BusyBox seem to suggest that developers should focus on unit testing activities.

Non-functional issues are not of primary importance for this kind of tool, due to its nature. BusyBox is a tool provided without a graphical interface, thus, approaches such as capture and replay are infeasible. The use of monitors that probe memory usage and execution/response time are not meaningful since the tool only provides calls to simple functions. Finally, since BusyBox provides replacements for most of the utilities usually found in GNU, developers should pay attention to testing security aspects. It is realistic to imagine a scenario in which a developer inserts malicious code into a BusyBox function to remotely control the operating system of an end-user that has installed BusyBox. This requires the execution of acceptance tests that check the main functionalities of the tool to allow only for trusted behaviors. For example, a test case should verify that the Unix command `su` (superuser) must not record the typed password.

As for issues I2, I3, I5, the BusyBox project is not supplied with project plans, documents that describe the system requirements analysis, risk analysis, technical documents that describe the use of standard protocols or patterns, architectural models, etc. The only standard to which developers pay attention, albeit without completely adhering to it, is the "Shell and Utilities" portion of the Open Group Base Standards. This strongly limits the applicability of all testing solutions that are based on project specifications such as model-based and conformance testing techniques. However, the web portal of BusyBox provides a section that describes all the features and functionalities offered by BusyBox in a structured way. Each feature description reports the input and output parameters, the behavior the feature should have, and how to use the feature. This fosters the applicability of black-box techniques such as Category Partition and Catalog-based testing techniques in addition to white-box testing techniques. Moreover, developers and end-users should share testing knowledge with each other. Unit, system, and regression test results should be provided to the global community to favor and simplify BusyBox's integration testing.

As for I4, BusyBox is a vital and consolidated project (latest revisions are usually only a few days old and the release analyzed is V1.14.0) supported by a medium-size community of developers (currently, 38 accounts exist). It is characterized by a collaborative development process and rapid system growth. At the time of writing, the community performed 24,110 revisions, and forums and mailing lists are still alive and useful. Also, the bug tracking system seems to indicate an active community of developers (the mean time required to fix a bug is quite short: 3 to 4 days). All of these considerations seem to suggest the need for a strong regression testing activity during the whole development process of BusyBox, to avoid bugs introduced by excessive creativity.

Summarizing, the BTP for BusyBox should take into account the following activities: (1) unit testing; (2) integration testing; (3) regression testing; (4) security testing; (5) identification and use of test management tools; (6) fault-based testing techniques; (7) acceptance/system testing; (8) documentation of test results.

Step S3: ATP analysis

Issue	BusyBox characteristic
I1.1	the whole project is managed via SVN
I1.2	the whole project is well structured in 28 folders
I1.3	information about releases are visible
I1.4	information about code revisions are visible
I1.5	sensible data are manipulated (e.g., username, pwd)
I1.6	all the scripts are open source to the community
I1.7	the popular GPLv2 license is used
I1.8	log files are not available
I2.1	the project plan/roadmap is unavailable
I2.2	the risk analysis is unavailable
I2.3	the requirements analysis is unavailable
I2.4	the goal analysis is unavailable
I2.5	UML diagrams are unavailable
I2.6	the standard "Shell and Utilities OGB" is used
I2.7	coding standards and conventions are not identified
I2.8	performance requirements are not meaningful
I2.9	BusyBox does not follow a specific architectural style
I2.10	BusyBox is designed without a GUI
I2.11	BusyBox does not integrate external libraries/plugins
I3.1	none specific development process is followed
I3.2	developers are unstructured
I3.3	a sandbox environment is not provided
I3.4	none specific IDE is used/recommended
I3.5	none specific testing platform is used/recommended
I3.6	the Bugzilla bug tracking system is integrated
I4.1	24110 revisions in total
I4.2	38 developers/contributors
I4.3	the analyzed release is: V1.14.0
I4.4	the number of new/resolved/closed bug is reported into Bugzilla
I4.5	BusyBox is a vital project
I5.1	the system-level documentation is unavailable
I5.2	the library-level documentation is unavailable
I5.3	a simple features-level documentation is available
I5.4	a short user manual is available (README file)
I5.5	bug reports are available
I5.6	the code documentation is unavailable
I5.7	documents are also disseminated via a mailing list
I5.8	installation requirements are not documented
I5.9	test documentation is unavailable

Table 2. Step S1 outcome for BusyBox

Currently, BusyBox is released along with a test suite that can be executed by end-users and developers to identify problems and bugs when BusyBox is installed on machines different from the tested ones. A quick look at the test suite suggests that developers have designed test cases to only stress each feature of BusyBox separately. Unfortunately, reports and documents that discuss the test cases and the execution results are not provided. The testing plan provided for BusyBox does not have the ability of automatically logging and collecting the test results, and users must manually signal potential bugs to the community.

