The United Nations University

UNU-IIST

International Institute for
Software Technology

Pre-proceedings of the 4th International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2010)

Pisa, Italy, 17-18th September 2010

Satellite Event of the 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2010)

Luís S. Barbosa, Antonio Cerone and Siraj A. Shaikh (Editors)

August 23, 2010

UNU-IIST Report No. 439 $\boxed{\mathcal{R}}$

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research $\boxed{\text{R}}$, Technical $\boxed{\text{T}}$, Compendia $\boxed{\text{C}}$ or Administrative $\boxed{\text{A}}$. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST's home page: http://www.iist.unu.edu, if you would like to know more about UNU-IIST and its report series.

Peter Haddawy, Director

The United Nations
University

UNU-IIST

International Institute for
Software Technology

P.O. Box 3058
Macao

Pre-proceedings of the 4th International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2010)

Pisa, Italy, 17-18th September 2010

Satellite Event of the 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2010)

Luís S. Barbosa, Antonio Cerone and Siraj A. Shaikh (Editors)

Abstract

This volume includes abstracts of the two invited contributions and full papers of the eleven peer-reviewed contributions presented at OpenCert 2010.

Luís S. Barbosa is Associate Professor at the Department of Informatics of Minho University and Director of MAP-i, the joint doctoral programme in Computer Science of the three top universities in North Portugal (Minho, Aveiro and Porto). Currently his research interests include formal models for software components, services and architectures; architectural understanding and re-engineering of legacy software; and coalgebraic semantics. In these areas he coordinated 3 research projects, bilateral partnerships with Brasil and China, and served as the portuguese coordinator for the LerNet ALFA network. For the last 4 years he has published 4 book chapters, 10 journal papers and more than 25 international conference papers on the above mentioned topics, and served on the PC of several international conferences.

Antonio Cerone is Research Fellow at the International Institute for Software Technology (IIST) of the United Nations University (UNU). His research focuses on education, e-learning and open source software methodologies and certification, as well as on formal methods and their application to interactive systems, system biology, ecosystems, information security, asynchronous hardware, concurrent and real-time systems, and safety-critical systems. Antonio is the leader of the UNU-IIST Programme on "Strategies and Technologies for Education" and coordinates the UNU-IIST Postgraduate Programme. He is the Chair of the Steering Committee of the IEEE International Conference on Software Engineering and Formal Methods (SEFM). He is a member of Formal Methods Europe (FME) and the European Association for Software Science and Technology (EASST). He is also Editorial Board Member of Electronic Communications of the EASST (ECEASST)

Siraj A. Shaikh is a Senior Lecturer at the Department of Computing and the Digital Environment, part of the Faculty of Engineering and Computing at Coventry University, UK. His research interests include information and network security with a focus on identity management and communication networks. His doctoral and post-doctoral research involved design and verification of security systems, alongside interactive systems and open source software certification. He is a Chartered Member of the British Computer Society (BCS) and the Secretary for the International Federation for Information Processing (IFIP) Working Group 11.4 on Network Security. Previously, he has worked as a Research Fellow at Cranfield University, UK, and a Postdoctoral Research Fellow at the International Institute for Software Technology of the United Nations University (UNU-IIST), Macau SAR China.

Preface

Free/Libre/Open Source Software (FLOSS) has proved to be very successful over the years. This has brought with it an increasing interest to use FLOSS for complex and industrial-strength applications. An important aspect of this is the formal and rigorous analysis of the quality and safety properties of FLOSS along with other related challenges.

This volume contains the preproceedings of the 4th International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2010) held as a satellite event of the 8th IEEE International conference on Software Engineering and Formal Methods (SEFM 2010), at Pisa, Italy on 17th-18th September 2010. The aim of this workshop is to bring together researchers from the academia and the industry interested in the quality assessment of FLOSS and ultimately provide a platform for the establishment of a coherent certification process for FLOSS.

The preproceedings include a total of thirteen papers, two of which are invited contributions by Mario Fusani of CNR-ISTI and Panagiotis Katsaros of Aristotle University of Thessaloniki. The eleven other papers are submitted contributions and have each been reviewed by at least three programme committee members.

The organisers would like to express their gratitude to all the contributors, the programme committee and all the referees for their hard work and support. The result in your hand would not have been possible without their effort and commitment.

