View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidade do Minho: RepositoriUM

Towards rigorous analysis of Open Source Software

Luis S. Barbosa [1]

Departamento de Informática (HASLab)
Universidade do Minho
Braga, Portugal

Pedro R. Henriques [2]

Departamento de Informática (CCTC)
Universidade do Minho
Braga, Portugal

Alejandro Sanchez [3]

Departamento de Informática
Universidad Nacional de San Luis
San Luis, Argentina

Abstract

This paper discusses the (often hidden) potential of Open Source Software development to resort to, benefit from and cross-fertilize formal engineering methods, whose role is indisputable in the production of trustworthy software components. A strategy addressing the incorporation of formal verification methods in the Open Source Software lifecycle, in a somewhat less conventional way — that of assisting the re-engineering process of running code — is proposed.

Key words: Open Source Software, formal methods, program analysis.

[1] Email: `lsb@di.uminho.pt`
[2] Email: `pedrorangelhenriques@gmail.com`
[3] Email: `asanchez@unsl.edu.ar`

1 Introduction

The impact of Open Source Software on the way software applications and software-based services are currently developed, distributed and deployed, is indisputable. Usually acknowledged key benefits include rapid code turnover, extensive testing, supported maintenance and low development costs. Linux distributions, Apache and MYSQL, serve as paradigmatic examples of its success and resilience.

Open Source Software is being increasingly adopted by industry, also for mission and safety-critical applications. In general, experience has shown that many open source software products are reliable and have achieved adequate functionality and scalability. For example, an extensive study carried on a few years ago showed that an active, mature open source initiative may have fewer defects than similar commercial projects[4]. Similarly, reference [Aea02] reports on a study of 100 open source applications concluding that structural code quality was higher than expected and comparable with commercially developed software.

This does not mean that Open Source Software is immune to the sort of correctness problems and vulnerabilities affecting software in general. Software development *de facto* standards are still pre-scienctific in their lack of sound mathematical foundations to provide an effective basis to predict and certify programs behaviour. Open source communities are no exception, even if failure is definitely not advertised:

[4] The study, *How open source and commercial software compare: A quantitative analysis of TCP/IP implementations in commercial software and in the Linux kernel*, 2003, is available from `www.reasoning.com/downloads/opensource.html`.

> We tend not to hear very much about the failures. Only successful projects attract attention, and there are so many free software projects in total that even though only a small percentage succeed, the result is still a lot of visible projects. We also don't hear about the failures because failure is not an event. There is no single moment when a project ceases to be viable; people just sort of drift away and stop working on it. There is not even a clear definition of when a project is expired. Is it when it hasn't been actively worked on for six months? When its user base stops growing, without having exceeded the developer base? What if the developers of one project abandon it because they realized they were duplicating the work of another–and what if they join that other project, then expand it to include much of their earlier effort? Did the first project end, or just change homes? Because of such complexities, it's impossible to put a precise number on the failure rate. But anecdotal evidence from over a decade in open source, some casting around on SourceForge.net, and a little Googling all point to the same conclusion: the rate is extremely high, probably on the order of 90 to 95%.
>
> K. Fogel, in [Fog05]

Certifying software with respect to precise specifications of their behaviour and/or given levels of performance and security, constitutes the overall agenda of the so-called *formal* methods. Qualifier *formal* stresses that such a certification is not a matter of opinion (i.e., a legal argument), but has a similar status to that of a mathematical proof, in the sense that precise mathematical techniques are used either to build and compose the software, or to guide a systematic verification procedure. Formal methods in Software Engineering is no longer an esoteric issue, but essential to obtaining the highest degrees of assurance required by trustworthy systems. And industry is becoming more and more aware of this fact. On the other hand, the maturity of current tools to support formal development, analysis and verification is now much more adequate for industrial use than it has been in the past, when it was extremely hard for non-specialists to use such methods.

Open Source Software, however, by the very nature of its open and unconventional development model, in which coding and debugging efforts are shared among a distributed, heterogeneous community, with decentralized control mechanisms, makes software quality assessment, let alone full certification, particularly hard to achieve. On the other hand code is exposed, freely available, often complemented with heavy volumes of informal documentation (in the form of source code comments, wiki notes, forum threads,

...), offering an enormous potential for verification and analysis.

