An Approach to Establish a Software Reliability Model for Different Free and Open Source Software Development Paradigms

Kemal Kağan Işıtan

University of Tampere
School of Information Sciences
Computer Science
M.Sc. thesis
Supervisor: Eleni Berki
May 2011

University of Tampere
School of Information Sciences
Computer Science
Kemal Kağan Işıtan: An Approach to Establish a Software Reliability Model for Different Free and Open Source Software Development Paradigms
M.Sc. thesis, 42 pages, 3 index and appendix pages
May 2011

Abstract

In the modern software development world, the two universally accepted and followed ways of software development are proprietary and open source. Both these two different software development types have one very crucial element in common: the success of the end product. Even though the business success of open source can be judged in terms of profit or loss, the developed software and the project itself are subject to criteria such as reliability, maintainability and security as success indicators. A software project and the developed software can be called reliable if the end product can satisfy the requirements of metrics measuring these criteria.

In this thesis work, different reliability models and current proposed metrics for OSS development are studied in order to establish a software reliability model for Free and Open Source Software development. Based on the literature review, different reliability models and metrics are compared and contrasted and a classification of different open source paradigms is presented. These findings are also analyzed and verified on different case studies on Open Source Software reliability evaluation.

Key words and terms: open source, free and open source software, reliability, reliability metrics, reliability models

Contents

1. Introduction

1.1. Problem Background

In the modern software development world, the two universally accepted and followed ways of software development are proprietary and open source. In terms of quality and market share evaluations, reports show that some open source software products beats the similar closed source or commercial equivalents [Wheeler 2007]. Both these two different software development types have one very crucial element in common: the success of the end product. The success indicators of the two different development types both share elements between each other and also have very distinct characteristics which cannot be applied to the counterpart. The Open Source Software development suffers more when the success indicators are tried to be evaluated and measurement metrics are applied because of the nature of its development.

The main characteristic of Open Source Software (OSS) is that the end product's source material can be accessed by anyone who has the right of using the software. In addition to this, OSS programs are programs whose licenses give users the freedom to run the program for any purpose, to study and modify the program, and to redistribute copies of either the original or modified program without having to pay royalties to previous developers [Wheeler, 2007]. However, these are not the only criteria of the open source definition. The Open Source Initiative constructed the Open Source Definition as follows [OSI, 1998]:

Free Redistribution

The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.

Source Code

The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed.

Derived Works

The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software. The mere ability to read source isn't enough to support independent peer review and rapid evolutionary selection. For rapid evolution to happen, people need to be able to experiment with and redistribute modifications.

Integrity of The Author's Source Code

The license may restrict source-code from being distributed in modified form *only* if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.

No Discrimination Against Persons or Groups

The license must not discriminate against any person or group of persons. In order to get the maximum benefit from the process, the maximum diversity of persons and groups should be equally eligible to contribute to open sources.

No Discrimination Against Fields of Endeavor

The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.

Distribution of License

The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.

License Must Not Be Specific to a Product

The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution.

License Must Not Restrict Other Software

The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software.

License Must Be Technology-Neutral

No provision of the license may be predicated on any individual technology or style of interface. Conformant licenses must allow for the possibility that redistribution of the software will take place over non-Web channels that do not support click-wrapping of the download, and that the covered code (or re-used portions of covered code) may run in a non-GUI environment that cannot support popup dialogues.

The Open Source Definition can be extended to define two other alternative terms in the Open Source field, namely Free/Open Source Software (FOSS) and Free/Libre/Open Source Software (FLOSS). The *free* here in these terms refers to the freedom given to the user to share, study and modify the source code of the software. OSS/FOSS/FLOSS also includes products which are also free in terms of price. However in this thesis Free and Open Source Software or OSS denotes the freedom of the user.

While the markers of success in any business are profit and loss, it is very hard to argue that the success of Open Source Software is dependent on the profit or loss of the development community [Metcalfe, 2004]. Even though the business success of open source can be judged in terms of profit or loss, the developed software and the project itself are subject to criteria such as reliability, maintainability and security as success indicators. A software project and the developed software can be called reliable if the end product can satisfy the requirements of metrics measuring the above criteria. Hence we can identify the first important aspect of software success as the reliability models and metrics. However, in the Open Source Software development there are no commonly accepted models or metrics compared to the proprietary software equivalents.

In the absence of such a set of OSS reliability models and metrics which are proven to be accurate for every OSS project, there is a question how to measure the success in different ways. One general conclusion is that an OSS project can be seen to have failed when another project uses its code and advances the development of the software. If the project has a very active development community, further developments based on the current code will be expected and the initial project will fail. The irony here is that we can consider the initial project a successful one which attracted many developers [Metcalfe, 2004]. Projects can prepare themselves for these failures by supplying sufficient comments in the code for an easier understanding by further developers or for documenting the history of the code's development. This introduces the second important aspect of open source software reliability, which is the developer community.

After combining the reliability models and metrics with the developer community aspect of open source software development, one may wonder whether there are any open source software development models or metrics which can satisfy the majority of the developer community. In addition, even if there are proven ways of producing reliable open source software, how can we know that different methods or metrics will work in harmony with each other? This study will therefore concentrate on the prospect of open source software reliability by following a unified model considering different aspects.

1.2. Research Questions

The main research question of this thesis is:

> **Is it possible to establish a software reliability model for Free and Open Source Software development?**

In order to answer this general question, we need to propose some sub-questions and narrow down the research. The following questions will mainly define the focus of this thesis:

a) What are the advantages of using a software reliability model and reliability metrics for Open Source Software development?

b) Is it possible to evaluate a proposed reliability model on a specific case? Are there any secondary factors which affect software reliability and quality for Open Source Software projects?

1.3. Research Methods

In this thesis, the aim is to answer the proposed sub-questions first and finally try to construct a conclusion for the main research question. Therefore for every sub-question the following research methods will be followed:

a) The first sub-question will be answered by performing a literature study. Different reliability models and current proposed metrics for OSS development will be studied.

b) Both the first and the second sub-question will be answered by performing a literature study and following a classification method by doing an analysis

of the current OSS development paradigms. In this sub-question the source of the problems in Open Source Software development will be the basis of the classification.

c) The second sub-question will also be answered by analyzing case studies. In this sense, different approaches about Open Source Software reliability evaluation will be compared and contrasted and the different problems found in question two will be focused on.

1.4. Structure of the Thesis

This thesis will be structured in the following way. Chapter 2 will cover the current state of art about the paradigms of Open Source Software development. The concept of Software Reliability will be presented. Different reliability metrics and models will be discussed and their theories and terminology will be presented.

In Chapter 3, different Open Source Software development paradigms and problems will be analyzed and classified into different categories. This categorization will be based on the nature of the different OSS development paradigms.

In Chapter 4, different case studies, possible research agendas and suggestions about Open Source Software reliability evaluation will be discussed and an analysis will be given concerning the research questions and classification in Chapter 3.

Finally, in Chapter 5 conclusions of this thesis will be discussed, including an examination of the limitations of the study as well as a presentation of possible further work in this area.

2. State of the Art

2.1. Software Reliability

A successful product must have many attributes including functionality, usability, capability, performance, documentation, maintainability and reliability. Reliability is essentially being able to deliver usability of services while assuring the constraints of the system and is considered a part of quality assurance.

For the last 60 years people became more dependent on computer systems and the benefits they offer. Almost every electronic device has an embedded computer application or a fragment of software inside. All these devices take place within the different industries and sectors such as communication, transportation, health, security, construction or governmental organizations. While the demand on computers increases day by day the complexity of the computer systems, especially the embedded software, also grows. This growth of complexity also increases the risk and effect of consequences of a possible failure of the computer system. In the past we can see drastic examples of software failure causing inconvenience, economic damage or even loss of life. While trying to emphasize the need for reliable software, Lyu points out the following examples for each failure type [Lyu, 1996]:

- In the NASA Voyager project, the Uranus encounter was in jeopardy because of late software deliveries and reduced capability in the Deep Space Network.
- The ozone hole over Antarctica would have been detected sooner if a data analysis program had not suppressed the anomalous data because it was “out of range”.
- The radiation therapy machine “Therac-25” had a perfect safety record until software errors in its complex control systems malfunctioned and claimed several patients’ lives in 1985 and 1986.
- On October 26, 1992, the Computer Aided Dispatch system of the London Ambulance Service broke down right after its installation, making the world’s largest ambulance service completely jammed.

As the examples signify the importance of reliable software, software companies recognize the tremendous need for systematic approaches using software reliability engineering techniques [Lyu, 1996]. In relation to these techniques Lyu expresses the software reliability in terms of three identifiers:

1) Probability of failure-free operation over a specified time interval
2) Mean time to failure (MTTF), the predicted elapsed time between inherent failures of a system during operation
3) Expected number of failures per unit time interval (failure intensity)

The definitions of the above identifiers are further analyzed in the following chapter. What these identifiers share in common is that one can try to analyze the reliability of software, both open source or proprietary, in terms of related analysis associated to any of the identifiers. IEEE standards further connect them into a single definition of reliability as "*the ability of a system or component to perform its required functions under stated conditions for a specified period of time"* [IEEE 610.12-1990] where the system or component denotes software in software engineering terminology.