Luís Soares Barbosa
Minho University
Braga, Portugal

Antonio Cerone
United Nations University
Macau SAR China

Siraj Ahmed Shaikh
Coventry University
Coventry, United Kingdom

Conference Organisation

Programme Chairs

Luís Soares Barbosa
Antonio Cerone
Siraj Ahmed Shaikh

Programme Committee

Admir Abdurahmanovic
Bernhard Aichernig
Andrea Capiluppi
Gabriel Ciobanu
Ernesto Damiani
Jim Davies
Roberto Di Cosmo
Fabrizio Fabbrini
Maria João Frade
Jesus Arias Fisteus
Dan Ghica
Tomasz Janowski
Paddy Krishnan
Paolo Milazzo
Josè Miranda
John Noll
Alexander K. Petrenko
Simon Pickin
Sulayman K. Sowe
Ralf Treinen
Joost Visser
David von Oheimb
Tanja Vos
Anthony Wasserman

External Reviewers

Daniela da Cruz
José Pedro Correia
Miguel Alexandre Ferreira
Victor Kuliamin
Francisco Moura
Giovanni Pardini
Alberto Proença
Nuno Rodrigues

Table of Contents

Component certification as a prerequisite for widespread OSS reuse

[1]Panagiotis Katsaros [1]Ioannis Stamelos

[1]Aristotle University of Thessaloniki
54124 Thessaloniki, Greece
{katsaros,stamelos}@csd.auth.gr

Abstract: Open source software is the product of a community process that in a single project may employ different development techniques and volunteers with diverse skills, interests and hardware. Reuse of OSS software in systems that will have to guarantee certain product properties is still complicated. The main reason is the many different levels of trust that can be placed on the various OSS sources and the lack of information for the impact that a reused OSS component can have on the system properties. A prerequisite for promoting widespread reuse of OSS software is certification at the component level in an affordable cost. This talk will address the main technical issues behind OSS component certification by formal and semi-formal techniques, as well as the business incentives that raised the need for the OPEN-SME European funded project. OPEN-SME aims to provide an OSS software reuse service for SMEs, in order to address the problem that OSS evolves by volunteers that usually cannot elaborate an effective component-based development process. We briefly mention how OSS is partitioned into software components and then we show how the provision of verifiable certificates can provide assurance that an OSS component conforms to one or more anticipated requirements, necessary for reusing it in a system.

Keywords: component certification, open source software reuse, component-based development

Security Evaluation and Hardening of Free and Open Source Software (FOSS)

Robert Charpentier[1], **Mourad Debbabi**[2], **and TFOSS Research Team**[2*]

[1] Defence Research and Development Canada, Valcartier, Quebec, Canada
[2] Computer Security Laboratory, Concordia University, Montreal, Quebec, Canada

Abstract: Recently, Free and Open Source Software (FOSS) has emerged as an alternative to Commercial-Off-The-Shelf (COTS) software. Now, FOSS is perceived as a viable long-term solution that deserves careful consideration because of its potential for significant cost savings, improved reliability, and numerous advantages over proprietary software. However, the secure integration of FOSS in IT infrastructures is very challenging and demanding. Methodologies and technical policies must be adapted to reliably compose large FOSS-based software systems. A DRDC Valcartier-Concordia University feasibility study completed in March 2004 concluded that the most promising approach for securing FOSS is to combine advanced design patterns and Aspect-Oriented Programming (AOP). Following the recommendations of this study a three years project have been conducted as a collaboration between Concordia University, DRDC Valcartier, and Bell Canada. This paper aims at presenting the main contributions of this project. It consists of a practical framework with the underlying solid semantic foundations for the security evaluation and hardening of FOSS.

Keywords: Free and Open Source Software, Security Hardening, Static Analysis, Dynamic Analysis, Aspect Oriented Programming.

