Certification of open-source software: A role for formal methods?

Luis S. Barbosa[1], Antonio Cerone[2], Alexander K. Petrenko[3] and Siraj A. Shaikh[4]

[1] CCTC & Dep. of Informatics, Minho University, Braga, Portugal
`lsb@di.uminho.pt`
[2] International Institute for Software Technology, United Nations University, Macau SAR China
`antonio@iist.unu.edu`
[3] ISP RAS, Russian Federation
`petrenko@ispras.ru`
[4] Department of Informatics and Sensors,
Cranfield University, Defence Academy of the United Kingdom
Shrivenham, SN6 8SA, United Kingdom
`s.shaikh@cranfield.ac.uk`

Abstract. Despite its huge success and increasing incorporation in complex, industrial-strength applications, open source software, by the very nature of its open, unconventional, distributed development model, is hard to assess and certify in an effective, sound and independent way. This makes its use and integration within safety or security-critical systems, a risk. And, simultaneously an opportunity and a challenge for rigourous, mathematically based, methods which aim at pushing software analysis and development to the level of a mature engineering discipline. This paper discusses such a challenge and proposes a number of ways in which open source development may benefit from the whole patrimony of formal methods.

Keywords: Open source software certification; formal methods; software quality.

1 Introduction: The certification challenge

The answer is yes.
But could you please repeat the question?
(Woody Allen, 1985)

Proved successful over the years, Open Source Software had a global impact on the way software systems and software-based services are developed, distributed and deployed. In particular, it has delivered some high-quality operating systems, such as GNU/Linux and FreeBSD, middleware, such as the Sendmail mail server, the Apache web server, the MySQL data base management system and the Samba network protocol, as well as very popular application software, such as OpenOffice and the Mozilla browser. All of them remain, among many other examples, as a testimony to its success and resilience.

Widely acknowledged benefits of open source software include reliability, low development and maintenance costs, as well as rapid code turnover. Moreover, companies become aware that integrating open source software into commercial products, made available by liberal open source software licences, reduce development costs while offering usually high-quality, extensively tested components. Furthermore, Governments all over the world are becoming aware of the growing dependence on proprietary formats and software in their administration, and regard open source software as a warranty of technological independence, which turns out to a strategic advantage, mainly in the developing world.

However, state-of-the-art open source software, by the very nature of its open, unconventional, distributed development model, makes software quality assessment, let alone full certification, particularly hard to achieve and raises important challenges both from the technical/methodological and the managerial points of view.

This makes the use of open source software, and its integration within industrial-strength applications, with stringent safety or security requirements, a risk. And also a serious concern because, as recently reported in the Gartner Report [29], *spending on software related to Linux server platforms is on a compounded annual growth rate of 35.7 percent from 2006 to 2011 and (...) by 2012, more than 90 percent of enterprises will use open source in direct or embedded forms.*

This state of affairs has been identified either in the open source software community (as witnessed, for example, in an increasing number of mentions in recent editions of the *IFIP Conference on open source software*) and in the formal methods communities (open source software as a target domain for formal methods was recognised in recent editions of prestigious FM and SEFM international conferences). The relevance of this problem is further emphasised by a number of panels organised in open source software forums (eg, the OSS Watcher) and industry oriented initiatives. Finally, a series of workshops promoted by the United Nations University, with the acronym OpenCert, have addressed specifically this challenge since 2007.

Actually, OpenCert stands for a broader initiative aiming at launching an informal network of people from Academia and Industry interested in strengthen the role of open source software by applying formal methods to its certification. The possible institution of an international certification authority for open source software, has been occasionally mentioned as a long term objective.

This paper, echoing a number of questions addressed in the OpenCert workshops, aims at contributing to a wider discussion on which role formal methods, i.e. methods rooted on mathematical semantics for computing phenomena, can play in open source software certification and in what ways can such a role be made effective.

The relevance of formal methods for open source software is discussed in the following section. Section 3 addresses quality models and certification of open source software, paving the way for a more detailed discussion on three complementary ways formal methods can be adopted to increase confidence on open source software components: model-based testing, support to program understanding and support to system development. Each of these topics is addressed separately in sections 4, 5 and 6, respectively.

2 Why formal methods?

The use of precise and mathematically sound techniques to design and engineer software is advantageous for a variety of reasons. A well-defined notation allows an expression of requirements in a manner that is both clear and comprehensive, resulting in a formal specification. A model of the system can then be developed and checked for correctness with respect to it. This is achieved by a variety of means. One approach is to use *theorem-proving* where logical axioms and inference rules are used to construct a proof, usually in a semi-automatic procedure. Another approach is to use *model-checking* whereby an exhaustive search of all possible states of the system is performed to demonstrate whether it is correct; one advantage of this approach is that in case the system is flawed then a counter-example is produced, which can be helpful in determining where the flaw is in the system. Of interest here is not only correctness, but also (usually intricate) properties of the system that need to be established. For example, system designers are often interested to check whether a concurrent system is deadlock-free. Such questions have to be answered before a system is implemented and operational, and, in most cases, can only be done so if the system is modelled and designed formally. Development of critical and dependable software therefore necessitates a formal approach.