The certification problem for Open Source Software raises specific challenges and opportunities, both from the technical/methodological and the managerial points of view (see, *e.g.*, [DAI09] for an extensive review on the security dimension). Not by chance the discussion on how formal development methods can be brought to Open Source Software practice, has been the focus of a series of workshops promoted by the United Nations University, with the acronym OpenCert since 2007 (see `opencert.iist.unu.edu/` and [BCS10] for the latest proceedings).

This paper aims at contributing to this debate: to what extent, and in which ways, may research in formal methods and accompanying tools become meaningful and usable for open source development and certification? There is certainly no single answer. In the sequel we argue for a lightweight , 'backward' approach: rather than insisting on the effective introduction of formal methods in the development process, we suggest the dissemination of rigorous program understanding and analysis techniques suitably integrated in an open infrastructure where open source code can be registered and analyzed in a number of different ways.

Paper outline.

Section 2 discusses the dichtonomy *Formal Methods* vs *Open Source Software*, pointing out which characteristics of Open Source Software one may build on to introduce such methods without disturbing its peculiar, but successful development cycle. Then, section 3 describes our proposal of a certification infrastructure for Open Source Software. Sections 4 and 5 make such a proposal more concrete through a brief summary of two tools developed witin the authors' research team to be part of the envisaged infrastructure. Finally, section 6 concludes and gives some pointers for future research.

2 Quality, Formal Methods and Open Source Software

A standard approach to reduce risks in using an artifact is to establish an independent certification process. However, no certification standards exists that could be used to assess or classify the quality of Open Source Software. Such, a standard would certainly have to include the maturity of the development process, but open source reality has long been ignored in academia and its study largely reduced to a social phenomenon. Empirical studies exist (see e.g. [Aea02,MFH02,MHP05]) but are still insufficient. Moreover they tend to focus on the *context* of software production, i.e, on the factors that determine the development conditions and, thus, are expected to influence its final quality, rather than on the *product* itself. Technical, or *product oriented* quality, on the other hand, deals with factors directly influencing maintain-

ability, reliability and portability, which are extremely relevant for industry integrating Open Source Software in their own solutions.

That is precisely the focus of formal or rigorous engineering methods. Despite the complex, decentralized nature of Open Source Software development process, it is possible to identify a number of its characteristics which, in our opinion, favor a fruitful interaction with such methods. Our claim is that any proposal for incorporating this sort of methods in Open Source Software development should build on the following:

- *High code modularity*, leading to and stimulating separate development, without a need to change or understand the core system, or interfere with each developer's progress. This not only reduces the risk of propagating defects, but is also the key for a successful introduction of tight control cycles based on rigorous methods, which are, in their majority, compositional. A popular study of the Linux kernel development [LC03] concluded that modularity let multiple developers work on the same solution, often in competition, increasing the probability of timely, high-quality solutions.
- *Rapid release cycles* which keep code reviewers and developers interested and motivated, quickly resulting in systematic and high quality extensions. This also makes possible similarly rapid verification or analysis cycles and the suitable feedback of their results into the development process.
- *Independent and active* code review, typically lead by people outside the project team. A publicly visible bug and issue tracking tool is used by nearly all big open source projects. Users post bugs and enhancement requests. Each such post becomes, in effect, a tiny public mailing list focused solely on that issue. The introduction of formal analysis methods, simply adds to this already present critic ability.
- *Large, sustainable communities* to develop, test and debug code effectively. An investigation of open source projects evolution cited in [Abe07] found that a large base of voluntary contributing members was one of the most important success factors. Rigorous analysis methods, timely applied, help the coevolution of a product and its community, and reinforce positive feedback as well as the reward- and-recognition culture which facilitates internal cohesion.
- *Traceable pedigree*. Unlike closed software, where the identity of the real supplier is often hidden, the lineage of a open source product can easily be traced: it is easier to determine exactly who did what, and who has modification rights. This provides a sound basis on top of which composition mismatches and errors detected during analysis can be traced to their origins and easily corrected.
- *Tool-mediated communication* is extensively used. Actually, a ubiquitous trait of open source development is that tool mediation is the norm. This

enables leaders to shift the burden of policy enforcement from people to tools, which support authentication, regulation of commit privileges, audit and notification. Again, the introduction of analysis and vertification, corrective steps in the development cycle can easily benefit from this tool-mediated communication.