1 Introduction

During the past two decades, the software market has been dominated by Commercial-Off-The-Shelf (COTS) products that offer a myriad of functionalities at reasonable prices. However, the intrinsic limitations of COTS software such as security weaknesses, closed source code, expensive upgrades, and lock-in effect have emerged over time. This led to the development of a parallel “economy" based on Free and Open Source Software (FOSS). The latter refers to software whose source code is made available for use and modification without the expensive license fees imposed by COTS software vendors. FOSS is developed either by volunteers, non-profit organizations, or by large computer firms who want to include “commodity" software to give a competitive advantage to their hardware products. To date, thousands of FOSS projects are carried out via Internet collaboration. A plethora of high-quality applications are available for use or modification at no (or small) cost. Many of these FOSS products are widely available and are considered to be as mature as their COTS equivalents. FOSS is now perceived as a viable

[*] The TFOSS research team is comprised of: D. Alhadidi, M. Azzam, N. Belblidia, A. Boukhtouta, A. Hanna, R. Hadjidj, H. I. Kaitouni, M. A. Laverdière, H.Z. Ling, S. Tlili, X. Yang, and Z. Yang.

long-term solution that deserves careful consideration because of its potential for significant cost savings, improved reliability, and support advantages over proprietary software [CC04].

Technically, the secure integration of FOSS in IT infrastructures is very challenging and demanding. Methodologies and technical policies must be adapted to reliably compose large FOSS-based software systems [Bol03]. This requirement is exacerbated by the fact that our dependency on software will continue to grow in the next decade. Recent studies confirm that the level of reliability and security currently offered by commercial products is clearly inadequate and that an order of magnitude increase is needed to cope properly with cyber threats [FR03]. A DRDC Valcartier (Defence R&D Canada Valcartier)-Concordia University feasibility study, completed in March 2004, addressed these issues and considered the technological options to cope with the security and reliability of complex information systems including FOSS and COTS software [CC04]. It concluded that the most promising approach is to combine advanced security design patterns and Aspect-Oriented Programming (AOP). This facilitates the separation of the definition and implementation of quality and functional specifications. Such a “separation of concerns" will ease the development of secure design patterns to be applied to a wide range of applications. Time and cost investments were also evaluated for the scientific demonstration of these concepts.

Following the recommendations of this study, a three-year project has been conducted as a collaboration between Concordia University, DRDC Valcartier, and Bell Canada. This paper aims at presenting the main contributions of this project. More precisely, it presents a practical framework with the underlying solid semantic foundations for the security evaluation and hardening of free and open source software. The evaluation aims to automatically detect vulnerabilities in FOSS that will be corrected by the systematic injection of security code thanks to dedicated aspect-oriented technologies. The security code is meant to be derived from security hardening patterns.

The remainder of this paper is organized as follows. Section 2 surveys the related work. In Section 3, we present our first contribution involving static analysis and model checking for detecting security vulnerabilities. Section 4 shows our contribution for security hardening, which is based on aspect-orientation. Finally, Section 5 concludes the paper.

2 Related Work

Security code analysis includes security code inspection, automatic analysis and static analysis techniques. Security code inspection techniques are borrowed from software engineering practices [Fag76] and adapted specifically for security purposes. Automatic analysis techniques generally scan the code looking for security sensitive coding patterns that are compiled in checklists. The available techniques are limited to vulnerable coding patterns such as buffer overflows, heap overflows, integer overflows, format string vulnerabilities, SQL injection, cross-site scripting and race conditions [Gre]. Among the tools that implement these techniques, we can cite: Flawfinder [Whe], Coverity [Cov] and PolySpace [Pol]. Static analysis is used to predict security properties of programs without resorting to their execution. Static analysis techniques include flow-based analysis [BDNN01], type-based analysis [CGG02] and abstract interpretation [CC77]. Finally, the evaluation by security testing is based on the design and execution of test cases in order to identify vulnerabilities in the security features of the software [Her03, WTS03].