The software industry, by and large, so far has either failed to recognise the effectiveness of formal methods or levied criticism on the inaccessibility and cost of using them [15,3]. Confusion also abounds on the exact role that formal methods should play in the software development process [27]. There is emerging evidence, however, that formal methods are being adopted in the industry and deliver on the promise [16]. Examples of successful industrial use are available for a range of formal techniques, from model-oriented frameworks such as VDM [14,27] to process algebras such as CSP [26,11]. Some of the criteria for effective use in the industry include introducing formal methods incrementally [21], at an early stage of requirements analysis [16] and applied to selected, possibly critical, components avoiding overuse [4,5]. The merits of good tool support and early exposure in terms of training and curriculum have also been highlighted [14].

Recently a number of experiments of application of formal methods to open source software have been reported in the literature. Such efforts have come with a realisation that formal verification could significantly raise confidence in the design of open source software. So far, however, they have target a low level of abstraction, acting directly over source code, since all of it is available in open source software projects, and addressing its verification against the absence of deadlocks [7], memory access violations [8] and others [18]. While such experiments would be of interest to developers, who are likely to prefer a post-hoc detection and correction of bugs, a range of possiblities could open up if formal methods were introduced at the early specification and design stage: both to help in requirements analysis and producing provably correct design. Not only would this reduce bugs and vulnerabilities in the code produced, but also serves to provide means for easier certification. Proposals as in [9] to encourage the use of formal methods and integrate it in the development cycle of open source software also highlight certification as a useful by-product. Such use could help in certification in two main ways [33]. First, the results of formal verification can serve as evidence to build up an assurance case for a system. Such evidence could then be tied to varying levels of assurance requirements as part of a certification infrastructure. Second, formal development and analytical tools can be used in the development phase to construct software correctly; this approach has also come to be known as *correctness by construction* [17]. The use of such an approach – the process – can then be mandated in a certification system.

Finally, to answer the question, we propose that open source software undoubtedly stands to benefit from formal methods. This will allow more rigorous and automated support in the development process to assure safety and dependability attributes. Requirements engineering, software design and development, revision and maintenance and, documentation are all to benefit from the use of formal methods. All this becomes even more important considering the reputation that open source software has built up over the years, which gave industry leaders confidence that the open source community could deliver much more. Spiralling costs of development, innovative developmental approaches, user testing and feedback, and project durability and sustainability are other motivating factors for industry leaders to look to the open source community for this purpose.

3 Quality and certification for open source software

It should be clear at this point that a proper, scientific approach to software certification has to be framed into a solid mathematical framework. Such is, as discussed above, the rationale underlying this broad, but precise, family of semantic-based techniques known as *formal methods*. The particular nature of open source development does not make the resulting software an exception to this rule. But, on the other hand, entails the need for a careful assessment of its application.

This section intends to clarify at which point formal methods become essential to a certification process for open source software. Later, sections 4, 5 and 6 will address how this can be achieved.

Actually, open source software certification can be discussed at several different, yet interrelated, levels. Since 2007, in the series of United Nations University sponsored OpenCert workshops some of these levels have been identified and a number of questions formulated. In particular,

– Which quality models are suitable for open source software and which sort of certification process can assess each of them?

– Which strategies for carrying-on the certification process?
– Which certification infra-structures?
– Which business model to support it?

We shall briefly address the first of them in the sequel; the remaining ones will be tackled, in a tentative way, along the paper.

When discussing open source software certification a first distinction has to be drawn between what counts as an object of certification: the product of a development process or the process itself? *Context certification* is clearly in the process side. Its object is the context of software production (people, process, projects), i.e., a number of factors related to how software is developed, rather than to the software itself. Each of them are well characterised in the literature and often associated to specific standards and certification processes.

Context certification, quite popular in the proprietary software industry, is hardly applicable to open source software development which is highly decentralised, opposed to regulations, understood as possible curtailment of freedom, and lacks central management which makes it difficult to define a standard that could suggest indicators of the technical rigour used by a distributed community of volunteers and identify the human processes involved. However, efforts to tune context certification to open source software, lead to the inclusion in a recent proposal of a open source software quality model [35] of so-called *quality by access* (related to availability of source code from an easily accessible medium) and *quality by development* (related to the efficiency of the entire development process).

On the other end of the spectrum is *product certification*, which addresses both *technical quality* and *design quality*. The former is related to factors directly influencing maintainability, reliability and portability (seeking answers to the question *how well constructed the software is?*). Therefore, it constitutes a direct concern for developers, rather than for end-users. As recognised in [10], technical quality *may become, in the longer term, the largest contributor to the total cost of ownership of a software system* [because] *software with high technical quality can evolve with low cost and low risk to keep meeting functional and non functional requirements.* This reference proposes a layered model for measuring and rating technical quality of open source software in terms of norm ISO/IEC 9126, based on source code metrics, computed automatically.