Step S4: Maturity level evaluation

When comparing the BTP derived for BusyBox during step S2 and the one currently available, it is apparent that the testing activity of BusyBox is actually poor, and the huge number of bugs posted by the community may be a confirmation of this. The intersection between BTP and ATP does not exceed 25% (only 1 activity out of 8 is currently supported by the ATP), thus BusyBox has ML = 1. This suggests the need for applying all the testing techniques identified by OSS-TMM during step S2 to increase the quality of the product.
We contacted a contributor of BusyBox to have him also apply OSS-TMM to BusyBox. He derived a maturity level equal to the one we assessed (ML=1). This confirms the reliability of our evaluation.

Step 5: Testing process improvement

To improve the testing process of BusyBox, we followed the guidelines suggested by the outcome of step S2. We selected a test management tool (TestLink) to create and manage test cases and test plans, execute test cases, track test results dynamically, and generate reports. We planned integration, acceptance/system and regression testing activities, and then we generated a set of test cases for each activity. Then, we executed the test suites on a machine hosted in our lab with the following environment: Gentoo-r6 Linux distribution; Kernel 2.6.18; C compiler: gcc 4.1.2.

Table 3 reports the data collected during the execution of the test suites. Column <<*BusyBox*>> reports the version of BusyBox under test; Columns <<*#of TCs*>>, <<*Passed TCs*>>, <<*Failed TCs*>> report the total number of test cases for each test suite, the number of test cases that succeeded (i.e., that ended with no failures), and the number of test cases that failed (i.e., that resulted with the software having a failure), respectively. Acceptance and System testing has been performed by executing three different sets of test cases against three different versions of BusyBox (Table 3 (a)).
A simple example of test case is `test (pwd) = (busybox pwd)`: this test case simply verifies that the working directory returned by the operating system is equal to the one returned by BusyBox. Regression testing has been performed by re-executing the 358 successful test cases, designed for BusyBox V1.12.1, against the new stable releases V1.12.4, V1.13.0 and V1.13.2 (Table 3 (b)). When executing the test suite against BusyBox V1.12.4, none of the test cases failed. However, when executing the test suite against the BusyBox V1.13.0, three test cases which refer to the `taskset` command could not be executed because `taskset` has been removed from this version of BusyBox, and just one test case failed. The analysis of the test results allowed us to identify a new error introduced by a code change (related to the `cpio` command) in V1.13.0. The error even persisted in BusyBox V1.13.2.

Integration testing has been performed by executing 48 test cases that we designed to stress the interoperability between the BusyBox implementations of the most common UNIX utilities (e.g., `cp` command in combination with `touch` and `cmp` commands). Two test cases failed: the first one failed due to an unsupported option (`-t`) for the `od` command when piped with the `echo` command; the second one due to an unsupported option (`--date=`) for the `touch` command when piped with the `mv` command.

Acceptance/System Testing			
BusyBox	**#of TCs**	**Passed TCs**	**Failed TCs**
V1.10.1	312	291	21
V1.12.1	387	358	29
V1.13.2	390	359	31

(a)

Regression Testing			
BusyBox	**#of TCs**	**Passed TCs**	**Failed TCs**
V1.12.4	358	358	0
V1.13.0	358	354	1
V1.13.2	358	354	1

(b)

Integration Testing			
BusyBox	**#of TCs**	**Passed TCs**	**Failed TCs**
V1.13.2	48	46	2

(c)

Table 3. Test cases results for BusyBox

Experience Conclusions
This study shows evidence about the simplicity of OSS-TMM, as well as the real benefits introduced by a well planned testing process. Step S1 to Step S4 required a limited effort (one author of this paper worked two full days for this task), while Step S5 required a much more significant effort (one of the authors of this paper fully worked 1 month for this task due to the effort spent for implementing and executing the test cases). The restructured process provided the ability to detect three new errors in BusyBox with a real improvement of BusyBox's quality.