For FOSS security hardening, four approaches could be distinguished: analyzing, monitoring, auditing, and rewriting [MM00]. Analysis-based techniques range from simple scanning of code in order to detect malicious code to sophisticated semantics-based analysis of programs. One popular form of analysis-based techniques is certified compilation, which leverages the information generated by the compiler in order to endow the code with a security certificate. This could take the form of proofs as in PCC [Nec97], structured annotations as in ECC [Koz98], or typing annotations with typed assembly languages TAL [MWCG99], STAL [MCGW98], DTAL [XH99], Alias Types [SWM00], HBAL [AC03] and Linearly Typed Assembly Language [CM03]. Nevertheless, static analysis is to some extent complex and in some regards undecidable. Monitoring is based on background daemons watching the execution of a program to prevent, at run-time, any harmful operation from taking place [HMS03]. The main drawback of monitoring is the overhead in terms of performance that is induced by the daemons. With auditing-based approaches, the system activity is recorded in an audit trail. This provides a sequence of events related to a trace of program execution and allows to track back any harmful action. If any malicious code causes damage, the audit trail allows to do the recovery and to take the necessary precautions for the future. As of the rewriting-based approach, the code is modified to prevent deviation from the security policies in place. A rewriting tool inserts extra code to perform dynamic checks that ensure that “bad things” cannot happen. Among the research contributions in rewriting-based security, we can cite [RW02].

In our project, we use aspect-orientation as an enabling technology that allows the systematic injection of security in FOSS. Aspect-Oriented Programming (AOP) [KLM$^+$97] promotes the principle of separation of concerns, thus allowing smooth integration of security hardening mechanisms inside existing software. The most prominent AOP languages are AspectJ [KHH$^+$01] and Hyper/J [TO00], which are built on top of Java programming language. A similar work has also been done to provide AOP frameworks for other languages. For instance, AspectC [CKFS01] is an aspect extension of C that is used to provide separation of concerns in operating systems. Similarly, AspectC++ [SGS02] and AspectC# [Kim02] are respectively AOP extensions of C++ and C# languages. Some attempts have been made to use AOP for security. For instance, Cigital Labs have conducted a DARPA-funded project [Lab03], where the AOP paradigm has been used to address software security. The main outcomes of this project are a security dedicated aspect extension of C called CSAW [Lab03] and a weaving tool. De Win [WPJV02] has explored the use of AspectJ to integrate security aspects within applications.

3 Static Analysis and Model-Checking for Vulnerability Detection

Our approach brings into a synergy static analysis and model-checking in order to leverage the advantages and overcome the shortcomings of both techniques. The core idea is to utilize static analysis for the automation and the optimization of program abstraction processes. Moreover, programmers take advantage of model-checking techniques to define a wide range of system-specific security properties. As a result, our approach can model-check large software against customized system-specific security properties. Our ultimate goal is to provide a security verification technique for open source software, thus we base our approach on GCC, which is usually a defacto open-source compiler. The language-independent and platform-independent GIMPLE representation [Nov03] of GCC facilitates static analysis by providing easy access to flow, type, and alias information. Being based on GIMPLE, our approach can be extended to support other languages such as C, C++, and Java. For the verification process, we use the Moped model-checker for pushdown systems [KSS]. The latter are known to efficiently model program execution and inter-procedural behavior. Moped has a procedural input language called Remopla to define programs as pushdown systems. As such, the program abstraction derived from the GIMPLE representation is serialized into Remopla representation. In addition, we enrich program abstractions with Remopla constructs that compute and capture data dependencies between program expressions. Therefore, we are able to detect insidious errors that involve variable aliasing and function parameter passing. Security properties and program Remopla model are input to Moped in order to detect security violations and provide witness paths leading to them.

Moped allows the verification of reachability properties by looking for the reachability of a specific statement in the Remopla code. Though interesting, this capability is not directly sufficient for verifying security properties. In fact, a security property is the description of a pathological behavior in the execution of a program. Such a behavior requires in general an elaborated formalism to be specified and can rarely be stated as the simple reachability of a specific statement in the program. To specify security properties, we use the formalism of *security automata*. A security automaton is a simple automaton with two spacial states: *start* and *error*, and transitions are mapped to instructions or statements in the program to verify. The reachability of the error state in the security automaton when synchronized with the program behaviors is an indication of the occurrence of the pathology. To overcome the limitation of Moped in this regard, we translate a security automaton into a Remopla representation then synchronize it with the Remopla model of the program in question. This comes to synchronizing the pushdown systems of the program and the security automaton. As such, the problem of verifying a security property is translated into detecting the reachability of the error state in the synchronized model.

3.1 Design and Implementation

Fig. 1 depicts the architecture of our security verification environment. The security verification of programs is carried out through different phases including security property specification, static pre-processing, program model extraction, and property model-checking. In the following paragraphs, we describe the input, the output, and the tasks of each of these phases.