Design quality, on the other hand, is related to the certification of functionality (*does the software satisfy its functional requirements?*) and, in general, to software correctness. This is the specific target of formal methods, although application can be done with variable degrees of strictness. The following sections make the case for formal approaches to design quality certification from *model-based testing* and *code analysis* to *verification* and *formal development.*

Finally, an increasingly important certification layer in open source software project, which curiously, involves aspects of both technical and design quality, is related to *security.* Security certification, involving compliance to the Common Criteria (ISO/IEC 15408), is fundamental to foster open source software adoption in critical markets, such as telecommunications, and military. Reference [12] provides an updated and extensive discussion of this issue.

4 Open source software and advanced testing techniques

Testing is perhaps the most popular framework for program analysis and, simultaneously one in which a fruitful connection to formal modelling can be easily obtained. In brief, tests can be generated, in a quite effective form, from formal models.

This section considers two questions relating testing and open source software: whether modern testing techniques are needed by the open source software community, and what specific value they could bring in. To bound the scope of the discussion, we restrict ourselves to functional testing and test development leaving out monitoring and debugging, as well as fault tolerance, performance and security testing.

The topic is not rhetoric as lots of open source software projects utterly ignore most software quality issues. And often the community itself puts under question the need for testing in open source software projects. The most cited expression of such doubts is *The Cathedral and the Bazaar* by Eric S. Raymond [28]:

> Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone.
> Or, less formally, “Given enough eyeballs, all bugs are shallow.” I dub this: “Linus’s Law”.
> My original formulation was that every problem “will be transparent to somebody”. Linus demurred that the person who understands and fixes the problem is not necessarily or even usually the person who first characterizes it. “Somebody finds the problem,” he says, “and somebody else understands it. And I’ll go on record as saying that finding it is the bigger challenge.” (...) But the key point is that both parts of the process (finding and fixing) tend to happen rapidly.

Nevertheless, some of the most successful open source software projects, devoted to the development of very complex software put a lot of their effort into testing and quality assurance, just because of the inherent high complexity of products to be developed. Examples of such projects are Linux Kernel [40] and GCC [41][5]. In such cases testing and quality assurance are necessary for project evolution. Moreover, the need for advanced testing techniques is growing with the growth of software complexity.

[5] Note that the GCC project uses assertions spread throughout the code. This practice significantly helps application of verification techniques, since a first step towards domain formalization is already done. Formal methods can be applied with less effort on the base of properties and requirements formalized by domain experts.

Still another reason for the case of using advanced testing in open source software projects is their decentralised environment in which traditional testing and quality assurance techniques used in proprietary software development do not work properly. Open source software projects need specific technologies to make development and code base consistent, to check their conformance to business requirements and standards. Without technological support for quality assurance complex software projects cannot exist.

What values can be brought into open source software projects by advanced testing techniques? An example of such a technique is model based testing (MBT), sometimes almost identified to specification based testing (SBT) approaches. Both require to construct tests on the base of a formal model of the behavior of the system. Note that 10-15 years ago formal description of a complex system behavior seemed to be hardly possible. First applications of MBT targeted only critical software and hardware components, not whole systems. Models were represented mostly as finite state machines, labeled transition systems or Petri nets. These formalisms are useful to model simple low-level protocols, processor units, simple software modules, but become unsuitable for the description of operating systems, database management systems or compilers, which requires sufficiently detailed models for the construction of non-trivial tests.

MBT also used specialized languages and notations, such as B, Lustre, Larch, SDL, MSC. Some of these notations are textual, some graphical, and finally some others, like SDL or EXPRESS, resort both to text and graphics. MBT was quite successful in specific domains including telecommunication protocols, critical parts of real-time and embedded systems. Transition to wider domains was hindered by two reasons: poor expressivity of the modelling notation and too complex relations between models, systems, and tests. These problems motivated the development of more effective and powerful MBT techniques and tools. For example, ASN.1 was introduced to describe complex data structures of protocol messages in addition to sequences of actions described by SDL. To simplify establishing relationships connecting models and implementations, extensions to programming languages were suggested (e.g., ADL [34], JML [22], UniTESK [20], Spec# [1]). Actually, it seems that complex systems can hardly be described with the help of specific, thin scope languages. But extensions to programming languages used in system's source code can cope with this and may be used successfully for modelling and test construction.

Professional testing is always targeted to assessment of some quality characteristics of the system under test. We consider here only functional testing, but functionality itself has various aspects. Interoperability tests and standard conformance tests, sanity tests, tests based on common use cases and tests targeted to "dark corners" corresponding to rarely used but rather complex cases, are very different. The same is true for unit tests, integration tests, system tests based on GUI or input languages like SQL for DBMS. Two main approaches exist for test adequacy evaluation: one resorts to coverage of functional requirements and situations of their interaction, the other is based on coverage of structural elements of the system up to single statements and control or data flow transitions. So, test generation and test adequacy evaluation mostly use the structure of requirements model and structure of the system itself. Knowledge on probable or usual errors and defects (the so-called error or fault model) can be used as additional source of quality assurance.