- Last but not the least, and contrary to a widespread belief, Open Source Software development, being distributed and multi-centered is far from being anarchic. Typically, composition, configuration, and information flow in and out of the project's server is somehow (but effectively) controlled. Project initiators and main contributors often exercise tight control over the engineering practices, not by limiting the developers behavior in their own personal space, but by limiting the kinds of transactions developers can make upon the persistent project state on the server. This may provide the needed infrastructure for enforcing quality checks based on formal technics.

If formal methods offer a valuable contribution to assess and promote Open Source Software reliability, an almost reverse claim can also be made: the relevance of Open Source Software for the formal methods community cannot be underestimated. Actually, open source licenses, allowing others to study, use, improve, and release new versions, are an essential ingredient to promote and disseminate tools supporting formal development methods — a first-class vehicle for making continued research possible. Sadly, many such tools have completely disappeared because they were not released under open source licenses. The absence of an open source license for ESC/Java, for example, created a difficult situation for many people and companies depending on this popular tool, once Compaq/HP decided to abandon its maintenance.

A key benefit of using Open Source Software is that the code can be compiled freely. As technology advances fast the ability to recompile the source code becomes more and more important. Although a general remark, this applies indeed to support tools for formal methods: Open Source Software remains the key. Similar remarks apply to their long term survival.

Another argument (made mostly in the context of mathematical proofs) stems from the scientific validity and acceptance of computer generated, or computer assisted proofs. For such proofs to be included as standard material, the software system used to arrive at the result must also be available to researchers, e.g., to independently check the proof for its correctness.

3 The reverse perspective

The considerations above lead the authors' current research towards addressing the incorporation of formal verification methods in the Open Source Software lifecycle in a very peculiar perspective: that of assisting the re-engineering process of running code.

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 are discussed elsewhere [BCPS10]. Our starting point here is the fact that, faced with a high risk dependence on open source components, often to be embedded on their own software, industry is more and more prepared to spend resources to increase confidence in (the level of understanding of) their code. From this point of view, the same principles and calculi used for (formal) program development can be applied in the reverse direction, from concrete to abstract models, for understanding and documenting implementations. More precisely, we seek

- Developing *program understanding and analysis techniques* and combine them for quality assessment of open source code. As Open Source Software offers full access to source code this enables the effective application of approaches and tools entirely targeting code analysis. The nature of Open Source Software entails the need for integration of techniques spanning the "micro" to the "macro" levels (e.g., from 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). Sections 4 and 5 details two such techniques developed in this context.
- Catering for their smooth integration into the peculiar development process of Open Source Software without disturbing its collaborative, distributed and heterogeneous character. This amounts to establish feedback loops in open source development, making publicly available a number of interrelated analysis tools, to enhance the overall software reliability.

Our proposal to achieve the latter objective is through an *online, open infrastructure* in which independently developed analysis tools (with different levels of sophistication) are inserted to monitor, assess and, at a later stage, certify open source products. Ideally, such an infrastructure would allow for the registration of open source projects, their source code visualization 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. It will 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, one may expect that feedback loops will have an effective impact in the overall quality of Open Source Software products, with none or minimal intrusion on their life-cycle.

Such a certification infrastructure, currently under development at Universidade do Minho, adopts an open architecture, in the sense that new analysis or visualization components can be easily added relying on an open, general format for data/code representation. In the very spirit of Open Source Soft-

ware, this will allow separate use and distribution of the framework, such that third parties can use it and plug-in their own analysis and visualization components.

The following two sections outline two of such plug-in tools — Gamma and CoordPat already developed in this project for source code analysis.

4 Gamma: A plug-in for assertion-based slicing

The Gamma toolkit [dCHP10,BdCHP10] is an *assertion-based slicer* equipped with a *verification* component, to generate verification conditions, and a program visualization functionality. Its purpose is to extract slices from code through a number of different families of slicing algorithms (precondition, postcondition, and contract-based).