In order to achieve software reliability, different techniques for measurement have been developed. Their main purpose is to test the software and measure the reliability according to the predefined criteria of the techniques. The final result then offers the software developers or users an understanding of the reliability of that software. This process is known as the Software Reliability Engineering and can be summarized as the following diagram:

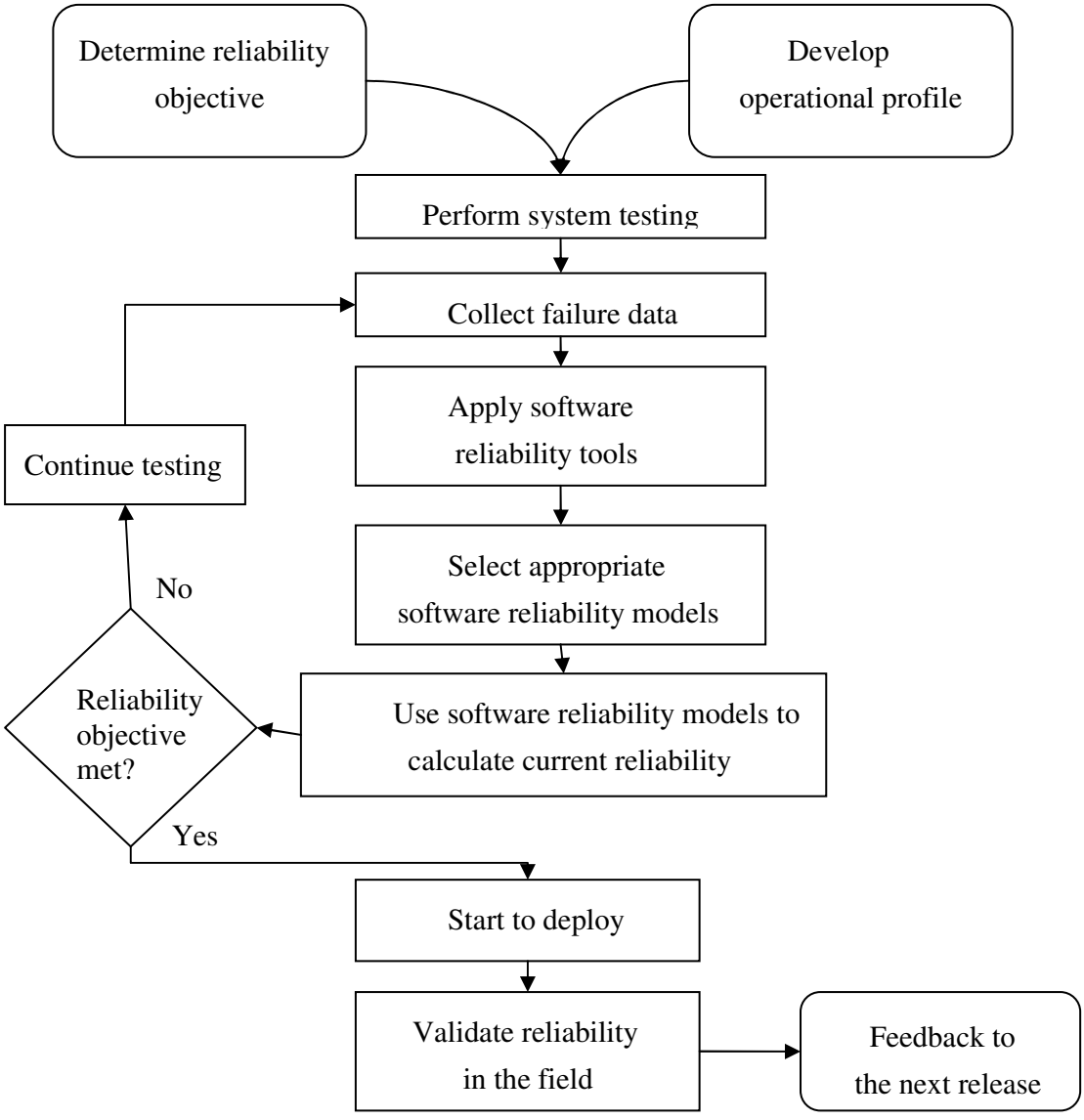

Figure 1. Software reliability engineering process overview [Lyu, 1996]

In addition to measurement, there are models for software reliability which define how software should be developed while sustaining reliability. The models are based on some assumptions, state and failure based analysis and mathematical derivations indicating reliability. Examples are the Basic Markov Model and the Goel-Okumoto Model [Musa, 1980; Xie et al., 2004]. The problems with these models are that assumptions are very hard to verify, they cannot be applied to specific software and there is no specific classification for them.

Different characteristics of the above models will be discussed after introducing the basic reliability concepts commonly used in different models.

2.2. Basic Reliability Concepts

Wasserman [Wasserman, 2002] gives a modern definition of reliability including time, condition and customer matters: "*Reliability is the probability of a product performing its intended function over its specified period of usage, and under specified operating conditions, in a manner that meets or exceeds customer expectations.*".

What happens when a product cannot perform its intended function is called a failure. Wasserman [Wasserman, 2002] also groups the possible reasons for product failures into three groups:

1. Design deficiencies
   - Omitting an important customer requirement or design feature.
   - Deficiencies in product design, which lead to early failures.
   - The design of the process can also have deficiencies, resulting in a defective product.

2. Quality control
   - Due to quality-control problems, inefficient or unsuccessful products which lead to performance problems while being used.
   - Possible damage to products while handling or distribution.

3. Misuse
   - Possible misuse of the product by customer or during service.

In addition, the possible failure outcomes define the reliability levels of products. For example, spacecraft or flight control systems will need higher reliability levels compared to simple software for personal usage. So for different projects different reliability issues should be considered and proper methods should be applied. This paper mainly focuses on software reliability and the word *system* indicates a software system or a software product.

In mathematical terms the reliability function R(t) is the probability of a system performing its function without failure in the interval 0 to t. There can be a predefined random value for the time-to-failure of that system. We can denote this function as R(t).

$$R(t) = P(T > t),\ t \geq 0$$

We can also define the failure probability of a system as F (t).

$$F(t) = 1 - R(t) = P(T \leq t)$$

If the random variable T has a density function f (x) then we can calculate R(t).

$$R(t) = \int_{t}^{\infty} f(x)dx$$

Then we can define the expected time to next failure, in other words mean time to failure (MTTF).

$$MTTF = \int_{0}^{\infty} tf(t)dt = \int_{0}^{\infty} R(t)dt$$

In almost all reliability models, these formulas are used in the mathematical representation of failure probabilities and rates. Then based on these calculations the overall reliability of the system is analyzed with further measurement criteria depending on the model chosen.

2.3. Common Techniques in Reliability Analysis

When analyzing the reliability of systems, different techniques can be chosen for different purposes. The most common techniques are reliability block diagrams, network diagrams, fault tree analysis, the Monte Carlo simulation and the Markov model. We can analyze each technique and their basics as follows.

2.3.1. Reliability Block Diagram

This technique offers an easy understanding of reliability expression and evaluation. In a reliability block diagram, each system component is shown in blocks and they are connected to each other either in series or in parallel as shown in Figure 2.

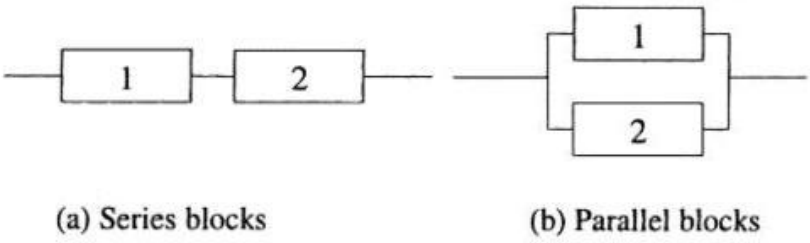

Figure 2. System component relationships [Xie et al., 2004]

If the reliability of a block *i* is already known or estimated then the system's total reliability can be calculated as follows, assuming that the components are independent of each other.

$$R_{system} = R_1 \cdot R_2 \quad \text{for series connection and}$$

$$R_{system} = 1 - \prod_{i=1}^{2} (1 - R_i) \quad \text{for parallel}$$

After calculating the total reliability of simple blocks, these blocks can be merged into one block and by iterating this process the whole system's reliability can be calculated.

2.3.2. Network Diagram

Network diagrams are very similar to block diagrams but they are mainly used to represent networked systems such as a distributed computing system, LAN/WAN or wireless communication channels. Each node on a network diagram represents a distinct network element that has its own resources and elements to execute a networked application. The reliability calculation is similar to reliability block diagrams.

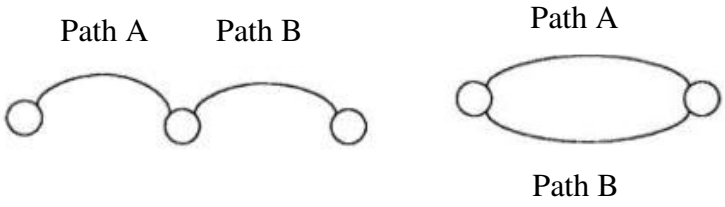

Figure 3. Network links of two connected nodes [Xie et al., 2004]

2.3.3. Fault Tree Analysis

A fault tree of a system shows which combinations of component failures will result in a system failure. This is a very common tool for system safety analysis and can be implemented in many reliability testing applications. In a fault tree diagram all component failure events are represented by nodes and they are connected to each other with simple logical operators such as "AND" and "OR". After each failure events are

connected with these logical operators and a final output event (which mostly shows the system failure) is reached (Figure 4-5).

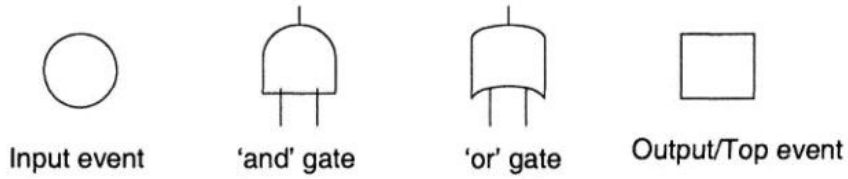

Figure 4. Basic elements of a fault tree diagram [Xie et al., 2004]

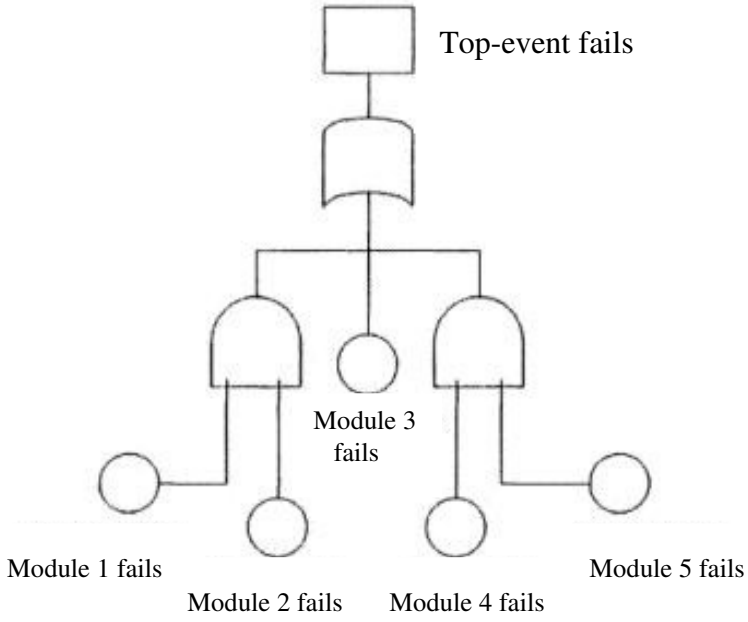

Figure 5. A fault tree diagram for 5 modules [Xie et al., 2004]

2.3.4. Monte Carlo Simulation

Monte Carlo simulation is a technique where a series of events are applied repeatedly which use parameter values. Xie et al. [Xie et al., 2004] summarizes these series of events as follows:

1. Simulate random numbers.
2. Evaluate the desired function.
3. In order to obtain n samples of desired function apply steps 1 and 2 repeatedly. At the end, the system failure times will be T(1), T(2), …., T(n).
4. Calculate the expected value of the system failure time using:

$$E(t) = \frac{1}{n}\sum_{i=1}^{n} T(i)$$

5. Obtain a sample standard deviation of the estimated value.

This method requires a custom and complex program and may also take a considerable amount of time in order to obtain precise results.