From the technical point of view, open source software projects can resort to the complete spectrum of test development and execution tools. Still practice shows that even mature open source software projects have only centralized support for basic system tests. There are several causes for this situation: immature technologies capable to maintain huge test suites for rapidly evolving software; weak rationale for resource spending on unit tests when their results have no direct influence on the overall quality of the system. Furthermore there is a common belief that system tests are more cost-effective while validity of this statement is debatable.

A real-world example of an open source software project using advanced testing technologies is the LSB Infrastructure Program [42]. This project aims at developing and disseminating the standard called "Linux Standard Base" (LSB), whose purpose is to increase Linux application portability across Linux distributions, to reduce market fragmentation for Linux distribution and application vendors, and to improve maturity and stability of the overall Linux platform. The LSB Infrastructure Program collects data about the actual state and trends in development of the most popular Linux libraries; the collected information is used to add interfaces (functions, types and global variables) of new libraries to the standard and to define a specification of each interface. Even if early specifications incomplete and informal, at a later stage, according to the workgroup priorities, they get more complete and, ultimately, formal. As specifications mature, so grows the quality of test suites checking conformance of Linux distributions against the standard. At present a complete formal specification is available for LSB Core group of libraries. Specifications and tests generated from them are published on the web-site of the OLVER project [43], originally supported by the Russian Ministry of Science and Education.

The OLVER project gathered a valuable experience in the practical use of formal methods for open source software. The underlying platform is based on UniTESK technology. Behavior of functions in LSB Core is specified in the form of pre- and post-conditions using the specification extension of the C programming language as a modelling notation. Tests are generated by the CTESK tool based on test scenario descriptions. Usually these scenarios do not describe directly a test case. A typical test scenario describes a simplified state structure of an abstract FSM and input iterators for SUT operations. Such description can be used for test generation on-the-fly and allows to achieve both an effort reduction in test design and good test coverage quality.

We shall now introduce some statistics with respect the OLVER project. Over 1500 interfaces (C functions) were formally specified and tested. Typical values are as follows: size of specification is 100 Kloc, size of test scenario description is 50 Kloc, size of implementation under test is 280 Kloc (taking into account only the size of glibc library and disregarding part of implementation located in the Linux kernel). Specification and test scenario design effort (including test debugging, documentation, publication of the testing result) is about 3 person*day per 1 interface. This project has allowed to discover over 150 errors in standards and over 50 errors in Linux distributions (excluding a lot of errors in mathematical functions, see details in [19]). Most of the errors detected were published on the OLVER web site.

It is important to note that interest of practitioners, such as Linux Foundation members, was raised not by its formal, strict and complete specifications nor by the quality of tests. Actually, what distinguishes the test suite and project infrastructure from other related projects, is the systematic approach to total standard analysis, the elicitation of atomic requirements and their cataloguing along with providing the necessary infrastructure for requirements traceability. The systematic nature of the project, typical for formal methods applications, turned out to be the key factor determining the subsequent development of the LSB Infrastructure Program. It is worth noting that another reason for positive acceptance of OLVER relies on the use made of a C extension for specification purposes, relatively easy to understand. Current activities in this program go in two direction. The purpose of the first one is to perform an initial study of existing documentation and to elicit and collect atomic requirements. The second direction aims at formalising requirements of the selected interfaces. Formal specifications are used as a basis for the development of test scenarios, which results in tests. Note that it is impossible to entirely decouple the development of specifications from that of test scenarios, because they are debugged jointly and usually simultaneously. Both of them, however, can be claimed correct (and complete) only after actual test execution. Specification and test development for complex interfaces are more error-prone than implementation development.

Verification technology and process becoming mature paves the way for establishing a certification procedure. Actually, the Linux Foundation has a remarkable experience of a certification system deployment. The system provides services to certify LSB conformance for both Linux distributions and Linux applications (see [32] for details). LSB is not the only standard that can demonstrate a positive experience of an industrial-scale formal methods application. Reasonable candidates for formalization are Internet standards, documentation standard (like ODF), and programming language standards. Many standards, especially new and immature ones, contain a significant amount of errors. Even in mature standards, as POSIX, we have detected errors. Actually, revealing errors in standards is another important output of formal based projects.

Formalization could be well accepted beyond software — the movement for open standards and open hardware designs (see, for instance, [39]) will eagerly apply formal methods as soon as they get close to problems and practical needs of the corresponding industry sectors.

5 The ‘backward’ perspective

If rigourous modelling plays a fundamental role in test planning and generation, ensuring higher levels of design quality entails the need for the incorporation in the open source software lifecycle of formal methods for verifying or enforcing software correctness with respect to some specification of its intended behaviour. Typically, formal methods are designed to be applied during the development phase, preferably from very early design stages. Difficulties and strategies for proceeding this way in open source software development are addressed in the following section. For the moment, however, we shall discuss the complementary, ‘backward’ perspective: that is, the use of such methods to assist the re-engineering process of running code.