This plug-in is intended to operate over source code suitably annotated with *contracts* in the sense of the *design by contract* paradigm — an approach that advocates specifying the behavior of program routines through the use of annotations, and checking them individually, either statically or dynamically, to obtain globally correct programs.

Of course this may sound strange with respect to its main application target – Open Source Software. Actually, even if annotated Open Source code may soon emerge as part of a code documentation effort whose need the community is increasingly aware of, such is clearly not dominant today. However, recent advances in automatic inference of annotations provide other tools which act as pre-processors for Gamma. For example, a component of Frama-C [CCPS09], a popular open source framework based on static analysis, automatically infers the preconditions for a given procedure.

Gamma, whose implementation includes a new, very efficient slicing algorithm [BdCHP10], is not only useful for code analysis, but also to assist automatic code adaptation. This may involve elimination of code redundancy, but can also go much further. For example, suppose there is a library containing a procedure that implements a traversal of some data structure, and collects a substantial amount of information in that traversal. Now suppose this library is to be reused dropping the requirement that all the information collected in the traversal should be used. In this case the procedure respects a weaker specification, and thus it makes sense to produce a specialized, corresponding version of the library. This is crucial for software reuse, and open source development heavily depends on reuse.

Specializations of assertion-based slicing, for example to focus exclusively on post-condition annotations, may be used to study when a property is valid in a specific section of a program (for example inside a critical region to be executed on a specific thread) and false elsewhere. Therefore, it may be used to study the correct behaviour of code with respect to that section. Similarly, the

property may correspond to some invariant on a data structure, which ought to be maintained. The toolkit also generates, in a step-by-step fashion, a set of verification conditions given as input to automatic SMT provers, which allows to establish the initial correctness of the code with respect to their contracts.

5 CoordPat: A plug-in for architectural analysis

If Gamma addresses code *micro* level, i.e., the level of procedures and statements, CoordPat is oriented toward code analysis *in the large*. Basically, it may be regarded as a tool for reverse architectural analysis, providing a systematic way to encode and identify coordination patterns in source code. It aims at uncovering, registering and classifying architectural decisions often left undocumented and hardwired in the application code. Moreover, through the systematic, tool-supported discovery of architectural decisions, it is expected to entail the reconstruction of the corresponding specifications.

Actually, current software systems 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, at the source code level, with the application. Therefore, its precise identification becomes a major methodological (and technical) problem which cannot be overestimated and to which this tool aims at contributing. Not seldom open source applications 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 contexts, to control architectural drifts, i.e., the accumulation of architectural inconsistencies resulting from successive code modifications.

CoordPat implements a rigorous methodology [RB10,RB08] to extract, from source code, its *coordination layer*, i.e. the architectural layer which captures system's behaviour with respect to its network of interactions. The qualifier is borrowed from research on *coordination* models and languages [GC92], which emerged in the nineties to exploit the full potential of parallel systems, concurrency and cooperation of heterogeneous, loosely-coupled components.

The extraction methodology combines suitable slicing techniques to build a family of *dependence graphs* by pruning a *system dependence graph* [HRB88] first derived from source code. After the extraction stage, the tool exploits such graphs to identify and combine instances of *coordination patterns* and then reconstruct the original specification of the system's coordination layer. The word *pattern* is used here with the usual meaning: a way to describe and reuse standard solutions for recurrent problems. Thus, CoordPat maintains an incrementally-built repository of patterns used to guide the analysis

process.

Coordination patterns are described in a formal, graph-based language for which a relational semantics was introduced in [ORHB10]. A pattern repository is integrated in the tool and dynamically populated by the users. The tool also provides features for (*i*) basic editing of coordination patterns, (*ii*) their syntactic and semantic validation, (*iii*) graph rendering for their visualisation (see e.g. Fig. 1) and (*iv*) pattern discovery in a dependence graph previously extracted.