These techniques are used not only for software systems but also for every project or product development which requires reliability measurement and analysis. As we can see, they have different approaches for measurement and they offer us many options to choose from.

2.4. Models for Software Reliability

In addition to the reliability measurement techniques, there are also models for reliability which were developed especially for software.

2.4.1. Basic Markov Model

The basic Markov model was developed by Jelinski & Moranda in 1972, and is a modified version of the general Markov model for software reliability. Xie et al. [Xie et al., 2004] explain that the Jelinski-Moranda (JM) model also has some assumptions:

- The initial number of software failure is an unknown but a fixed constant.
- If a new fault is detected, it is removed immediately and no new faults are added.
- The times between failures are random variables which are independent and exponentially distributed.
- All other faults in the software affect the software failure rate in the same amount.

After following these steps and eliminating each fault the ultimate goal is to detect the final fault and reach a faultless state. The model can be observed in Figure 6, where N_0 denotes the initial state of the software, Φ denoting the failure rate contributed by each fault and reaching a final state with 0 faults.

Figure 6. Basic Markov Model, states of the system [Xie et al., 2004]

This model seems fairly simple and easy to apply. However, in order to reach successful results by using this method, there should not be any modification to the software. These modifications may bring more failure, and there is a probability that the system will turn back to a state where already detected and solved failures must be

handled again. Moreover, testing the software must be standardized in order to detect failures one by one and assign them the same value for calculating software failure rate.

In addition to the possibility of making wrong assumptions (such as the initial number), this model also has other weaknesses. For example it cannot be guaranteed that the testing process will find every failure. If the proper test case is not applied, a possible reason for failure can be missed and if that failure is not observed while using the software, it does not mean that the software is reliable completely. Furthermore, detecting and removing a failure may need software modification which is a contradiction for the model's assumptions. Because of these weaknesses several Extended Markov Models have been introduced with slight modifications to the assumptions.

Different from the Markov models, there are also some Non-Homogeneous Poisson Process (NHPP) models. A simple example of a NHPP process is the arrival rate of customers to a shop in a day, where this rate increases in certain hours of the day and decreases in others. In the software models that are based on NHPP, the failure occurrence rate is a function of time and the aim is to have a decreasing rate after removing a fault and keeping it constant until a new fault is found. An example of this model can be the Goel-Okumoto model.

2.4.2. Goel-Okumoto (GO) Model

The GO Model [Goel and Okumoto, 1979] assumes that the failure process is an NHPP. The basic assumptions of the model can be listed as follows:

- The faults detected at a time t follows a Poisson distribution.
- The faults have the same chance to be detected and they are independent of each other.
- Similar to the Markov model, all detected faults are removed immediately and no new faults are added. [Xie et al., 2004]

The two models JM Markov and GO can look very similar but the basic assumption creates a very solid distinction between them. The removal of a fault creates a discrete change in the failure intensity in JM Markov; however in GO the failure intensity is considered as a continuous function over the whole time.

Like the Markov model, there are many NHPP models which are based on similar assumptions but with some extensions and modifications. Examples of these are the Musa-Okumoto model, the Log-power model and the Duane model. [Musa, 1980; Xie et al., 2004]. In addition to the software specific reliability models, there are also commercial models developed particularly for Open Source Software.

2.4.3. Navica's Open Source Maturity Model (OSMM)

The Open Source Maturity Model, a trademark of Navica [Navica, 2007], is an open source product providing a formal methodology to assess the maturity of an open source product based on the needs of the assessing organization. Navica's OSMM was developed following the book of its CEO Bernard Golden, entitled *Succeeding with Open Source* [Wilson, 2006]. When OSMM is used to determine the maturity of an OSS, a maturity score is created at the end for a comparison with similar products. However, this model and the maturity score were not intended for the comparison of OSS with closed-source proprietary counterparts. This model works on three different principles [Wilson, 2006]:

- *Maturity tests:* The model proposes questions for the tested open source software in order to evaluate its maturity. The questions are generated as both quantitative and evaluative and separated into different categories such as Software, Support, Documentation, Training, Integration and Professional Services. For each question, a score is generated depending on how well the software performs.
- *Requirements weightings:* The maturity tests are given appropriate weights according to the relevance and importance to the intended use and users of the software.
- *Final scores awarded:* A final score is awarded in order to see if the software is ready for use and to indicate its evaluation compared to other similar software.

After the maturity tests are performed, the score for each category is multiplied by its weightings and a final score between zero and one hundred is produced. This final score then can be used to identify the maturity of the OSS and different OSS products can be compared with each other by using the final score or the sub score of the test categories mentioned above.

Similar to the Navica's OSMM, there are also other OSS assessment methodologies. Besides sharing most of the functionalities and basic properties, there are also minor differences which are summarized in Table 1.

Criteria	**OSMM Capgemini**	**OSMM Navica**	**QSOS**	**OpenBRR**	**QMM**
Seniority	2003	2004	2004	2005	2008
Authors/ Sponsors	Capgemini	Navica	Atos Origin	Carnegie Mellon Silicon Valley, SpikeSource, O'Reilly, Intel	QualiPSo project, EU commission
Assessment Model	Practical	Practical	Practical	Scientific	Scientific
Detail levels	2 axes on 2 levels	3 levels	3 levels or more (functional grids)	2 levels	3 levels
Predefined criteria	Yes	Yes	Yes	Yes	Yes
Technical /Functional criteria	No	No	Yes	Yes	Yes
Scoring Model	Flexible	Flexible	Flexible	Strict	Flexible
Scoring scale by criterion	1 to 5	1 to 10	0 to 2	1 to 5	1 to 4
Iterative process	No	No	Yes	Yes	Yes
Criteria weighting	Yes	Yes	Yes	Yes	Yes
Comparison	Yes	No	Yes	No	No

Table 1. OSMM Methodologies [Wikipedia – Methodologies, 2010]

3. Analysis and Classification of Different OSS Paradigms

The quality of open source software can be measured by software reliability models and different reliability models have been used in cases of software released for public use. The common models can model the whole lifecycle of an OSS project or can try to evaluate the quality by analyzing the defect arrival patterns after release such as the Rayleigh model and reliability growth models. Both of these example models require data for metrics and inputs for their predefined criteria. After gathering the required data and inputs, one can then implement a simulation model for the development process of OSS projects [Antoniades *et al.*, 2002-2007]. A sample simulation model may consist of the following structure.

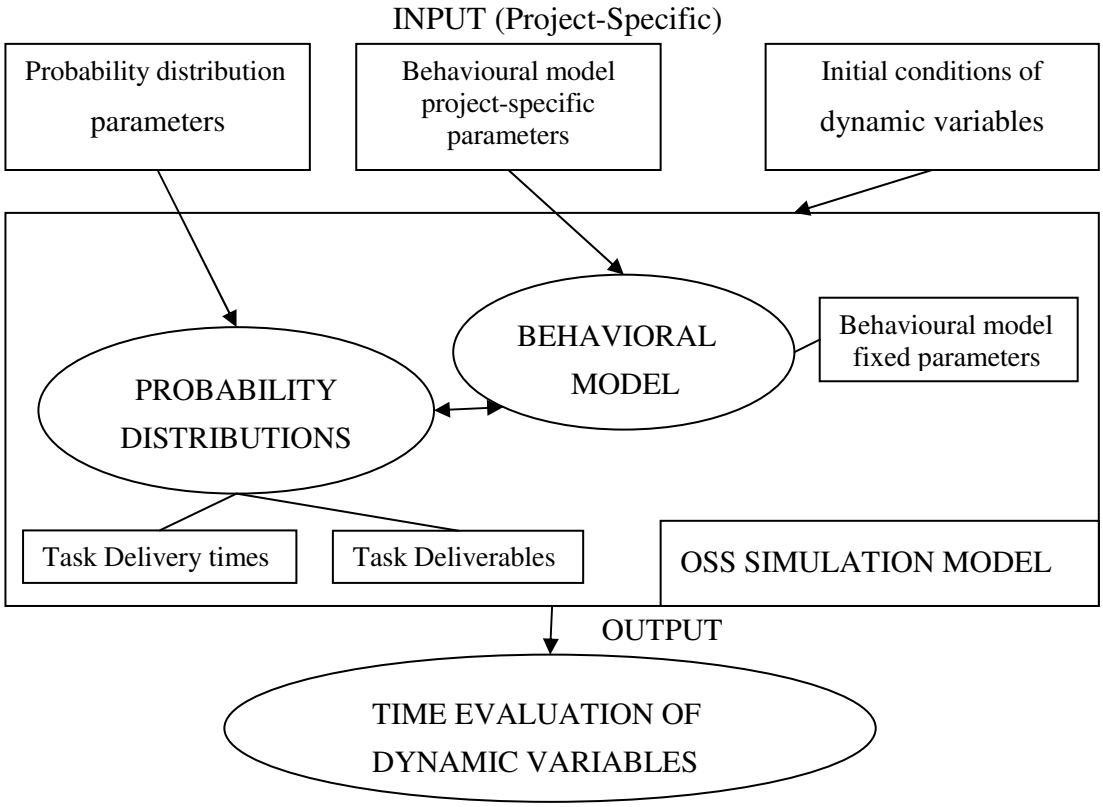

Figure 7. General Structure of an OSS dynamic simulation model [Antoniades *et al.*, 2002]

There are some certain differences between OSS development and proprietary and commercial software development. Hence, a software reliability model for open source software would have its distinct properties, required inputs, development characteristics and metrics compared to Closed Source Software (CSS).

Before analyzing different OSS paradigms, the main characteristics of OSS development are explained next.