Actually, faced with a high risk dependence on legacy software, managers are more and more prepared to spend resources to increase confidence on - i.e. the level of understanding of - their code. As a research area, program understanding affiliates to reverse engineering, understood as the analysis of a system in order to identify components and behaviours to create higher level abstractions of the system. If forward software engineering is often regarded as an almost lost opportunity for formal methods (with notable exceptions in areas such as safety-critical and dependable computing), its converse looks a promising area for their application. This is due to the complexity of reverse engineering problems and exponential costs involved. In such a setting, the same principles and calculi used for program developing can be used, applied in the reverse direction, from concrete to abstract models, for understanding and documenting implementations.

Such techniques seem promising for the whole process of open source software certification because

- they do not interfere in the development process, and therefore have no negative impact on the community practices,
- and, on the other hand, full access to source code enables the effective application of approaches and tools entirely based in code analysis.

In practice, the success of formal methods used this way requires a suitable combination with other, often rather informal, techniques for code analysis. The latter are intended to prepare the way to the former. Furthermore, the nature of open source software entails the need for integration of techniques spanning the ‘micro’ to the ‘macro’ levels (e.g., from code slicing to architectural recovery) and with different levels of formality (e.g. from statistical analysis based on code metrics to the identification and formal verification of hidden invariants). Let us briefly mention a few such techniques which could play a role in open source software certification.

Architectural reconstruction. Current software development rely more and more on non trivial coordination logic for combining autonomous services often running on different platforms. Open source software is no exception. As a rule, however, in typical, non trivial software systems, such a coordination layer is strongly weaved within the application at source code level. Therefore, its precise identification becomes a major methodological (and technical) problem which cannot be overestimated along any program understanding process.

Open access to source code provides an opportunity for the development of methods and technologies to extract, from code, the relevant coordination information. Note that open source software applications often emerge by composition of multi-source, heterogeneous and previously unrelated pieces of code, which makes architectural recovery processes both useful and challenging. Moreover, there is a need, particularly critical in open source software contexts, to control architectural drifts, i.e., the accumulation of architectural inconsistencies resulting from successive code modifications. References [31,30] report on a technique which adapts typical program analysis algorithms, namely slicing [37], to recover coordination information from legacy code. This is based on a notion of *coordination dependence graph*, a specialisation of standard program dependence graphs [13] used in classical program analysis. The recovered coordination patterns are automatically expressed in ORC, a formal orchestration language developed by J. Misra and W. Cook [25]. ORC specifications are amenable to formal reasoning (e.g. to compute equivalent implementations) and can be animated to simulate such patterns and study alternative coordination policies.

Type reconstruction. Type reconstruction is useful in understanding programs written in untyped or weakly typed languages, such as ASP, VBScript, JavaScript, Tcl/Tk, and Perl. The approach proceeds by discovering type relations between variables via static analysis of the program code (e.g., an assignment of a variable to another may assert a subtype relation). The technique worked out for reconstruction of types from Cobol legacy systems [38], and is also used in decompilation. It has a great potential for application to heterogenous, multi-lingual systems.

Logic mining and documentation analysis. Logic mining refers to (semi-)automatic extraction of business rules (domain knowledge) from code to improve the results of data reverse engineering. Such rules characterise conditions in the code and abstract them into logic properties, which are then verified as invariants to be maintained by the application.

Documentation analysis has, however, a broader scope. It relies on the basic observation that besides source code, the fundamental source of information about open source software lies in documentation which is usually spread over partially completed reports, unstructured code annotations, discussions on mailing lists or wikis, etc. Such documentation is typically produced and edited by several people. Research in this area aims at developing techniques and tools to analyse and extract information from open source software documentation and to render it in a form useful for reconstructing program meanings.

Applying formal methods 'backwards' requires, however, specific ways to smoothly integrate them into the open source software very peculiar development process without disturbing its collaborative, distributed and heterogeneous character. This means to establish feedback loops in open source software development, making publicly available a number of interrelated analysis tools which could be used in several different ways by the open source software community. A possibility worth to study would take the form of an online infrastructure – a *certification portal* – in which independently developed analysis tools, with different levels of sophistication, could be inserted for monitoring, assessment and, at a later stage, certification of open source software products. Such an infrastructure will allow for the registration of open source software projects, their source code visualisation and analysis at different levels, as well as the rendering of analysis results in suitable, flexible formats to both open source software developers and users. Actually, the portal would not only provide support for open source software analysis, but also make the evolution of open source software projects clearly visible to the open source software community. In the long run, it is expectable that, feedback loops made possible through it, would have some effective impact in the overall quality of open source software products, with minimal intrusion on the peculiar, but successful open source software development life-cycle.