- Phase1. Security Property Specification:
  - Input: Security properties.
  - Output: Remopla automata of security properties.

  The first step of our verification process requires the definition of security properties describing what not to do for the purpose of building secure code. We provide users with a tool in order to graphically characterize the security rules that a program should obey. Each property is specified as a finite state automaton where the nodes represent program states and the transitions match program actions. Final states of automata are risky states that should never be reached. To ease the property specification, our tool supports syntactical pattern matching for program expressions and program statements. The graphically defined properties are then serialized into the Remopla language of Moped model-checker.

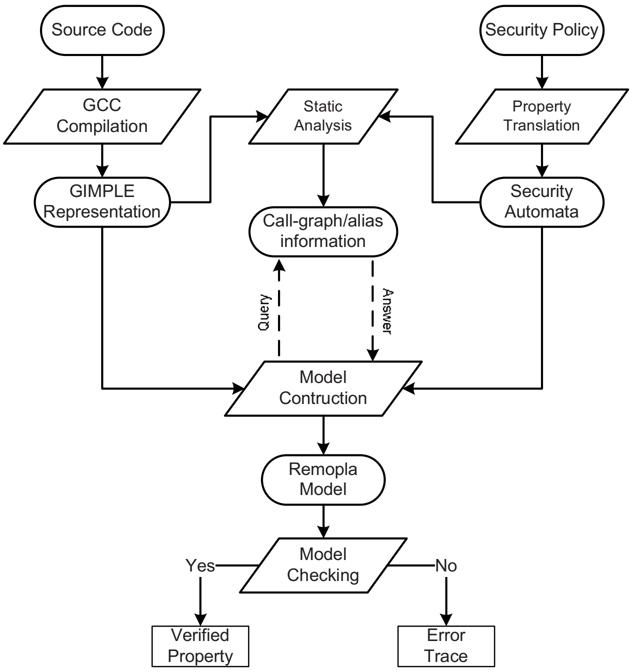


Figure 1: Security Verification Framework

- Phase2. Static Analysis for Pre-processing:
  - Input: Program GIMPLE representation and security properties.
  - Output: Call-graph and alias information.

Given a program and a set of security properties to verify, this process conducts call-graph analysis and alias analysis of the program. By considering the required properties, this phase identifies property-relevant behaviors of the analyzed program and discards those that are irrelevant. Besides, we resort to alias analysis in order to limit the number of tracked variables. We only consider variables that are explicitly used in security-relevant operations together with their aliases. All other variables are discarded from the verification process. The static pre-processing phase helps generating concise models that reduce the size of state spaces to explore.

- Phase3. Program Model Extraction:
  - Input: Program source code and specified security properties.

  - Output: Control-flow driven Remopla model or data-driven Remopla model.

  Both the program and the specified properties are translated into Remopla representation and then combined together. The combination of program models and security properties serves the purpose of synchronizing the program behaviors with the security automaton transitions. In other words, transitions in security automata are triggered when they match the current program statement. Our verification approach carries out program model extraction in two different modes: the control-flow driven mode and the data-driven mode. The control-flow mode preserves in the Remopla model the flow structure of the program, but discards data dependencies between program expressions. The resulting Remopla model is efficiently used to detect temporal security property violations and scales to large programs. On the other hand, our data-driven model captures data dependencies between program expressions. Hence, it enhances the precision of our analysis and reduces the number of false positives.

- Phase4. Program Model-Checking:

  - Input: Remopla model.
  - Output: Detected error traces.

  Model-checking is the ultimate step of our process. The generated Remopla model is given as input to the Moped model-checker for security verification. An error is reported when a security automaton specified in the model reaches a risky state. The original version of Moped has a shortcoming in a sense that it stops processing at the first encountered error. We have done a modification to Moped in order to be able to detect more than one error in a run. Moreover, we have developed an error trace generation functionality that maps error traces derived from the Remopla model to actual traces from the source code.