3.1. Open Source Software Development

The OSS development mainly relies on the practice of welcoming every enthusiastic individual who would like to contribute in the project. On top of this, the freedom of using, modifying and distributing OSS leads to more robust software and more diverse business models [Wu and Lin, 2001]. This freedom of participation constructs the OSS development cycle for both individuals and groups and a typical OSS project follows the cycle summarized as a flow in Figure 8. This typical OSS project development is originated by a personal need which results in a project initiation if any similar one doesn't exist or contribution to the existing one. Following the open source principals of using mailing lists, version control systems, writing documentation and manuals, deciding on a license model, the projects matures into its final stage of releasing.

This development cycle brings many reliability concerns such as controlling the several participants spread across different regions, verifying the contributions' validity, agreeing on a release version or concurrent/separate development dilemma. Background and development styles of the developers and the personal needs about why a certain OSS should be developed can be the examples of people related problems threatening the overall success of an OSS. There are several paradigms similar to the ones above and while trying to create a reliability model for OSS their effects on a possible model need to be analyzed and taken into account.

In order to serve the intention of this thesis study which is to establish a software reliability model for open source software, different OSS paradigms were studied and proposed solutions for this purpose were analyzed. The paradigms were categorized into three different aspects as the most significant variables for achieving such a model. These three aspects of reliability of OSS are the importance and characteristics of the target product, implementation characteristics of the OSS project in connection with its source code and finally the open source community.

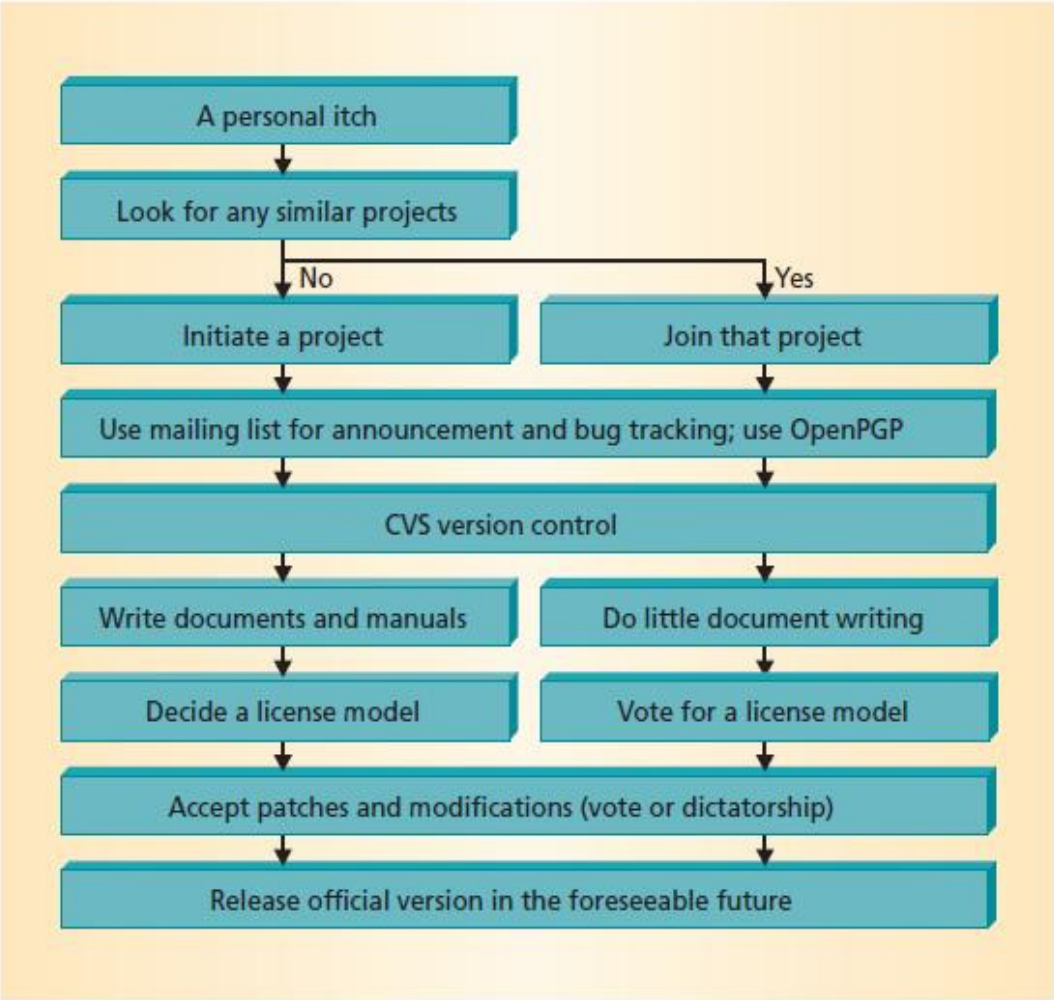

Figure 8. Open Source System Development Cycle [Wu and Lin, 2001]

Importance and Characteristics of the Target Product

Most OSS projects are initiated by personal needs. These personal needs usually grab the attention of different software developers and motivate them to contribute to the project [Crowston and Scozzi, 2002]. In connection with this contribution, the developers will finally be the real users of the software which grants them the ability to comprehend the requirements of the project clearly. Crowston and Scozzi consider this as an evolution of OSS engineering into developing software that meets developers' needs. They also claim that the identification of user needs and demands for improvements in traditional software development process is eliminated since the developers in the OSS project know their own needs. By looking at the recent OSS products, we can clearly see that there are very successful OSS tools for software development and end-user tools to satisfy the personal needs of the developers such as email, word processing and automated tools but in contrast we see very few OSS products built for the issues that the developers rarely come across. A significant example area where we can see the connection between the needs and successful

product development is the game development following open practices [Isitan *et al.*, 2011]. Openness is, naturally, a significant factor for games evolution, overall acceptance and success. What openness can provide to the product development of personal needs is the integration of different knowledge, skills and expertise can provide a more formalized, flexible, development approach that is time efficient and increases developers' productivity. These needs also derive the responsibilities of the contributors such as configuration management, release scheduling, code acceptance process and other management activities [Samoladas *et al.,* 2004]. Coordination of the development process, definition of these responsibilities and the rules to be followed are crucial for the success of the project.

There have been studies to find a relation between the modularity of an application and its quality by looking at the size of application and user satisfaction as the two. The assumption that a program is modular if its components are made of small size is reasonable in terms of open source applications, because the development methods of Open Source software allow the developers keep the application as compact as possible (in other words modular) with several revisions [Stamelos *et al.,* 2001].

In their study, Stamelos *et al.,* use the number of executable statements and program length as the indicators of the program size and try to reach an index for every program in the case study. However the user satisfaction levels are only divided into four categories and the number of test users seems to be surprisingly low (4 in this case). Also the chosen applications are totally random and we're not provided any information about the background/familiarity of the test users with those applications.

The case study results show that the users found the smaller sized programs (more modular) as more satisfying and easier to use. An assumption here is that the reverse can be reached easily by choosing users experienced with high-level and complex programs but still which are really easy for them to use. In fact the satisfactions of end users are valid if the user is going to use that program for a long time or for a purpose, not just for once for a case study. A similar study can be replicated by using this matching: complex applications-experienced users, to reach other result sets.

Crowston and Scozzi [2002] further stress the importance of the requirement analysis in reliable OSS development. The main focus of their research is how the requirements are gathered and analyzed for OSS projects with regard to the non-developer topics, where the developers do not have a deep knowledge of the field. It would seem that the requirement elicitation and analysis will always be the basis of the OSS development and we can say that the more the developers are connected with the requirements and closely related to the project needs, the more reliable the end product will.

We can see that the characteristics of the target product influence the overall reliability of it. While the developer community is selected and requirements are set, proper documentation and technical support should also be taken into consideration. Samoladas et al. [2004] denotes *a ready interpretation of the OSS development process is that of a perpetual maintenance task*. This brings up the necessity of target product definitions since there will be series of maintenance tasks and debugging of current functionalities in addition to the new feature implementations.

A serious limitation of the open source software development is the patents and trade-secrets used by companies. In most cases if the target product is open source software, it cannot contain a number of important algorithms which are patented and trade-secret by the implementing company/community. When the developer community demands the use of a patented algorithm either they are refused its use, which might then need to be open source, or they have to wait for the algorithm to enter the public domain as in the case of the widely-used RSA algorithm [Yeates 2005]. For this reason, we can say that a software reliability model should consider the need to use a patented algorithm in the development of the open source software and the available solutions to overcome the problem.

Implementation Characteristics of the OSS Project

Previously we mentioned the importance of developers of a specific OSS project as its end users upon completion. In the product development lifecycle of the OSS, the inspection and contribution to the source code of an OSS project always take place in terms of bug fixes and addition of features. Since the users are the developers themselves, they usually fix the bugs themselves rather than waiting for another developer to do so. As a result, the bug fixes and implementation of additional features take place more quickly and successfully [Crowston and Scozzi, 2002]. Crowston and Scozzi also suggest that OSS engineering process should define the bug fixing process and handling of projects symptom reports beforehand. In my approach to develop a software reliability model for OSS products, time and performance are important variables affecting the general reliability of a product, so I support the idea of the contribution of end users as the real developers to achieve a more reliable OSS product.

The implementations of most OSS projects evolve from their closed source counterparts. Even though there are clearly distinct and diverse properties of OSS software development process, one may consider trying the find the similarities between two and constructing an OSS reliability model similar to CSS counterparts when possible. Zhou and Davis [2005] suggested that it might be possible to build a general reliability model for open source software projects to highlight key features and critical time points which can be used to assist in making adoption decisions. The research focused on eight example open source projects which were developed under

the SourceForge.net community and labeled as most active in August 2004. The proposed distribution model was the Weibull distribution family, a continuous probability distribution and especially Rayleigh distribution which is a special form of Weibull where the magnitude of the distribution reaches a peak over time then starts to decay gradually. Figure 9 shows several Weibull probability density curves with different shape parameters and the Rayleigh distribution is represented when the parameter is equal to 2.