Apache Httpd Case Study

Here, we apply OSS-TMM to the Apache Httpd [http://httpd.apache.org/] project. Apache Httpd is an OSS HTTP server for operating systems such as UNIX and Windows, which provides HTTP services in sync with the current HTTP standards. Apache Httpd is a fairly large project with 135000 C/C++ lines of code (LOC) and a medium size community of developers that contributes to its development (60

Issue	Apache Httpd characteristic
I1.1	Apache is managed via a Historical Archive
I1.2	the whole project is well structured
I1.3	information about releases are visible
I1.4	information about code revisions are visible
I1.5	SSI and AAA modules are security critical
I1.6	all the test scripts are open source to the community
I1.8	access log, error log files are collectable
I2.7	coding standards are specified in a style guide
I2.8	performance constraints: resource usage, latency, throughput, scalability
I2.11	external modules: mod python, mod ftp, mod mbox
I3.5	The Apache-Test Framework is recommended
I3.6	the Bugzilla bug tracking system is integrated
I4.1	the number of revisions is huge
I4.2	more than 60 developers/contributors (from: http://httpd.apache.org/contributors/)
I4.3	the analyzed release is: V2.2.11
I4.4	the number of new/resolved/closed bug is reported into Bugzilla
I5.8	installation requirements are documented (but not up-to-date)
I5.9	detailed test documentation is unavailable

Table 4. Step S1 outcome for Apache Httpd

developers).

We applied OSS-TMM to Apache Httpd from the end-user's point of view with the aim of demonstrating how a non-skilled user can derive the maturity level of the product he is evaluating. As in the previous case study, we sequentially applied all the steps of our OSS-TMM, with the exception that we focused on the steps followed by a end-user interested in evaluating the Apache Httpd product. A PhD student not involved directly in this work did the assessment. Hence, we first analyzed Apache Httpd through our checklist (step S1), we identified the BTP in relation to the actual characteristics of Apache Httpd (step S2), we analyzed the current testing activities for Apache Httpd (step S4), and we estimated the maturity level of the product by comparing the BTP and the Apache ATP (step S4). For space reason, we are not able to report the complete checklist of Apache Httpd. Please, refer to (OSS-TMM, 2010) to download Apache's checklist.

Step S1: Analysis of Issues

In Table 4, we summarize all the Apache Httpd characteristics derived from the analysis of the project through the checklist.

Step S2: BTP derivation

The data collected during the previous step show that Apache Httpd is managed via a historical archive well structured in packages and folders for each version of Apache Httpd [http://archive.apache.org/dist/httpd]. For each directory the age of the latest revision, the size of the folder and a description is provided. This facilitates the applicability of all the white-box testing techniques and a clear identification of the units that compose the entire system. Due to the high number of packages and folders, testers should pay attention to unit, integration and regression testing activities. Apache Httpd exports two modules (SSI and AAA) that are security critical and thus require special attention when testing them. Apache Httpd provides the ability of collecting error log files, thus simplifying the applicability of Dynamic analysis techniques. Coding standards are defined in a style guide thus requiring code inspection to verify the adherence to [http://httpd.apache.org/dev/styleguide].

Apache Httpd has a lot of performance constraints to take under control, such as the usage of resources, latency and throughput, thus requiring a strong activity for performance testing. Another important aspect that characterizes the Apache Httpd project is the use of third-party modules (e.g., *mod_python, mod_ftp, mod_mbox*). It is important to periodically verify the compliance of new Apache versions with the integrated modules. As for I4, Apache Httpd is a vital and consolidated project (latest revisions are usually only a few days old and the release we analyzed is V2.2.11) supported by a medium-size community of developers (currently, 60 contributors). It is characterized by a collaborative development process and rapid system growth, and forums and mailing lists [http://httpd.apache.org/lists] are still alive and useful. Also, the bug tracking system [http://httpd.apache.org/bug_report]

Issue	TPTP characteristic
I1.1	TPTP is managed via CVS (http://dev.eclipse.org/viewcvs/)
I1.2	the whole project is well structured
I1.3	information about releases are visible
I1.4	information about code revisions are visible
I1.6	all the test scripts are open source to the community (see CVS)
I1.7	TPTP integrates several OSI licenses (www.eclipse.org/tptp/home/project_info/releaseinfo/4.7/Open_Source.htm)
I2.1	the project roadmap is available (www.eclipse.org/projects/project-plan.php?projectid=tptp)
I2.5	system diagrams are available
I2.6	standard protocols are used (www.eclipse.org/tptp/platform/documents/)
I2.7	coding standards are specified (http://wiki.eclipse.org/index.php/Development_Conventions_and_Guidelines)
I2.9	TPTP uses a specific architectural style (www.eclipse.org/projects/dev_process/development_process_2010.php)
I2.10	TPTP provides a GUI
I2.11	a lot of external modules are used
I3.1	a specific development process is used
I3.2	developers are structured in teams
I3.5	the Eclipse IDE is used
I3.6	the Bugzilla bug tracking system is integrated
I4.1	the number of revisions is huge
I4.2	more than 60 developers/contributors (from: http://dash.eclipse.org)
I4.3	the analyzed release is: V4.7.0
I4.4	the number of new/resolved/closed bug is reported into Bugzilla
I5.1, I5.2, I5.3, I5.4, I5.5, I5.6, I5.8	TPTP documentation is available (www.eclipse.org/tptp/platform/documents/)
I5.9	test documentation is unavailable