6 The 'forward' perspective

The success and popularity of open source software is only partly due to the great advantage of low cost products, for which open source licences prevent developers from charging for software distribution. In fact, open source products often bear higher reliability, efficiency, usability, and in general a better quality than functionally equivalent proprietary software. This is in some way surprising, given the peculiarity of the open source development process, which counts on a distributed community of free developers governed through a "democratic" leadership rather than by a strictly "autocratic" central management as usual in software industry.

The lack of central management would suggest the absence of a rigourous verification process, based on planning and carrying out accurate review and sophisticated testing, of software product on one hand. On the other hand, the community involved in the open source development is very complex; it involves not just developers but also users, reviewers and testing specialists, with major intersection among these categories of community members.

Thereby, in his famous book *The Cathedral and the Baazar* [28], cited above, Raymond suggests that the high quality of free software is partly due to the high degree of peer review and user involvement. Although this hypothesis is based on anecdotal rather than empirical evidence and has not been tested through empirical data analysis, it is widely accepted in the open source community: the high quality of software produced through a open source development is a fact that nobody can deny, though we are still far from understanding all factors that contribute to the emergence of such high quality in the final product [36].

Raymond's hypothesis, however, even if it were confirmed through a rigorous empirical analysis, would not be sufficient a guarantee for applications in domains such as industrial process control, transportation, avionics, e-commerce, health and defence, for which security and/or safety are central issues. Government regulations and international standards actually require security- or safety-critical software to undergo a certification process, which includes a thorough verification where each and every functional and operational property receives rigorous treatment. Standards more and more often include explicit recommendations for the use of formal methods, at least in the verification of the most critical system components [2]. However, today we lack standards and methods to be used in a certification process to assess the quality of open source software. In fact, the lack of central management in open source software projects [23,24] makes it difficult to adopt an existing standard or to define a new standard that could suggest indicators of the technical rigour used by a distributed community of volunteers and identify the human processes involved in the project.

In this perspective, the use of rigourous review and formal verification directly on the source code, or at a model-level through reverse engineering techniques, as suggested in Section 5, would probably be ineffective not only due to the high complexity of security- and safety-ctitical systems, and the cost of fixing errors once the source code has already been produced, but also due to the complexity and nature of the development process itself. Once, errors are found in a very complex and critical end-product, it is not clear how to procede in fixing the errors.

One solution could be to pass the results and the error reports produced by the analysis back to the open source community involved in the development and expect that the community as a whole will activate the steps needed to fix the errors. This solution, however, appears as a big challenge. In fact, a large coordination effort is needed by the team leader to make sure that complete information about the nature, propagation and impact of the discovered error is effectively communicated in an understandable way to all relevant members in the community. Due to the great variety in backgrounds and skills among the community members it is virtually impossible to present such information in a format understandable to every member. In particular, it is expected that only a small number of members (if any) have some experience with formal methods, and this would create further problems if error reports refers to a reverse engineered model rather than to source code.

An alternative solution could be to fix the error outside the software development process, devoting an ad-hoc team to this purpose. Such a solution, however, would result in a development model that is not purely open source, possibly with a decrease in the above-mentioned quality aspects that usually emerge through open source development. As a secondary impact, this may even result in a decrease of trust in the product among users.

Therefore, although the use of a *certification portal for open source software*, as presented in Section 5, is definitely important as the final stage of a project and, possibly, as the final step of each product release, it must be complemented with the use of formal methods in a more integrated and distributed way. A possible way to achieve this goal is to incorporate within the open source development process formal design and verification methodologies whose successful use could itself increase the quality of the software produced.

Cerone and Shaikh [9] propose a 'pilot' open source project to facilitate such an effort. The most difficult task in incorporating formal methods within the open source software development process is to preserve the intrinsic freedom that characterises contributions by the volunteers who join the project. In fact, it would not be acceptable to explicitly enforce the use of a specific formal modelling framework to be adopted by all project participants.

In order to support open participation and, consequently, bottom-up organisation and parallel development, the project must introduce and present formal methods only as a possible but not mandatory option available to the contributors. On one hand this approach may require an additional effort by the leader team in integrating those contributions which do not make any use of formal method. On the other hand, it is also expectable that the challenge proposed by such an effort would attract new potential project volunteers who are keen to reverse engineer code, produced by other actors in a traditional way, into changes and extensions to the formal model. From this point of view we can say that the approach proposed in Section 5 can be incorporated, at least partly, within the forward perspective.

Therefore, the leader team should indeed propose one (or more) simple formal modelling framework(s) to be used within the project. However, the basic (extensible) model of the system should be proposed by the leader team both in terms of its requirement, informally expressed as usual in English language, and formally modeled using the proposed formal framework.

Critical to the success of such a pilot project is building an effective team that could provide the expertise both across the formal methods and open source domain. Relevant to this is also the choice of software or tool proposed. Finally, the proposed effort needs to be monitored for its progress and effective use of formal modelling and analysis during the development cycle.