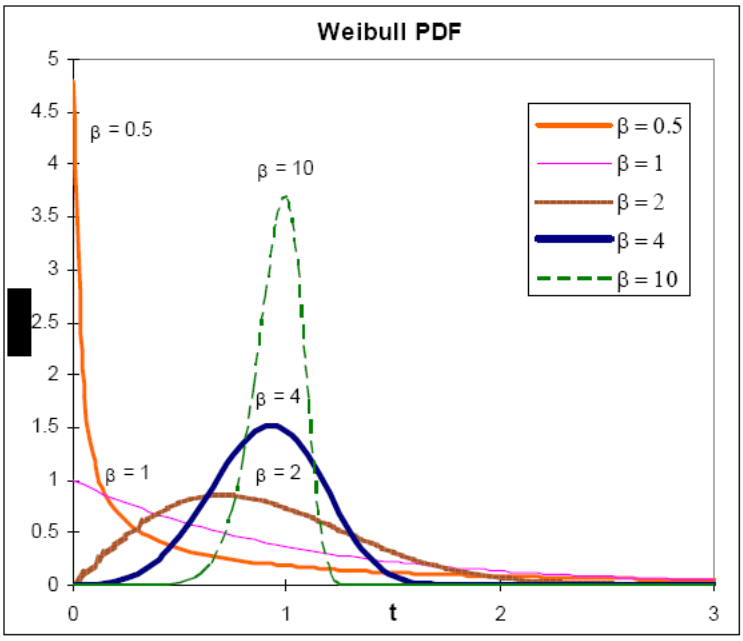

Figure 9. Weibull and Rayleigh (β=2) distributions [Zhou and Davis 2005]

The model was created by looking at the bug arrival rate through the lifetime of the project. As we can expect, after the release of the software many bug reports take place and while the project stabilizes with fixing the bugs less bug reports occur. Moreover, many open source software projects are considered to get on and off during its development period. Six of the eight examples showed the expected Weibull distribution and the other two didn't because they were not at their stabilized stages. In relation to the developer contribution suggestions of Crowston and Scozzi, one of the projects had only one developer and predictably the frequency of bug reports and activity indexes had large variations. The explanation given to these large variations was given as sometimes OSS project coordinators might make rapid changes between releases which might affect the code structure and bug arrival rate [Zhou and Davis, 2005]. When we analyze their findings in terms of constructing an OSS reliability model we can see that most of the OSS projects show similar reliability growth patterns. In contrast to the closed source projects, it is really hard to follow a special pattern which will fit all of the OSS projects. This implies the necessity of modeling different open source software projects individually. Moreover, using time series

analysis and metrics will fit more for OSS projects because of the variable bug rates and activities and also the get on vs. get off periods.

A similar example of bug report rate and the effect of subsequent changes were analyzed by Isitan et al. [2011]. In the analysis we see that the FOSS community accepts that the rate of bug reports and change requests can indicate the success and reliability of a project. Change requests and related releases show that the project under development becomes more decent and compatible with the original product. Thus, the availability to review the source code of different releases helps the people understand what to report and what not. The rate of bug reports and change requests is on peak level after the initiation of the project or major releases and declines throughout the lifetime of project.

One interesting criteria that Zhou and Davis took into account while trying to build an OSS reliability model was the relationship between page view, including project or software's home or info pages, or number of downloads with software reliability. Their conclusion was these popular measures are not highly related with reliability metrics and are not suitable for measuring project quality. However, if further variables such as developer and downloader profiles, documentation, interoperability and coverage were taken into account I think that some correlation between these popular measures and OSS reliability could be reached. In fact the open source motto "with enough eye balls, all bugs are shallow" implies that the page view and download numbers would for sure support the bug fixes, additional features and hence general reliability.

Maintainability is an important concern in software development and with many contributors and rapid changes in the source code of open source projects it's even more significant. Related to this characteristic, Samoladas et al. analyzed the results of some case studies concerning the maintainability by looking at the source code of the projects listed in Table 2 and creating a maintainability index (MI).

Project Mnemonic Code	Application Type	Total Code Size (KLOCs)	No. of releases measured	Project Evolution Path
OSSPrA	Operating system application	343	13	OSS project that gave birth to a CSS project while still evolving as OSS
CSSPrA	Operating system application	994	13	CSS project initiated from an OSS project and evolved as a commercial counterpart of OSSPrA
PrB	Operating system application	860	10	CSS project that opened its code and was transformed to an OSS project
PrC	Programming language	1050	11	Pure OSS project
PrD	Database management system	1411	8	Academia project that gave birth to an OSS project
PrE	Internet application	1198	14	Pure OSS project

Table 2. The characteristics of sample open source projects [Samoladas *et al.,* 2004]

Their analysis is based on the MI which includes size of code, code complexity, lines of comments, number of operations in code and conditional statements and finally gives an MI value after proper calculations. Five different projects were compared with each other by looking at their MI index values through their successive version releases. One interesting outcome of this study was the OSS code quality seems to be equal or better than the CSS codes in most cases [Samoladas *et al.,* 2004]. Furthermore the MI values for all the projects appear to be decreasing for the latest releases of them. Analyzing the MI indexes and constructing a framework based on these MIs looks like a good approach to be used in an open source reliability model. However, one thing that the researchers didn't take into consideration was the adequate information gathering about the characteristics of the chosen projects. This was claimed to be studied in the extension of their work with further cases which should be done in order to reach more concise and accurate results. In my opinion, the study can be extended by looking at another index rather than the MI and compare the results of two different index calculations. In that way the comparison between OSS and CSS would be clearer. Even within the OSS projects some conclusions can be reached such as the relationships between number of lines of code, number of developers, size of a new/changed feature and reliability/maintainability. If some relations between reliability/maintainability with several variables are found, the characteristics of a reliable OSS model can be drawn.

Open Source Community

As mentioned before, it is very hard to argue that the success of open source software is dependent on the profit or loss of the developing community. In contrast, the commonly accepted criterion of project success is whether or not the developed

software preserves the trust and goodwill of the open source community which then results in the lead on the development path of the software [Metcalfe, 2006]. In order to preserve the trust of the community, safety measures for software reliability are always taken into consideration.

One of the biggest properties of open source development is the unlimited and unrestricted contributions of the developers to the project. In several cases, the developers who find the project interesting or appealing for their own use can join the development team, with some exceptions in open source software projects with restricted developer team for a specific period of time or release version. The ability to expand the developer community to anyone has indeed advantages and disadvantages. The biggest problem is the security concerns of the developed software. Any developer might be a possible attacker or exploiter who might add vulnerabilities to the source code. To overcome this possibility, most OSS projects implements the trusted contributor concept. A trusted contributor can commit his changes directly to the OSS project and all other contributors must have the approval of the trusted contributors. Furthermore, normal contributors can become trusted contributors in the lifecycle of the project and if they are caught to be adding vulnerabilities to the source code they might be expelled from the developer community and their contributions might be purged [Yeates 2005].

Here we can see the importance of the motivation of contributors within the open source community. In addition, the reputations of the most contributors increase. Gaining the trust of other contributors and achieving a higher status in the development team will elicit higher quality work and hence reliable software. At first this contribution subject might be seen as a security flaw, however further contributors to a project can find the already available vulnerabilities and they can fix them. This decreases the burden of finding and fixing the possible vulnerabilities from the actual developers which might take significant time and resources in the lifecycle of reliable software development. In fact it has been discovered that applying the fixes tends to be issued considerably faster for open source software than for closed source software [Yeates 2005].

The ease of contribution to an open source software project brings in further benefits. One of such benefits is the simplicity of the communication within the community. As long as the developers have more contributions as described before, the community is always exchanging information within themselves. This information exchange is especially crucial for the requirements elicitation phase, where the basic requirements for the target software is set. These requirements must be defined as clearly as possible in order to increase reliability and security of the software. Crowston and Scozzi emphasize the importance of this communication exchange and setting up the requirements accordingly. They suggest that, in order to achieve high quality open

source software and increased reliability the frequency of the feature requests submitted should be analyzed. Furthermore, how these feature requests are handled by the OSS developer community must be analyzed and different roles of developers in this handling process must be identified [Crowston and Scozzi, 2002]. In my opinion defining the roles of developers in requirement elicitation and construction phases and for feature request handling should be parts of a unified OSS reliability model too. The characteristics of a model should include clearly defined roles, process rules and routes similar to the roles and the way of constructing requirements as described above.

The study of Samoladas *et al.* provided us an understanding of the importance of source code measurement for assessing software reliability. This assessment can only be done by the existence of proper software testers, debugging utilities and availability of peer reviews of the source code. When these conditions are satisfied, monitoring and improvement of the source code can be performed increasing the reliability. The study also came up with the following conclusions in terms of the developer community [Samoladas *et al.,* 2003- 2004]:

- After measuring the code quality of similar CSS and OSS products, one can say that OSS code quality is at least equal or sometimes better than the quality of the CSS counterpart. This might be due the high motivation and skill of the OSS programmers, creating an important advantage against CSS programmers.
- Careful individual analysis is needed in OSS code assessment because of the rapid changes between releases. The OSS project coordinators are the mainly responsible for assuring code quality hence reliability.
- Maintainability of the OSS product deteriorates over time similar to the CSS counterparts. To prevent this, appropriate reengineering actions are necessary and preventive maintenance must be taken into account.

Finally the developers of an ongoing OSS project should think about the further contributors and non-developer people who are going to use the software. The information transfer to further contributors and non-developers is handled by writing support documentation. It is usually considered that the role of people involved in writing documentation not relevant in OSS projects, however it can reveal fundamentals in software development targeting non-developer users [Crowston and Scozzi, 2002] and data for reliability metrics to be used and construct a framework for a reliability model.

4. Case Studies on Open Source Software Reliability Evaluation

According to the research by Zhou and Davis [2005], the IT community is getting more used to applying open source solutions and in recent surveys majority of companies are found to be using open source software commonly as server operating system, web server and for web development. The two famous open source products Apache web server software and Linux operating system are the most common ones among these solutions and have proven their quality and reliability. Although these two famous solutions reflect the great success of open source development, people are still doubtful about deciding OSS solutions over their CSS counterparts. This hesitation is present in both non-commercial organizations such as government units and business people. Ray Lane, former Oracle executive, signified the two common apprehensions as the lack of formal support and velocity of change during keynote at the Open Source Business Conference 2004 [Farber, 2004]. We can certainly say that these hesitations are all originated from the quality, reliability and security levels of open source software and their evaluation.

As we have seen in the discussion section about different OSS paradigms, it is very hard to use a unified model for every distinct OSS project or component in order to evaluate their reliability. In this chapter I will try to signify the important characteristics of a target OSS reliability model, enclosing aspects of several evaluation approaches. In order to do this, different case studies and possible research agendas or suggestions will be compared and contrasted with the aim of reaching a final model.