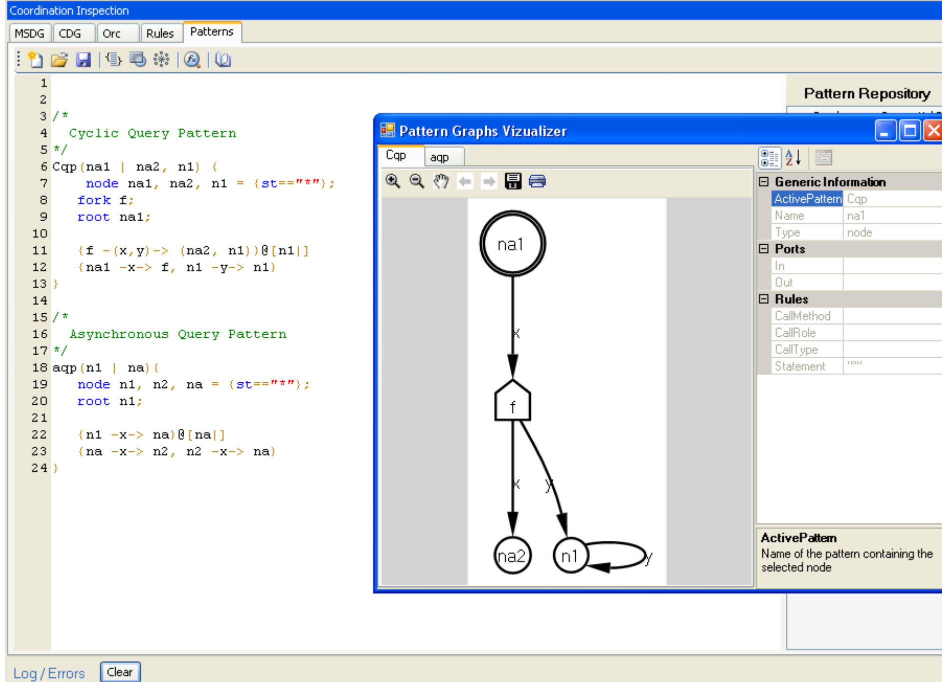

Fig. 1. The *Cyclic Query Pattern* and its graphical representation

6 Conclusions and future work

Open Source Software is software whose license gives users the freedom to run it for any purpose, to study and modify, and to redistribute copies of either the original or the modified program, without having to pay royalties to previous developers. Companies are becoming aware that integrating Open Source Software into commercial products (made available by liberal open source licenses) reduce development costs while offering high-quality, extensively tested components. Furthermore, governments are getting worried with the growing dependence on proprietary formats and software in their administration, and regard Open Source Software as a warranty of technological independence. This turns out to a strategic advantage, mainly in the developing world.

Strengthening the role of Open Source Software in the global IT sector is, therefore, a strategic aim and, so we believe, a condition for increased, democratic citizenship in our information-led societies. However Open Source Software quality can be very hard to measure and to compare [Spi11]. This could be substantially improved if there were appropriate standards, supported by analysis tools, for certifying such software. Developing such tools, making them widely available for the open source community, and, in the long term, contributing to the creation of an international certification authority for open source software, is the path to which we would like to contribute.

This paper summarizes current research at Minho University, Portugal, on a possible strategy leading to the establishment of an independent certification process, with potential for a long-term impact on the integration of trustworthy, open source components, in large, complex systems. In short, we made a case for formal methods use in the 'reverse' direction, i.e., to guide code based analysis of open source components with potential impact in their improvement and reuse.

As related work, ALITHEIA CORE [GS09] must be cited. This is an extensible platform designed specifically for performing large-scale software quality evaluation through the extraction and combination of a number of metrics on open source projects, resorting both to white-box test and code analysis. A central issue in this project is scalability to huge volumes of data, which entails the need for complex mirroring schemes and multicore execution. Several other projects exist proposing solutions for Open Source Software testing and evaluation. For example, QSOS (`www.qsos.org/`) is a methodology to assess, select and compare, open source components in an objective, traceable way. Project OSSTMM `www.isecom.org/osstmm/` developed a peer- reviewed methodology for performing security tests on Open Source Software. What distinguishes our own proposal is the explicit aim of incorporating formal methods in addressing Open Source Software certification.

But, of course, a lot of questions remain to be answered. To mention just one we have not addressed so far: *security*. Security requires a specific analysis, since open source development does not usually follow the best security practices. As [DAI09] notices, in a recent book on security certification of Open Source Software, *the lower number of security events involving Open Source Software may be ascribed to its smaller market share rather than to its robustness*. Tools for security analysis must definitively be pluged-in into the certification infrastructure suggested above.