7 Conclusions and future work

Where shall I begin, please Your Majesty?
Begin at the beginning, said the King.
(Lewis Caroll, 1865)

If, in concluding this paper, one is expected to answer the question in its title, we would certainly be affirmative. Not only there is a role to be played by formal methods in open source software certification, but also the concept of certification itself makes little sense if not framed in a rigourous setting, object of independent scientific scrutiny.

On the other hand, a lot of questions remain to be answered. A number of them are related to the concrete ways in which formal methods can be integrated in the open source software process: in a 'backward' or 'forward' perspective, as discussed above, focussing test generation or correctness verification, adopting a whole range of approaches, with a 'lighter' (as in [6]) or more strict (as in [9]) flavours.

Other questions concern the certification process itself, if by this we understand some form of automated analysis of source code performed by a trusted, independent agency. Actually, not only several technical and methodological issues remain to be tackled, but also the business model is unsolved (e.g., how will the process be managed and supported? who requests/provides the service? how can the certificates be trusted?, etc.)

Clearly, state-of-the-art open source software development has two main weaknesses. First, it lacks widely consensual, objective measures to assess product quality, let alone of effective, routine ways to compute them. Second, from a managerial point of view, open source software projects are hard to control and to predict due the lack of central management and defined responsibility. A strategy leading to the establishment of an independent certification process has potential for a long-term impact on the integration of trustworthy, open source software components, in large, complex systems. But it would certainly involve an effort to overcome the lack of specific certification standards to assess and classify the quality of open source software. How far we are from defining them, depends on building a solid understanding of the role of the open source software movement and of what makes an open source software project a success. More empirical studies, like those reported in [23], are needed in order to come up with a maturity model for open source software projects.

An important issue is, however, clear enough: whatever certification mechanisms are to be designed, they should smoothly integrate with open source software development, without disturbing the community and respecting its principles. Monitoring, rather than controlling, is the key idea. If this contributes to make evolution of open source software projects clearly visible to and traceable by the open source software community and to introduce control and feedback loops, supported by sound methodological principles, we will be already in right road.

Acknowledgements. L. S. Barbosa research was partially supported by the Cross project, under contract `PTDC/EIA-CCO/108995/2008`.

References

1. M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: An overview. In *Proceedings of CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer, 2004.
2. Jonathan Bowen and Victoria Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189–209, 1993.
3. Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.
4. Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, 1995.

5. Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods...ten years later. *IEEE Computer*, 39(1):40–48, 2006.
6. P. T. Breuer and S. Pickin. Approximate verification in an open source world. *Innovations in System and Software Engineering,*, 4(1):87–105, 2008.
7. Peter T. Breuer and Marisol García-Vall. Static deadlock detection in the linux kernel. In Albert Llamosí and Alfred Strohmeier, editors, *Reliable Software Technologies - Ada-Europe 2004, 9th Ada-Europe International Conference on Reliable Software Technologies, Palma de Mallorca, Spain, June 14-18, 2004, Proceedings*, volume 3063 of *Lecture Notes in Computer Science*, pages 52–64. Springer, 2004.
8. Peter T. Breuer and Simon Pickin. One million (loc) and counting: Static analysis for errors and vulnerabilities in the linux kernel source code. In Luís Miguel Pinho and Michael González Harbour, editors, *Reliable Software Technologies - Ada-Europe 2006, 11th Ada-Europe International Conference on Reliable Software Technologies, Porto, Portugal, June 5-9, 2006, Proceedings*, volume 4006 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2006.
9. Antonio Cerone and Siraj A. Shaikh. Incorporating formal methods in the open source software development. In Luis Barbosa, Peter Breuer, Antonio Cerone, and Simon Pickin, editors, *International Workshop on Foundations and Techniques bringing together Free/Libre Open Source Software and Formal Methods (FLOSS-FM 2008) & 2nd International Workshop on Foundations and Techniques for Open Source Certification (OpenCert 2008)*, pages 26–34, 2008.
10. J. P. Correia and Joost Visser. Certtification of technical quality of software products. In Luis Barbosa, Peter Breuer, Antonio Cerone, and Simon Pickin, editors, *International Workshop on Foundations and Techniques bringing together Free/Libre Open Source Software and Formal Methods (FLOSS-FM 2008) & 2nd International Workshop on Foundations and Techniques for Open Source Certification (OpenCert 2008)*, pages 35–51, 2008.
11. Sadie Creese. Industrial strength csp: Opportunities and challenges in model-checking. In Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors, *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004, Revised Invited Papers*, volume 3525 of *Lecture Notes in Computer Science*, pages 292–292. Springer, 2005.
12. Ernesto Damiani, Claudio A. Ardagna, and Nabil El Ioini. *Open Source Systems Security Certification.* Open Source Advances in Computer Applications. Springer Verlag, 2008.
13. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.
14. John S. Fitzgerald and Peter Gorm Larsen. Balancing insight and effort: The industrial uptake of formal methods. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*, pages 237–254. Springer, 2007.
15. Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.
16. Anthony Hall. Realising the benefits of formal methods. *Journal of Universal Computer Science*, 13(5):669–678, 2007.
17. Anthony Hall and Roderick Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, 19(1):18–25, 2002.
18. Alexey Khoroshilov and Vadim Mutilin. Formal methods for open source components certification. In Luis Barbosa, Peter Breuer, Antonio Cerone, and Simon Pickin, editors, *International Workshop on Foundations and Techniques bringing*

together Free/Libre Open Source Software and Formal Methods (FLOSS-FM 2008) & 2nd International Workshop on Foundations and Techniques for Open Source Certification (OpenCert 2008), pages 52–63, 2008.