Table 5. Step S1 outcome for TPTP

seems to indicate an active community of developers. All of these considerations seem to suggest the need for a strong regression testing activity during the whole development process of the Apache Httpd server, to avoid bugs introduced by excessive creativity. The project is well documented and planned.

Summarizing, Apache Httpd's checklist suggests a BTP that takes into account the following testing aspects: (1) unit testing; (2) integration testing; (3) regression testing; (4) security testing; (5) dynamic analysis techniques; (6) source code inspection through checklists for C/C++; (7) performance testing; (8) versioning compatibility checks; (9) use of test management tools; (10) fault-based testing techniques; (11) acceptance/system testing and installation testing; (12) pay attention to documentation and sharing of test results.

Step S3: ATP analysis

The end-user can surf the web portal of Apache Httpd to discover that Apache Httpd is currently released with a small test suite for testing the critical features of the project, and supports the *SPECWeb99* benchmark, *Flood* subproject, and the *Apache-test* framework. *SPECWeb99* and *Flood* can be used to gather important performance and security metrics for websites that use Apache Httpd. The *Apache-test* framework supports the definition of test suites for products running on the Apache Httpd, and can be used to run existing tests, setup a testing environment for a new project, and develop new tests. However, a complete test suite for integration and regression testing is not provided and source code inspection, and versioning compatibility checks are not yet performed.

Step S4: Maturity level evaluation

Comparing the BTP derived for Apache Httpd during step S2 and the one currently available ATP, it is clear that the testing activity of Apache Httpd is good and well planned. The testing aspects (1), (4), (7), (9), (11) and (12) of BTP are addressed by the ATP of Apache Httpd, thus implying an intersection between ATP and BTP equal to 50% and a ML = 3.

Experience Conclusions

Limited effort was required to accomplish all the steps (from S1 to S4): one PhD student, not directly involved in this work, took four days for this task. This case study demonstrates the support provided by the checklist, which greatly simplifies the analysis by suggesting step-by-step activities that non-skilled people can follow to determine the maturity level of the product under evaluation.

Moreover, we asked Apache contributors to fill in our checklist. A contributor of the Apache project applied OSS-TMM to Apache Httpd and he derived a maturity level equal to the one we assessed (ML=3) with a slightly different ATP/BTP ratio (67%). The contributor reported the availability of integration and regression test suites. The evaluation took 3 hours. The two evaluations confirmed the reliability of our experience.

TPTP Case Study

Here, we apply OSS-TMM to the TPTP [www.eclipse.org/tptp/] project. TPTP is an OSS platform that allows software developers to build test and performance tools that can be easily integrated with the platform. TPTP addresses the entire test life cycle, from early testing to production testing, including test editing and execution, monitoring, tracing and profiling, and log analysis capabilities.

TPTP is a large Java project with 200000 lines of code and a medium size community of developers that contributes to its development (60 developers).

Like in the previous case studies, we applied sequentially all the steps of OSS-TMM, from S1 to S4 and an author of this paper did the assessment. We first analyzed TPTP through our checklist (step S1), we identified the BTP in relation to the actual characteristics of TPTP (step S2), we analyzed the testing activities for TPTP to derive ATP (step S3), and we estimated the maturity level of the product by comparing the BTP and the TPTP ATP (step S4). For space reason, we are not able to report the complete checklist of TPTP. Please, refer to (OSS-TMM, 2010) to download the TPTP's checklist. The TPTP characteristics derived from the analysis are summarized in Table 5, which only lists the issues that characterize TPTP and are useful to derive the BTP.