4.1. Revisiting Software Reliability Fundamentals

The first approach about reliability evaluation of OSS is a research agenda by James A. Whittaker and Jeffrey Voas which supports that software reliability should be redefined as a function of application complexity, test effectiveness and operating environment [Whittaker and Voas, 2000]. In order to understand the outcomes and ideas for future work of case studies, one should consider the current definition and state of OSS reliability evaluation criteria and techniques. In addition, the limitations of the current state and possible solutions or suggestions to overcome them always have value in the path of accurate evaluation. The study gives a thorough overview of the current state of reliability evaluation and provides several bright ideas about how to improve it. Instead of testing a model or evaluating reliability of a software system, their study provides new ideas for further models and evaluation approaches. Given the definition of reliability as "*the ability of a system or component to perform its required functions under stated conditions for a specified period of time"* [IEEE 610.12-1990],

their study is questioning how successful the time and operating conditions variables are in reality for software reliability. The claim in the study is that the definition of software reliability can be inaccurate while being constructed based on hardware reliability and hence the time and operating conditions would differ between hardware and software. Software does not wear over time and it can be developed perfect and remain perfect. It can fail due to the undetected faults in the development and testing stages. It is true that time is an important variable in these stages as well such as the amount of time spent on development or on several different testing phases and conditions. What this study criticizes is that the reliability and mean time to failure formulas in software reliability are lacking the application complexity and thoroughness of testing in relation to the time variable. In fact those formulas treat every software and test case uniformly. In addition, the evaluated reliability is valid only for the testing profile used, assuming that the users will always follow the paths tested by the reliability evaluation on a specific system configuration. Whittaker and Voas describe this problem as "*Using software is like walking in a jungle. As long as you stay on the well-beaten path, you shouldn't get eaten by a lion, but if you stray from the path, you are asking for trouble"*. If the users choose to use the software in an untested way outside the profile or if the software is run on another system with different operating system, memory or processor, the calculated reliability of the software will become invalid. Furthermore, software repairs and modifications are also not addressed in the current definition of software reliability. These modifications can make the reliability and quality of the software better, worse or keep it unchanged. What Whittaker and Voas criticize is that reliability is more complex than formulating time and operational profile representation into something that can be evaluated mathematically.

The following case studies in this chapter will focus on issues such as calculating the reliability of modifications and component adaptations and also systems as a whole. They are chosen as an example for why the current definition of reliability is lacking and should be including more criteria especially in the Open Source case. First I will explain Whittaker and Voas' proposals for revisiting the software reliability fundamentals, along with their suggestions for improvements.

Application Complexity

The first claim is that looking only at test data and defining random variables for describing application reliability is not adequate. In order to get an accurate measure of an application's reliability, the application's size and complexity must be taken into consideration. Moreover, reliability models must account for an application's inherent complexity and consider the amount of testing compared to the size of the testing effort. The current measures such as Halstead's length measure and McCabe's cyclomatic complexity [McCabe, 1976] are not sufficient for measuring application complexity and

new methods are needed for complexity measurement and reliability evaluation. The study [Samoladas *et al.,* 2004] discussed before have taken a good approach in order to justify the importance of application complexity in OSS reliability. More similar studies are needed in order to accommodate code complexity metrics into open source software reliability evaluation and modeling. The application complexity plays a significant role in reliability evaluation which will be discussed in the Early Reliability Prediction study later in this chapter.

Test Effectiveness

The current reliability metrics concern about the time spent in testing instead of what exactly the testers performed during that time. The real test evaluation should focus on the variety of test data gathered from the user inputs and possible usage scenarios. By knowing that the test data is complete enough to cover all the scenarios, the reliability measurement of the software will be more accurate regardless of the time spent on testing and finding a possible bug. In theory a test suite should continue until all the faults in the software are found and fixed, however this is inefficient and costly. Instead of this approach, the study suggests to create a testing stoppage criteria based on test effectiveness. These criteria are created based on a set of guidelines for generating a good test suite. There is a possible weakness in this approach such that it is unknown for the testers where the real faults are. Whittaker and Voas' [2006] suggestion is to use software fault injection method in three steps.

- Insert simulated faults into the source code. An example simulated fault can be changing x = x + 1 to x = x - 1.
- Develop a testing suite which can detect all the simulated faults and distinguish the mutated code from the original source code.
- Use the test cases to test for real faults.

In practice a test suite detecting smaller faults are also successful in detecting bigger ones, but the reverse is not necessarily correct. There can be different test suites or reliability evaluation methods using the above steps and whichever is more effective catching the smaller faults is considered to reveal more about the code's reliability. Therefore instead of considering only the size of the test suite, the new definition of reliability theory must take the test effectiveness into account. The case-study [Samoladas *et al.,* 2003] gives a nice example of how a well-organized test suite can provide accurate reliability analysis of an open source software case which will be described later in this chapter.

Complete Operating Environment

The final proposal by this study emphasizes the importance of the testing and operating environment of the software being evaluated. Assigning a reliability value to software regardless of the environment it is run on is not sufficient and the value of operating environment in reliability assessment should be increased. This new proposed profile by the study should include the operating system, third-party application programming interfaces, language-specific runtime libraries, and external data files that the tested software accesses. This does not mean that each and every environment element must be included because overstressing them would result a reliability evaluation specific to that system with the elements. The suggestion here about how to support the new proposal is to create evaluation tools that contain the entire domain of inputs and test cases based on the environmental factors. This idea will be supported by a case-study of reliability assessment tool later in this chapter.

4.2. Early Reliability Prediction

In Whittaker and Voas' study [2006] we have seen how significant the application complexity is for software reliability assessment. On top of their suggestions, this next study by Lee et al. extends the idea of exploring code modularity and code maintainability as application complexity related factors that affect software reliability [Lee *et al.*, 2008]. Instead of exploring the idea on a complete OSS system, the authors decide to study the concept in OSS adopted software system in the early stage. Since open source software development is composed of several modifications and adaptations through the product lifecycle, this is a good example of understanding the reliability analysis while moving to a complete OSS system.

The main hypothesis of this study is based on the idea that the requirements, the modularity and the maintainability of OSS will affect the reliability of OSS adopted software systems. In order to prove this hypothesis three steps need to be followed:

- The requirements specification of the system is analyzed and weight indexes are found by mapping requirements and software modules which are linked with requirements. In most OSS cases, the requirements specification is the main building block for modifications. So by analyzing these specifications the weight indexes are calculated in order to find the initial fault rate.

- The product metrics of the software modules are measured. These metrics indicate values such as number of lines of code, code maintainability and code modularity. For code modularity and code maintainability representation Module Interaction Index (MII) and Maintainability Index (MI) are chosen respectively.

- Finally the initial fault rate is calculated from the collected metric values. This initial fault rate is then applied to an appropriate model and the software reliability is assessed. This calculation is based on the following formulas:

$$\delta = A*D*(SS*SM*SMI)$$

where δ denotes the initial fault density prediction, A as the Application Type, D as the Development environment, SS as the Software Size, SM as the Software Modularity and SMI as the Software Maintainability.

Then the initial fault rate (λ) as:

$$\lambda = \omega*F*K*(\delta \text{ *number of lines sources code})$$

where ω is the weight index, F is the linear execution frequency of the program and K is the fault exposure ratio.

Related to the Whittaker and Voas' study here we can see the application of embedding application complexity into the assessment of OSS reliability. Furthermore variables such as application type and development environment also fulfill the idea of constructing the complete operating environment for reliability assessment. Since this study is an ongoing effort, the results of the assessment cannot be analyzed right now; however the contribution of the idea "early reliability prediction" has a value in the nature of OSS development consisting of rapid changes and adoptions. As we have seen in different OSS paradigms, several parties and development stages will benefit from these early predictions. In the next case study we can see another method for evaluating open source component adoptions, but this time with a tool rather than the mathematical approach.

4.3. Trustworthiness Evaluation and Testing of Open Source Components

The three important proposals mentioned before for accurate reliability evaluation are taken into account in the case study of Anne Immonen and Marko Palvainen. Compared to the study by Lee *et al.*, their study [Immonen and Palvainen, 2007] does not consider the source code as the reliability evaluation criterion but instead it uses the reliability and availability goals of the Open Source Components (OSC) and transforms them into architectural elements and represents them in architectural models. These representations then are evaluated by a testing tool giving the reliability measurements

of the open source components and the whole system correspondingly. Moreover, this study also provides an insight understanding of the system being built before the component integrations. So it can be given as a practical example for early reliability prediction.

The evaluation in the study has technical and non-technical parts (Table 3).

Levels	Technical evaluation	Non-technical evaluation
Component	• Predicted reliability of new components • Reliability testing of OSCs • Analyzed reliability of OSC with the help of testing	• OS community and its reputation in a domain • Component development and certification processes • Component reputation and user communities • Understandability of a component • Component history, evolution and license
Architecture	• Predicted reliability of the system with OSCs	• Dependencies, constraints • Compliance with standards
System	• Reliability testing of the system with OSCs	• Component installation • Delivery

Table 3. The levels of the trustworthiness evaluation method [Immonen and Palvainen, 2007]

While both evaluation aspects have the same levels of interest as component, architecture and system levels, the technical evaluation contains quantitative reliability analysis while the non-technical part contains qualitative reliability analysis. Unfortunately the case-study focuses mainly on the technical evaluation even though the non-technical elements prove an example of how important the external factors are for reliability evaluation. The case-study and tool development is ongoing and the target is to integrate every element into it. This way the paradigms discussed in Chapter 3 can be justified while integrating their elements into a revised reliability model for FOSS.

The reliability analysis tool in the study enables a quick and repeatable reliability analysis at the component and architecture levels by calculating Probability of Failure (PoF) values for single components from the Markov chain model. The tool reads the sequence diagram for the whole system under evaluation and simulation model as an input, simulates the system and calculates the PoF for every component in the target system and for the whole system based on the reliabilities of the system execution paths. The analysis tool is developed as an Eclipse plug-in that performs the evaluation in unit and integration tests. The working principle of the tool is as follows:

- Code fragments of the system under evaluation are inserted and the dynamic behavior of the components are observed.
- Input attributes and return values of methods and states of the components are listed as logs.
- A sequence of named messages that are delivered between different components during the test are created.

- Based on the message sequences a BNF grammar and a symbol tree are created, where the nodes are associated to the extracted behaviors and messages.
- Finally the symbol tree is used to calculate the reliability values for the target components.

An example reliability evaluation under the tool can be observed in Figure 10. The upper part of the window represents the component diagram of the system to be analyzed in UML format. Based on the components architecture the BNF Grammar for the system (Figure 11) is generated. Next the message log (Figure 12) and parsed behavior tree for the message log (Figure 13) are generated. The lower left view in Figure 10 *Message Contents* includes all the input messages of the simulation and the middle view *Components* contains the components involved in an execution path of the system. Finally the *Results* view shows the analyzed components and calculated probability of fault values.