19. V. Kuliamin. Test construction for mathematical functions. In K. Suzuki, T. Higashino, A. Ulrich, and T. Hasegawa, editors, *Testing of Software and Communicating Systems (Proceedings of TESTCOM/FATES 2008)*, volume 5047 of *Lecture Notes in Computer Science*, pages 23–37. Springer, 2008.
20. V. V. Kuliamin, A. K. Petrenko, A. S. Kossatchev, and I. B. Burdonov. The unitesk approach to designing test suites. *Programming and Computer Software*, 29(6):310–322, 2003.
21. Peter Gorm Larsen, John Fitzgerald, and Tom Brookes. Applying formal specification in industry. *IEEE Software*, 13(3):48–56, 1996.
22. G. T. Leavens, A. L. Baker, and Clyde Ruby. Jml: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.
23. Martin Michlmayr. Quality improvement in volunteer free software projects: Exploring the impact of release management. In Marco Scotto and Giancarlo Succi, editors, *Proceedings of the First International Conference on Open Source Systems*, pages 309–310, Genova, Italy, 2005.
24. Martin Michlmayr, Francis Hunt, and David Probert. Quality practices and problems in free software projects. In Marco Scotto and Giancarlo Succi, editors, *Proceedings of the First International Conference on Open Source Systems*, pages 24–28, Genova, Italy, 2005.
25. Jayadev Misra and William R. Cook. Computation orchestration: A basis for wide-area computing. *Jour. of Software and Systems Modeling*, 6(1):83–110, 2007.
26. Jan Peleska. Applied formal methods - from csp to executable hybrid specifications. In Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors, *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004, Revised Invited Papers*, volume 3525 of *Lecture Notes in Computer Science*, pages 293–320. Springer, 2005.
27. Nico Plat, Jan van Katwijk, and Hans Toetenel. Application and benefits of formal methods in software development. *Software Engineering Journal*, 7(5):335–346, 1992.
28. E. S. Raymond. *The Cathedral and the Bazar.* O'Reilly and Associates, 1999.
29. Gartner Report. The state of open source - 2008. Technical report, 2008.
30. Nuno F. Rodrigues and Luís S. Barbosa. Coordinspector a tool for extracting coordination data from legacy code. In *SCAM '08: Proc. Eighth IEEE International Working Conference on Source Code Analysis and Manipulation.* IEEE Computer Society, 2008.
31. Nuno F. Rodrigues and Luís S. Barbosa. On the discovery of business processes orchestration patterns. In *2008 IEEE Congress on Services*, pages 391–398, Washington, DC, USA, July 2008. IEEE Computer Society, IEEE Computer Society Press.
32. V. Rubanov. Automatic analysis of applications for portability across linux distributions. In Luis Barbosa, Antonio Cerone, and Siraj A. Shaikh, editors, *3nd International Workshop on Foundations and Techniques for Open Source Certification (OpenCert 2009), Electronic Communications of the EASST*, volume 20, 2009.
33. John Rushby. What use is verified software? In *12th IEEE International Conference on the Engineering of Complex Computer Systems (ICECCS)*, pages 270–276, 2007.

34. S. Sankar and R. Hayes. ADL - an interface definition language for specifying and testing software. In K. Suzuki, T. Higashino, A. Ulrich, and T. Hasegawa, editors, *Proceedings of the workshop on Interface Definition Languages*, pages 13–21. Portland, Oregon, US, 1994.
35. Siraj A. Shaikh and Antonio Cerone. Towards a quality model for open source software. In Bernhard Aichernig and Luis S. Barbosa, editors, *First International Workshop on Foundations and Techniques for Open Source Certification (OpenCert 2007)*. UNU-IIST, Macau, 2007.
36. Siraj A. Shaikh and Antonio Cerone. Towards a metrics for open source software quality. In Luis Barbosa, Antonio Cerone, and Siraj A. Shaikh, editors, *3nd International Workshop on Foundations and Techniques for Open Source Certification (OpenCert 2009), Electronic Communications of the EASST*, volume 20, 2009.
37. F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
38. Arie van Deursen and Leon Moonen. An empirical study into cobol type inferencing. *Science of Computer Programming*, 40(2–3):189–211, July 2001.
39. http://www.opencores.org/.
40. http://kernel.org/.
41. http://gcc.gnu.org/.
42. http://linuxfoundation.org/navigator/.
43. http://linuxtesting.org/.