Starting from the TPTP characteristics highlighted in step S1, the best testing practices BTP for TPTP should take into account the following testing aspects: (1) unit testing; (2) integration testing; (3) regression testing; (4) licenses compatibility check; (5) black-box testing; (6) model-based testing; (7) conformance testing; (8) style check and inspection; (9) testing for architecture; (10) capture&replay; (11) usability testing; (12) versioning compatibility checks; (13) testing process check; (14) IDE potentialities exploitation; (15) fault-based testing; (16) improve integration/regression testing and test cases documentation; (17) documentation use for testing; (18) test management tool.

Analyzing TPTP's source code and the repository of the project, we found that TPTP is currently released with test suites for unit, integration, regression and performance testing. A complete list of the licenses used by the different components of TPTP can be found in the *project_info* section of the TPTP portal. A test suite that uses the *TPTP Automated Gui Recorder* is also available and attention is paid to versioning compatibility checks. Test reports are available in the CVS repository and test cases are documented. Concluding, the testing aspects (1), (2), (3), (4), (10), (12), (13), (14) and (16) of BTP are addressed by the ATP of TPTP, thus implying an intersection between ATP and BTP equal to 50% and a ML = 3.

Experience Conclusions

The internal assessment took 4 hours. A contributor of TPTP was also asked to apply OSS-TMM to TPTP and he derived a maturity level equal to ML=3 with a slightly different ATP/BTP ratio (56%). The contributor reported the availability of a test management tool for testing documentation that we were not able to find during our analysis. The external assessment took 3 hours. At any rate, the two evaluations confirmed the reliability of this experiment.

Maturity Level vs Bug Rate

We looked for correlation patterns between the maturity levels obtained and the quality of the three OSS projects, to comprehend whether high maturity of the testing process directly may imply high product quality , and low maturity low product quality. As for the dependent measure, we decided to select the bug rate (*BR*) of the product (i.e., the number of bugs divided by product size in thousands of lines of code), which is a sensible indicator of the overall product quality. Table 6 summarizes the results obtained. Bug data have been collected by analyzing the bug tracker system of each product with focus on *new*, *resolved* and *closed* bugs. Following the Bugzilla's documentation [www.bugzilla.org] the *new* status is for bugs that have been successfully reproduced by a member of the team; the *resolved* status is for bugs that are

fixed by coding a solution that is released as an official patch; the *closed* status is for bugs that have been fixed and whose fixes have appeared in a release of the project. We selected the latest stable release of each product to avoid strange bug distributions related to newly released and unstable products, and we just considered the bugs that refer to the selected releases. We used SLOCCount (developed by D. Wheeler) for counting the physical source lines of code of each project [www.dwheeler.com/sloccount/]. We limited this study to BusyBox, Apache Httpd and TPTP simply because the other projects we evaluated with OSS-TMM (see Table 7) have bug tracker systems that are different from each other, thus introducing heterogeneity in the collected bug data. BusyBox, Apache Httpd, and TPTP use the Bugzilla V3.6 bug tracker system, thus ensuring homogeneity on the bug data reported. An analysis of Table 6 provides some evidence that a high maturity level impacts both on the ability of detecting new bugs (i.e., the bug rate value related to new bugs proportionally increases) and the ability of resolving and closing the newly discovered bugs (i.e., the resolved/closed bug value increases).

Project	ML (ATP/BTP)	Bug Rate
BusyBox v1.14.0	1 (14%)	New: 0,006 Resolved: 0,19 Closed: 0,00
TPTP v4.7.0	3 (50%)	New: 0,037 Resolved: 0,07 Closed: 0,83
Apache Httpd v.2.2.0	3 (55%)	New: 0,096 Resolved: 0,93 Closed: 0,06

Table 6. OSS-TMM MLs vs Bug Rate

Other OSS-TMM Assessments

We also applied OSS-TMM to four additional OSS projects. We selected the projects by evaluating their size, organizational type (i.e., sponsored, foundation, spontaneous), and diffusion to identify a heterogeneous set of OSS projects. We selected: the Debian distribution (DebianOS) as a well-known, sponsored, and complex OSS product; the Data Display Debugger (DDD) as an unfamiliar, sponsored, and of reduced complexity project; the OSS database PostgreSQL as a specialized, sponsored, and complex product; the web content management system (Xoops) as a specialized, founded, and of reduced complexity project. In our evaluation, the four OSS projects obtained the maturity levels listed in Table 7. PostgreSQL has a very mature testing process with a complete test suite and an updated documentation, while the testing processes of the other projects are still in their infancy and need more attention.