Another main issue, requiring further experimental research, is the study of the potential impact of such an infrastructure in the concrete open source communities to which it is directed. In any case such an integration, or synergy, needs to be *non disturbing* of the community principles and (best) practices.

Acknowledgements. This research was partially supported by the Cross project, under contract `PTDC/EIA-CCO/108995/2008` with Fct, the Portuguese Foundation for Science and Technology. Several ideas discussed in this paper and pursued in the Cross project benefited from discussions with Antonio Cerone, Bernhard Aichernig and Siraj Shaikh on possible roles for formal methods in Open Source Software certification, namely in the context of the OpenCert workshops. Collaboration with Daniela da Cruz, Jorge Sousa Pinto and José Barros in the development of the Gamma toolkit, as well as with Nuno Oliveira and Nuno Rodrigues in the design of CoordPat, is greatly acknowledged.

References

[Abe07] M. Aberdour. Achieving quality in open source software. *IEEE Software*, pages 58–64, 2007.

[Aea02] L. Angelis and et al. Code quality analysis in open source software development. *Information Systems Journ.*, pages 43–60, 2002.

[BCPS10] L. S. Barbosa, A. Cerone, A. K. Petrenko, and S. A. Shaikh. Certification of open-source software: A role for formal methods? *International Journal of Computer Systems Science and Engineering*, (4):273–281, 2010.

[BCS10] L. S. Barbosa, A. Cerone, and S. Shaikh, editors. *Foundations and Techniques for Open Source Software Certification, Proc. OpenCert 2010, Pisa, September, 2009*. Electronic Communications of the EASST, volume 33, 2010.

[BdCHP10] J. Bernardo Barros, Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Assertion-based slicing and slice graphs. In *SEFM'10 — 8th IEEE International Conference on Software Engineering and Formal Methods*, pages 93–102. IEEE Computer Society, Conference Publishing Services (CPS), Sept 2010.

[CCPS09] L. Correnson, P. Cuoq, A. Puccetti, and J. Signoles. Frama-C User Manual. `http://frama-c.cea.fr/download/user-manual-Beryllium-20090902.pdf.`, November 2009.

[DAI09] Ernesto Damiani, Claudio Agostino Ardagna, and Nabil El Ioini. *Open Source Systems Security Certification*. Springer, 2009.

[dCHP10] Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Gamaslicer: an online laboratory for program verification and analysis.

In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications - LDTA '10*, pages 3:1–3:8. ACM, 2010.

[Fog05] Karl Fogel. *Producing open source software - how to run a successful free software project.* O'Reilly, 2005.

[GC92] D. Gelernter and N. Carrier. Coordination languages and their significance. *Communication of the ACM*, 2(35):97–107, February 1992.

[GS09] G. Gousios and D. Spinellis. Alitheia core: An extensible software quality monitoring platform. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 579–582. IEEE, 2009.

[HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conf. on Programming Usage, Design and Implementation*, pages 35–46. ACM Press, 1988.

[LC03] Gwendolyn K. Lee and Robert E. Cole. From a firm-based to a community-based model of knowledge creation: The case of the linux kernel development. *Organization Science*, 14:633–649, November 2003.

[MFH02] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, July 2002.

[MHP05] 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.

[ORHB10] Nuno Oliveira, Nuno Rodrigues, Pedro Rangel Henriques, and Lus Soares Barbosa. A pattern language for architectural analysis. In *SBLP 2010 14th Brazilian Symposium in Programming Languages*, volume 2, pages 167–180. SBC — Brazilian Computer Society (ISSN: 2175-5922), 2010.

[RB08] N. F. Rodrigues and L. S. Barbosa. Coordinspector: a tool for extracting coordination data from legacy code. In *Proc. IEEE 8th Inter. Working Conference on Source Code Analysis and Manipulation (SCAM'08), Beijing, 2008*. IEEE Computer Society, 2008.

[RB10] N. F. Rodrigues and L. S. Barbosa. Slicing for architectural analysis. *Sci. Comput. Program.*, 75(10):828–847, 2010.

[Spi11] D. Spinellis. Choosing and using open source components. *IEEE Software*, 28(3):96, 2011.