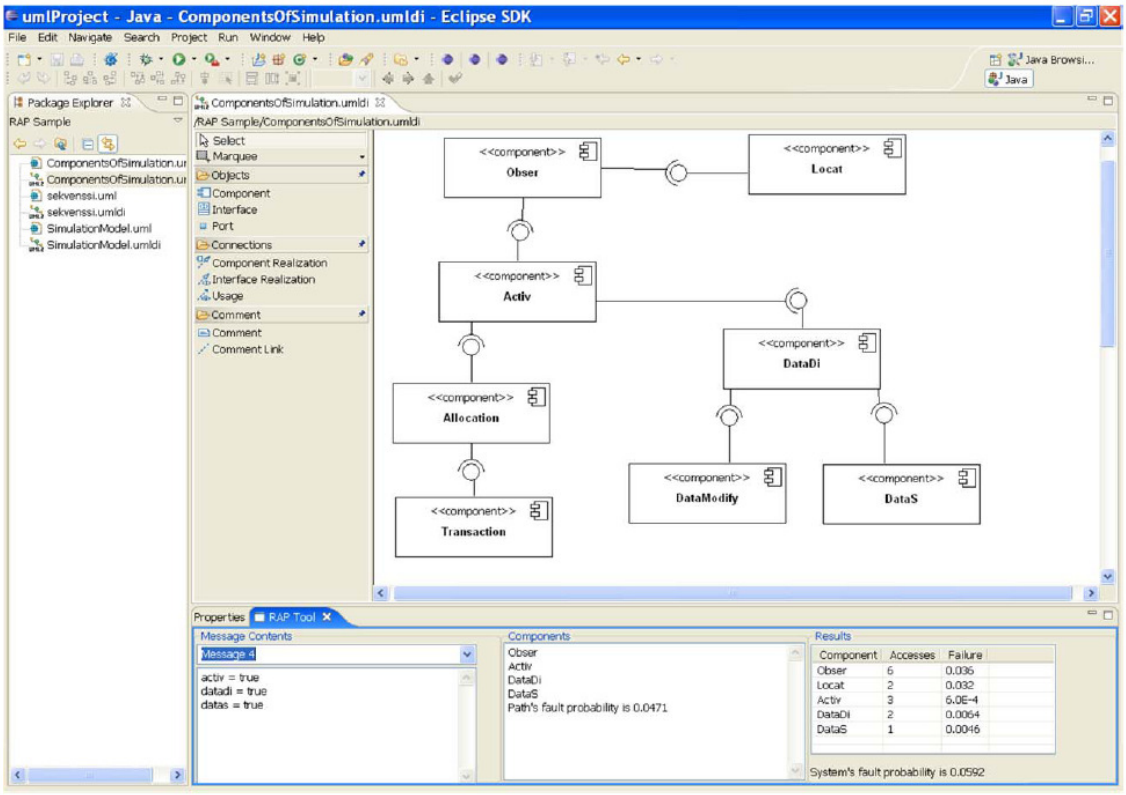

Figure 10. Reliability analysis tool in Eclipse [Immonen and Palvainen, 2007]

```
ReliabilityModel ::= ( ( CorrectBehaviour| FailureBehaviour)* ) <EOF>
CorrectBehaviour ::= ( ( <COMMA> )?REQUEST <COMMA> RESPONSE)
FailureBehaviour ::= ( ( <COMMA> )?REQUEST <COMMA> RESPONSE_FAILURE)
RESPONSE ::= <RESPONSE> Index
REQUEST ::= <REQUEST> Index
RESPONSE_FAILURE ::= <RESPONSE_FAILURE> Index
Index ::= <LEFTBRACKET> <INTEGER_LITERAL> <RIGHTBRACKET>
```

Figure 11. BNF Grammar for the system to be analyzed [Immonen and Palvainen, 2007]

Request [1], Response [2], Request [3], Response [4], Request [5], Failure_response [6], Request [7], Response [8], …

Figure 12. Message Log [Immonen and Palvainen, 2007]

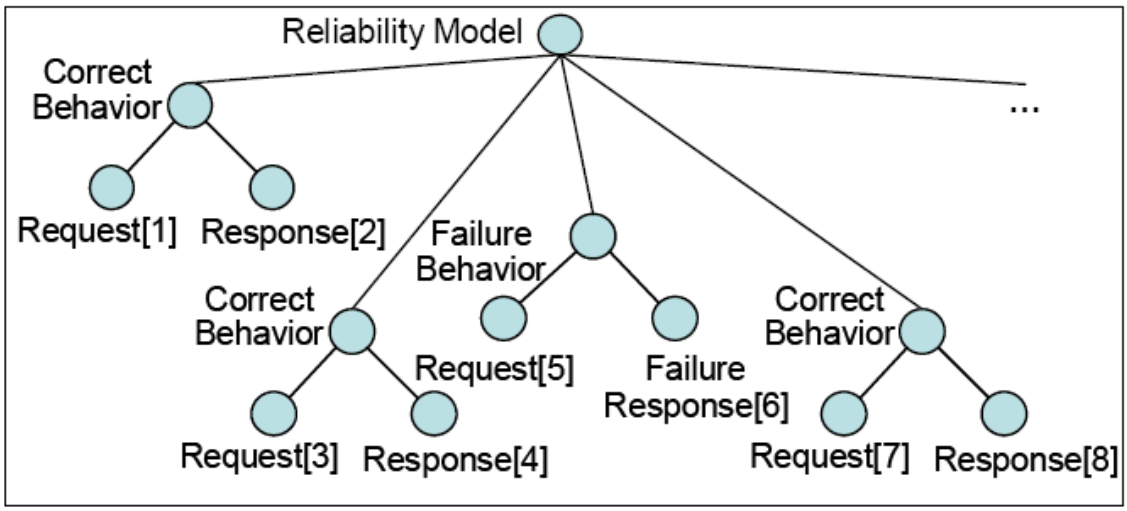

Figure 13. Parsed Behavior Tree for the Message Log [Immonen and Palvainen, 2007]

The reliability analysis tool looks very promising in terms of evaluating open source components and systems as a whole. However it is still lacking the representation of the criteria under the non-technical evaluation ideas. For some cases the non-technical elements can have more significant value over the technical elements so a balance between them must be created with a possible user control over them such as assigning weightings.

4.4. Exploring the Quality of FOSS: A Case Study of an ERP/CRM System

When talking about the complete operating environment I have mentioned the significance of creating evaluation tools that contains the entire domain of inputs and test cases based on the environmental factors. The next case study [Samoladas *et al.,* 2003] is a good example of how the operating environment and domain of the OSS being developed can have influence over its reliability. According to Samoladas *et al.* there is not much FOSS projects for systems that require a more detailed requirements document such as Enterprise Resource Planning (ERP) or Customer Relationship Management (CRM) systems. Since they are not suitable to be developed under FOSS methods, their reliability analysis would provide interesting results. The FOSS ERP/CRM system evaluated in this case study is Compiere [Compiere 2011] consisting of 594 Java files and 101,490 lines of code when the case-study was performed.

In order to evaluate the reliability of this FOSS product with an unusual genre/domain the researchers chose to use an external tool Logiscope® by Telelogic Tau™, distributed as Rational Logiscope by IBM as of 2011 [Logiscope 2011]. This tool provided code coverage analysis, code measurement, code checking against predefined programming rules and quality assessment for developers for identifying error prune modules. Similar to the reliability analysis tool [Immonen and Palvainen, 2007] evaluation this tool also provides an overview of component reliabilities, however the evaluation is done automatically while the testers need to create and simulate the component architecture themselves in reliability analysis tool.

The source code of the Compiere program was parsed into the Logiscope evaluation tool and the outputs are provided (Tables 4, 5, 6, 7). According to the evaluation outputs the Maintainability and Testability of Compiere were calculated as Good. In addition, Analyzability, Changeability and Stability were calculated as Excellent. Furthermore the reliability analysis of the classes and functions inside the whole code are also presented.

	Excellent	**Good**	**Fair**	**Poor**
Maintainability	13% (15%)	51% (60%)	35% (25%)	1% (0%)
Analyzability	75% (89%)	25% (11%)	0% (0%)	0% (0%)
Changeability	47% (56%)	21% (23%)	0% (0%)	32% (22%)
Stability	80% (80%)	17% (18%)	0% (0%)	4% (3%)
Testability	17% (19%)	47% (54%)	32% (23%)	3% (4%)

Table 4. Application Classes Results of the whole code [Samoladas *et al.,* 2003]

	Excellent	Good	Fair	Poor
Maintainability	73% (63%)	23% (31%)	3% (4%)	1% (2%)
Analyzability	81% (75%)	13% (16%)	4% (6%)	2% (3%)
Testability	84% (14%)	14% (19%)	2% (3%)	1% (1%)

Table 5. Application Function Results of the whole code [Samoladas *et al.,* 2003]

The study also includes representation of evaluation elements of source code metrics such as comments frequency, number of statements and number of paths.

Metric Name	Max	Min	Classes out of limits (%)
Class comments frequency	+∞	0.20	1.55% (1.17%)
Number of base classes	3	0	24.03% (10.53%)
Number of children	2	0	4.03% (3.12%)

Table 6. Classes Metrics Results of the whole code [Samoladas *et al.,* 2003]

Metric Name	Max	Min	Functions out of limits (%)
Comments frequency	+∞	0.50	17.88% (22.52%)
Number of statements	20	1	10.31% (13.33%)
Cyclomatic complexity	10	0	4.54% (5.06%)
Number of out statements	1	0	12.50% (17.58%)
Number of PATHS	60	1	4.56% (4.72%)

Table 7. Function Metrics Results of the whole code [Samoladas *et al.,* 2003]

The importance of this study lies on the idea that there can be efficient ways to evaluate Free/Open Source Software by means of external tools or developed models. Similarly the results of the study supports that successful and reliable FOSS can be developed in unusual genres for the FOSS while knowing that evaluating their reliability accurately is still hard. The outputs of the evaluation tool also seem to be accurate when we consider that the FOSS ERP/CRM tool in discussion has over 495,000 downloads at the time of the study, making it the most popular FOSS ERP/CRM tool. Another interesting focus of this case-study also simulates how much the FOSS ERP/CRM would cost if it was developed in a proprietary way and the cost was found to be 385,000-440,000 Euros with a development period of 8 man years. However the high quality source code is not the only factor affecting the success and reliability of an ERP/CRM system where further studies with different aspects are needed.