Project	OSS-TMM Maturity Level
DDD v3.3	ML=1
DebianOS v3.0	ML=1
PostgreSQL v8.3	ML=4
Xoops Core v2.3	ML=2

Table 7. OSS-TMM MLs for four OSS projects

5. RELATED WORK

Here, we compare our approach with what has been done in some related research areas that address software quality.

Software Process Improvement

Research in software process improvement focuses on certification models that deal with the quality of the software production process. The most important models are CMM and SPICE (Herbsleb et al., 1997; ISO/IEC15504, 2004). The Capability Maturity Model (CMM), and its extension CMMI, is a methodology that assists companies in understanding the capability maturity of their software processes. The maturity model involves several aspects related to five maturity levels (chaotic, repeatable, defined, managed, and optimizing), a cluster of Key Process Areas (KPA) (i.e., related activities that, when performed collectively, achieve a set of important goals), a set of goals (i.e., scope, boundaries, and intent of each key process area), common features (i.e., practices that implement a KPA), and finally key practices (i.e., the elements that effectively contribute to the implementation of the KPAs).

The Software Process Improvement and Capability dEtermination (SPICE / ISO15504) is a framework for the assessment of processes. The SPICE reference model focuses on a wider vision than CMM by taking into account five process and capability dimensions (customer-supplier, engineering,

supporting, management, and organization). In compliance with CMM, they define a scale of capability levels, a cluster of process attributes (to measure capability of processes), a set of generic practices (i.e., indicators to aid assessment performance), and a process assessment guide.

Our approach is built upon the general ideas proposed by CMM and SPICE. However, OSS-TMM uses a simpler and less rigid maturity model than CMM and SPICE, because of its purpose and its focus on the testing dimension. Moreover, the OSS-TMM process assessment also suggests how to improve the available testing process of an OSS product by recommending the most suitable testing techniques.

Software Product Quality

The most important standard to ensure the quality of the product is ISO9126 (ISO/IEC9126, 2001). The ISO9126 standard takes into account several aspects of the internal, external, and in-use quality of a software product and it defines a quality model that includes a set of characteristics and sub-characteristics related to functionality, reliability, usability, maintainability, efficiency, and portability. In ISO9126, a wide set of complex measures are defined to assess product quality, while ISO14598 (ISO/IEC14598, 2001) provides, among other things, an explanation of how to apply the ISO9126 model.

Our approach focuses on the quality of the testing process instead of the whole product quality and it simplifies the evaluation of the process maturity by providing a checklist instead of a complex list of measures. The steps that compose the OSS-TMM process assessment are compliant with the guidance and requirements for software evaluation highlighted in ISO14598.

Testing Maturity Models

Research in testing maturity models complements CMM with the focus on testing aspects. The first work on this research area is provided by Burnstein et al. in (Burnstein et al., 1996). They defined a Testing Maturity Model (TMM) that helps evaluate the testing process of software products. TMM identifies five rigid maturity levels, a set of maturity goals and sub-goals (equivalent to KPAs of CMM), and a set of activities, tasks and responsibilities (ATR) for each maturity level.

Other CMM-based testing models have been proposed. For example, the Test Improvement Model (TIM) (Ericson et al., 1997) and the Test Process Improvement Model (TPI) (Koomen & Pol, 1999) suggest ways in which testers can improve their work. TIM and TPI identify key areas for the testing process starting from the organization and planning of testing activities to test cases generation, execution, and documentation review. While the previous approaches have been designed with CSS characteristics in mind, OSS-TMM exploits the inherent characteristics and issues typical of OSS products. Hence, OSS-TMM defines four maturity levels that are not structured as sets of predetermined maturity characteristics and goals, but they depend on the actual characteristics of the product under evaluation. Moreover, OSS-TMM supports both testers in improving the testing process and also companies and end-users in assessing the quality and the trustworthiness perception of the OSS product.

OSS Quality Assessment

Research in OSS quality assessment extends CMM and CMM-compliant models to identify, from the set of CMM goals, only the subset that is relevant for OSS products. The first CMM extension for OSS is the Open Source Maturity Model (OSMM) (Duijnhouwer & Widdows, 2009). OSMM defines a methodology and a set of OSS ad-hoc indicators to assess the global maturity of an OSS product, helping end-users to choose between equivalent OSS products. Since the definition of OSMM, several other models have been developed (see as example (Taibi et al., 2007)). Recently, the Open Maturity Model (OMM) has been defined as an output of the QualiPSo project. OMM has been designed specifically for the Free/Libre Open Source software development process evaluation. The structure of the model resembles in many aspects the Capability Maturity Model (Petrinja et al., 2009) and OMM is compatible with CMM.

OSS-TMM does not provide a global assessment of the product quality but uses the testing process maturity level as an indicator of the process quality. This simplifies the applicability of the approach and the

identification of weaknesses into testing processes.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a set of testing guidelines that are *ad-hoc* for the inherent characteristics of OSS projects. The identification of the best testing technologies, which make most sense for the OSS products under test, is supported by a checklist of issues and testing guidelines. The usage of the guidelines and the checklist is merged in the OSS-TMM method that helps developers in assess and improve the testing process of their OSS products. Applications to BusyBox, Apache Httpd and TPTP show how the method comes into play on real-life projects. The method is currently under evaluation in Siemens AG to assess some internally used OSS products. The results of this adoption will be the basis for refinement of OSS-TMM. We believe that continuous gathering and analysis of experiences will help pinpoint specific issues of OSS testing and better address the building of a more useful OSS-TMM. In addition, it will help reduce at least some of the subjectivity of OSS-TMM, in the same way as happened with other models in the past.

ACKNOWLEDGMENTS

The research presented in this article was partially funded by the IST project QualiPSo [http://www.qualipso.org/], sponsored by the EU in the 6th FP (IST-034763); the FIRB project ARTDECO, sponsored by the Italian Ministry of Education and University; and the projects “Metodi e tecniche per la modellazione, lo sviluppo e la valutazione di sistemi software” and “La qualità nello sviluppo software,” funded by the Università degli Studi dell’Insubria. We also acknowledge the developers of BusyBox, Apache Httpd and TPTP for their evaluations.

REFERENCES

IEEE Std 1008-1987 (1986), IEEE standard for Software Unit Testing.

IEEE Std 829-2008 (2008), IEEE Standard for Software and System Test Documentation

ISO/IEC 14598-1 (2001), Information technology - software product evaluation. Part 1: General overview. Int. Organization for Standardization.

ISO/IEC 9126-1 (2001), Software engineering - product quality. Part 1: Quality model. Int. Organization for Standardization.

ISO/IEC 15504-1 (2004), Information technology process assessment. Part 1: Concepts and vocabulary. Int. Organization for Standardization.

Aberdour, M.. 2007. Achieving Quality in Open Source Software. In *IEEE Software,* 24(1), pp. 58-64.

Agruss, C. (2000), Software installation testing - how to automate tests for smooth system installation. In *Software Testing & Quality Engineering*, pp. 32-37.

Arkin, B., Stender, S. & McGraw G. (2005) Software penetration testing. In *IEEE Security and Privacy*, 3(1), pp. 84-87.

Bernhard, P.J. (1994), A reduced test suite for protocol conformance testing. In *ACM Trans. Softw. Eng. Methodol (TOSEM)*, 3(3), pp. 201-220.

Burnstein, I., Suwanassart, T., & Carlson R. (1996), Developing a Testing Maturity Model for software test process evaluation and improvement. In *Proceedings of the IEEE International Test Conference (ITC)*, pp. 581-589.

Dannenberg, R.B., & Ernst, G.W. (1982), Formal program verification using symbolic execution. In *IEEE Transactions on Software Engineering (TSE)*, 8(1), pp. 43-52.

Duijnhouwer, F.W. & Widdows C. (2010), Open Source Maturity Model. Web published: www.osspartner.com. Accessed Oct. 2010.

El Emam, K. (1997), Spice: The Theory and Practice of Software Process Improvement and Capability dEtermination. In *IEEE Computer Society Press*.

Ericson, T., Subotic, A., & Ursing S. (1997), TIM - a Test Improvement Model. In *International Journal on Software Testing, Verification and Reliability*, 7(4), pp. 229-246.

Ernst, M.D., Cockrell, J., Griswold, W.G., & Notkin D. (2001). Dynamically discovering likely

program invariants to support program evolution. In *IEEE Transactions on Software Engineering (TSE)*, 27(2), pp. 99-123.

Fagan, M.E. (1986), Advances in Software Inspections, In *IEEE Transactions on Software Engineering (TSE)*, 12(7), pp. 744-751.

FOSSology Project (2010). Web published: http://sourceforge.net/projects/fossology/. Accessed: March 2011.

Garmus, D., & Herron D. (2001), Function point analysis: measurement practices for successful software projects. In *Addison-Wesley Longman Publishing Co., Inc.*, Boston, MA, USA.