5. Conclusions

In this thesis study different software reliability metrics, related models based on these metrics and different OSS paradigms were analyzed targeting to achieve reliability in OSS. All these themes are developed to determine whether an OSS application being developed is ready for use and how it performs compared to similar peers. Even after performing these metrics, following the proposed models and methods, it is really hard to decide that an OSS is fully reliable and to choose the viable software. We have also seen that the current state of Software Reliability definition needs to be improved for FOSS. The case-studies also showed that evaluating the reliability of FOSS is a tough process and it is almost impossible to find a unified approach to evaluate each and every FOSS product. Following my analysis on the state of art and available proposals about the OSS paradigms we can summarize the properties of reliable OSS based on the categories provided by Metcalfe [Metcalfe, 2004] and I will suggest that a successful open source software reliability model should take the following points into consideration:

Reputation

Successful software brings up reputation for its name and the developers among the similar alternatives in the market. In the open source case, the reliability of the software increases with the contribution of developers with high reputation. The quality of the output increases as we discussed about the developer contributions and community influence.

Ongoing Effort

The evidence of ongoing effort is an indicator of an effort of creating quality software. Number of bug fix requests, fixes applied, implemented features and active communication channels should be considered as input data for reliability metrics. These ongoing efforts also encourage the participation of further developers, resulting in easier bug spotting, faster code implementations and finally leading to a more robust and reliable software.

Standards and interoperability

The target product should implement the open standards and the product lifecycle and characteristics should be planned based on these standards. In addition to this, interoperability with other software is crucial since an ongoing open source project can be embedded into further projects or used simultaneously with another one. As we discussed in the reliability metrics and models such as reliability blocks and Markov

model, individual reliability of components directly affects the reliability of the system they are used in.

Support (Community)

The open source project should have a remarkable and active support community in order to assess the quality evaluation. Again the frequency of bug reports and response times, number of contributors, developer profiles and documentation can be the evaluation criteria for open source software while creating a reliability model.

Support (Commercial)

Similar to the community support, the reliability and quality of an OSS project can be assessed by the availability of commercial support. However, commercial support is usually available for more widely used products and bigger companies. This kind of support can be an extra or meta evaluation criterion in an OSS reliability model.

Version

The release of a new version usually comes up with solutions to existing bugs in the previous version. Extra feature requests can be handled, existing features can be enhanced and code quality can be increased. Hence, the overall quality and reliability of the OSS increases. In order to be used for reliability metrics or evaluation criteria of an OSS reliability model, relevant data about the releases such as number of versions and bug fixing frequencies are fundamental.

Documentation

For all the current developers, further contributors and end users, documentation is a central resource to assess the usability and quality of the software. Furthermore, from the documentation the history of bug fixes, implemented features and open issues can be traced. In addition, the overall characteristics of the end product can be derived from the documentation and the evaluation of how much it satisfies the user needs and how reliable it is can be concluded. For an OSS reliability model the quantity of available documentation and the depth of functional information can be used as evaluation criteria.

Coverage

The resulting product of an OSS project must be specialized before use. This means narrowing down the range of tasks the software can handle for a specific purpose or user need. As a result of specialization the effort for constructing and deploying it will reduce and the efficiency will increase and therefore the overall performance and reliability of the software will increase. Maintainability and modularity of the code will

also be affected in a positive way, increasing the user satisfaction levels. In an OSS reliability model, relevant data for coverage, maintainability and modularity can be used as evaluation criteria.

Based on the findings of the use cases, suggestions for improvements on the software reliability definition and ideas for a model for FOSS reliability, further research and development can be performed. It is really hard to create a unified software reliability model for different open source software development paradigms. However it is possible to improve current approaches for evaluating FOSS reliability, providing more and accurate options for evaluating a wide range of FOSS products.

References

[Antoniades *et al.*, 2002] Ioannis P. Antoniades, Ioannis Stamelos, Lefteris Angelis and Georgios L. Bleris. A novel simulation model for the development process of open source software projects. *Software Process Improvement and Practice*, **7**:173-188, 2002.

[Antoniades *et al.*, 2007] Ioannis P. Antoniades, Michalis Kontoyiannis, Ioannis Stamelos and Ignatios Deligiannis. Simulation of the temporal evolution of OSS projects: application to XMMS and MPLAYER. *Proceedings of the 2nd International Workshop on Public Data about Software Development. Limerick, Republic of Ireland. June 14,* 2007.

[Compiere 2011] Compiere Open Source ERP and CRM Software. http://www.compiere.com/

[Crowston and Scozzi, 2002] Kevin Crowston and Barbara Scozzi. Exploring the strengths and limits of open source software engineering processes: A research agenda. *Second Workshop on Open Source Software engineering. Orlando, Florida. May 25,* 2002.

[Farber, 2004] Dan Farber. Six barriers to open source adoption, *ZDNet Tech Update*, March 2004. Available as: http://www.builderau.com.au/strategy/businessmanagement/soa/Six-barriers-to-open-source-adoption/0,339028271,320283261,00.htm

[Goel and Okumoto, 1979] Goel, A. L., and Okumoto, K. A Markovian Model for reliability and other performance measures of software systems. *Proceedings of National Computer Conference*, pp. 769-774.

[IEEE 610.12-1990] IEEE Std 610.12-1990 IEEE Standard Glossary of Software Engineering Terminology – Description. Available as:

http://standards.ieee.org/reading/ieee/std_public/description/se/610.12-1990_desc.html

[Immonen and Palvainen, 2007] Anne Immonen and Marko Palvainen. Trustworthiness evaluation and testing of open source components. *Proceedings of the Seventh International Conference on Quality Software, QSIC2007. Portland, Oregon, USA.* 11-12 October 2007.

[Isitan *et al.*, 2011] Kemal Isitan, Timo Nummenmaa and Eleni Berki. Openness as a method for game evolution. 5 pages. To appear in *Proceedings of the IADIS International Conference on Game and Entertainment Technologies 2011.*

[Lee *et al.*, 2008] Wangbong Lee, Boo-Geum Jung and Jongmoon Baik. Early reliability prediction: An approach to software reliability assessment in open software adoption stage. In: *Second International Conference on Secure System Integration and Reliability Improvement. Yokohama, Japan.* 14-17 July 2008.

[Logiscope 2011] Rational Logiscope by IBM. http://ww.ibm.com/software/awdtools/logiscope/

[Lyu, 1996] Michael R. Lyu. *Handbook of Software Reliability Engineering*. IEEE Computer Society Press, New York, San. Francisco, et al., pp. 3-6, 257-262. 1996.

[McCabe, 1976] Thomas J. McCabe. A complexity measure. *IEEE Transactions of Software Engineering*, December 1976.

[Metcalfe, 2004] Randy Metcalfe. Top tips for selecting open source software. OSS Watch. Retrieved March 10, 2008. Available as: http://www.oss-watch.ac.uk/resources/tips.xml

[Metcalfe, 2006] Randy Metcalfe. Open source projects: criteria for success. OSS Watch. Retrieved March 10, 2008. Available as:

http://www.oss-watch.ac.uk/resources/projectsuccess.xml

[Musa, 1980] Musa, J.D. The measurement and management of software reliability. *Proceedings of the IEEE,* **68**(9), September 1980.

[Navica, 2007] Navica Open Source Maturity Model Website. http://www.navicasoft.com/pages/osmm.htm

[OSI, 1998] Open Source Initiative Website, The Open Source Definition. Retrieved October 29, 2010. Available as: http://www.opensource.org/osd.html

[Samoladas *et al.,* 2003] Ioannis Samoladas, Stamatia Bibi, Ioannis Stamelos and Georgios L. Bleris. Exploring the quality of free/open source software: a case study on an ERP/CRP system. *9th Panhellenic Conference in Informatics, Thessaloniki, Greece. November,* 2003.

[Samoladas *et al.,* 2004] Ioannis Samoladas, Ioannis Stamelos, Lefteris Angelis and Apostolos Oikonomou. Open source software development should strive for even greater code maintainability. *Communications of the ACM*, **47**(10):83-87, October 2004.

[Stamelos *et al.,* 2001] Ioannis Stamelos, Lefteris Angelis and Apostolos Oikonomou. Investigating the relationship between source code modularity and user satisfaction in open-source. *Proceedings of the 1st Panhellenic Conference with International Participation on Advances in Human-Computer Interaction*, *Patras, Greece*. pp. 77-81. December 2001

[Wasserman, 2002] Wasserman, Gary. *Reliability Verification, Testing, and Analysis in Engineering Design*. Marcel Dekker Incorporated, New York, NY, USA, pp. 2-10. 2002

[Wheeler, 2007] David A. Wheeler. Why Open Source Software / Free Software (OSS/FS, FLOSS, or FOSS)? Look at the Numbers!.

Retrieved January 10, 2010. Available as:

http://www.dwheeler.com/oss_fs_why.html

[Whittaker and Voas, 2000] James A. Whittaker, Jeffrey Voas, Toward a More Reliable Theory of Software Reliability. *Computer,* **33**(12): 36-42, December 2000.

[Wikipedia – Methodologies, 2010] Open Source Software Assessment Methodologies. Retrieved November 21, 2010. Available as:

http://en.wikipedia.org/wiki/Open_source_software_assessment_methodologies

[Wilson, 2006] James A. J. Wilson. Open source maturity model. OSS Watch. Retrieved March 10, 2008. Available as:

http://www.oss-watch.ac.uk/resources/osmm.xml

[Wu and Lin, 2001] Ming-Wei Wu, Ying-Dar Lin, Open Source Software Development: An Overview. *Computer*, **34**(6): 33-38, June 2001.

[Xie et al., 2004] Xie, M, Kim-Leng Poh, Yuan-Shun Dai. *Computing Systems Reliability: Models and Analysis*. Hingham, MA, USA: Kluwer Academic Publishers. pp. 71-113. 2004.

[Yeates, 2005] Stuart Yeates. Open source software and security. OSS Watch. Retrieved March 10, 2008. Available as:

http://www.oss-watch.ac.uk/resources/security.xml

[Zhou and Davis, 2005] Ying Zhou and Joseph Davis. Open source software reliability model: An empirical approach. *Proceedings of the Fifth Workshop on Open Source Software Engineering. ACM Press,* **30**(4):1-6, 2005.