Herbsleb, J., Zubrow, D., Goldenson, D., Hayes, W., & Paulk, M. (1997), Software quality and the Capability Maturity Model. In *Communications of the ACM*, 40(6), pp. 30-40.

Howard, M. (2006), A process for performing security code reviews. In *IEEE Security and Privacy*, 4(4), pp. 74-79.

Jacareto tool home page (2010). Web published: http://jacareto.sourceforge.net. Accessed: March 2011

Koomen, T. & Pol, M. (1999), Test Process Improvement: a practical step-by-step guide to structured testing. In *Addison-Wesley Longman Publishing Co., Inc.*, Boston, MA, USA.

Leung, H.K.N. & White, L. (1990), A study of integration testing and software regression a the integration level. *In Proceedings of the Conference on Software Maintenance (ICSM)*.

Mao, C., Lu, Y., & Zhang, J. (2007), Regression testing for component-based software via built-in test design. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pp. 1416-1421.

Morasca, S., Taibi, T., & Tosi, D. (2009), Certifying the testing process of open source software: New challenges or old methodologies? In *Proceedings of the IEEE International Workshop on Free/Libre/Open Source Software (FLOSS)*. Colocated with ICSE.

Morasca, S., Taibi, D., & Tosi. D. (2010), T-DOC: a Tool for the Automatic Generation of Testing Documentation for OSS Products. In *Proceedings of the IFIP International Conference on Open Source Software (OSS)*.

Morell, L.J. (1990), A theory of fault-based testing. In *IEEE Transactions on Software Engineering (TSE)*, 16(8), pp. 844-857.

Nielsen, J., (1999), User Interface Directions for the Web, *Communications of the ACM*, 42(1), pp. 65-72.

Open Source Testing Tools (2010). Web published: www.opensourcetsting.com. Accessed: March 2011.

Orso, A., (2010), Monitoring, analysis, and testing of deployed software. *In Proceedings of the FoSER workshop*, pp. 263-268.

OSLC Open Source License Checker V3. Web published: http://forge.ow2.org/projects/oslcv3/. Accessed: March 2011.

OSS-TMM Web Page (2010), Web published: http://qualipso.dscpi.uninsubria.it/web/oss-tmm/. Accessed: March 2011.

Ostrand, T.J., & Balcer, M.J. (1988), The category-partition method for specifying and generating functional tests. In *Communications of the ACM*, 31(6), pp. 676-686.

Petrinja, E., Nambakam, R., & Sillitti, A. (2009), Introducing the OpenSource Maturity Model. In *Proceedings of the IEEE International Workshop on Free/Libre/Open Source Software (FLOSS)*. Colocated with ICSE.

Pezzè, M., & Young, M. (2007), Software Testing And Analysis. Process, Principles, and Techniques. *Wiley*.

PMD Tool (2010). Web published: http://pmd.sourceforge.net/. Accessed: March 2011.

Pretschner, A., Prenninger, W., Wagner, S., Kuhnel, C., Baumgartner, M., Sostawa, B., & Zolch, R. (2005), One evaluation of model-based testing and its automation. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pp. 392-401.

Santelices, R.A., Chittimalli, P.K., Apiwattanapong, T., Orso, A., and Harrold, M.J. (2008). Test-suite augmentation for evolving software. In *IEEE International Conference on Automated Software Engineering (ASE)*, pp. 218–227.

The Qualipso Project 1 (2010). Web published: www.qualipso.eu/node/45. Accessed: March 2011.

Taibi, D., Lavazza, L., & Morasca, S. (2007), OpenBQR: a framework for the assessment of OSS. In *International Journal on Open Source Development, Adoption and Innovation*, pp. 173-186.

Tosi, D., & Tahir, A. (2010), How developers test their Open Source Software Products. A survey of well-known OSS projects. In *Proceedings of the 5th International Conference on Software and Data Technologies (ICSOFT)*.

Tuunanen, T., Koskinen, J., & Kärkkäinen, T. (2009), Automated software license analysis. In *International Journal on Automated Software Engineering,* 16(3), pp. 455-490. Springer.

Weyuker, E.J., & Vokolos, F.I. (2000), Experience with performance testing of software systems: issues, an approach, and case study. In *IEEE Transaction on Software Engineering (TSE)*, 26(12), pp. 1147-1156.

Zhao, L., & Elbaum, S. (2003), Quality assurance under the open source development model. In *International Journal of Systems and Software*, 66(1), pp. 65-75.