3.2 Results and Experiments

This section demonstrates the capability of our security verification framework in detecting real errors in large C software packages. We show that our approach can be efficiently used for uncovering undesirable vulnerabilities in source code. The CERT secure coding website [cer] provides a valuable source of information to learn the best practices of C, C++, and Java programming. It defines a standard that encompasses a set of rules and recommendations for building secure code. Rules must be followed to prevent security flaws that may be exploitable, whereas recommendations are guidelines that help improve the system security. The CERT standard also makes another difference between rules and recommendations stating that compliance of a code to rules can be verified whereas the compliance to recommendations is not always verifiable. To assist programmers with the verification of their code, we have integrated in our tool a set of secure coding rules defined in the CERT standard. As such, programmers can use our framework to evaluate the security of their code without the need to have high security expertise. CERT rules can mainly be classified into the following categories:

- *Deprecation rules*: These rules are related to the deprecation of legacy functions that are inherently vulnerable such as `gets` for user input, `tmpnam` for temporary file creation, and `rand` for random value generation. The presence of these functions in the code should be flagged as a vulnerability. For instance, CERT rule `MSC30-C` states the following "*Do not use the `rand()` function for generating pseudorandom numbers*".

- *Temporal rules*: These rules are related to a sequence of program actions that appear in source code. For instance, the rule `MEM3-C` from the CERT entails to *"Free dynamically allocated memory exactly once*". Consecutive free operations on a given memory location represents a security violation. Intuitively, these kind of rules are modeled as finite state automata where state transitions correspond to program actions. The final state of an automaton is the risky state that should never be reached.

- *Type-based rules*: These rules are related to the typing information of program expressions. For instance, the rule `EXP39-C` from the CERT states the following *"Do not access a variable through a pointer of an incompatible type*". A type-based analysis can be used to track violations of these kind of rules.

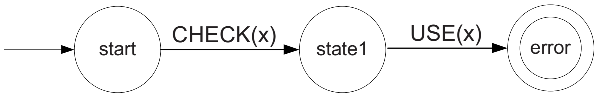
- *Structural rules*: These rules are related to the structure of source code such as variable declarations, function inlining, macro invocation, etc. For instance, rule `DCL32-C` entails to *"Guarantee that mutually visible identifiers are unique*". For instance, the first characters in variable identifiers should be different to prevent confusion and facilitates the code maintenance.

Our approach covers the first two categories of coding rules that we can formally model as finite state automata. In fact, we cover 31 rules out of 97 rules in the CERT standard. We also cover 21 recommendations that can be verified according to CERT. We conduct experiments that consist in detecting the defined set of CERT coding rules against a set of well-known and widely used open-source software. We strive to cover different kinds of security coding errors that skilled programmers can inadvertently produce in their code. The experiments are conducted in the two modes of our security verification tool: the control-flow mode that discards data dependencies and the data-driven mode that establishes data dependencies between program variables. To illustrate, Fig. 2 gives an example of a security automaton that captures the race condition errors. This security automaton can be used to check the compliance of source code to the following CERT rules:

- `POS35-C`: "Avoid race conditions while checking for the existence of a symbolic link".

- `FIO01-C`: "Be careful using functions that use file names for identification".

The Time-Of-Check-To-Time-Of-Use vulnerabilities (TOCTTOU) in file accesses are a classical form of race conditions. In fact, there is a time gap between the file permission check and the actual access to the file that can be maliciously exploited to redirect the access operation to another file. The automaton in Fig. 2 flags a check function followed by a subsequent use function as a TOCTTOU error. The analysis results are given in Table 1. The three first columns



CHECK = { access, stat, statfs, statvfs, lstat, readlink, tempnam, tmpnam, tmpnam_r }

USE = { acct, au_to_path, basename, catopen, chdir, chmod, chown, chroot, copylist, creat, db_initialize, dbm_open, dbminit, dirname, dlopen, execl, execle, execlp, execv, execve, execvp, fattach, fdetach, fopen, freopen, ftok, ftw, getattr, krb_recvauth, krb_set_tkt_string, kvm_open, lchown, link, mkdir, mkdirp, mknod, mount, nftw, nis_getservlist, nis_mkdir, nis_ping, nis_rmdir, nlist, open, opendir, pathconf, pathfind, realpath, remove, rename, rmdir, rmdirp, scandir, symlink, system, t_open, truncate, umount, unlink, utime, utimes, utmpname }

Figure 2: Race Condition Automaton (TOCTTOU).

define the package name, the size of the package, and the program that contains coding errors. The number of reported errors is given in the fourth column (Reported Errors). After inspection of the reported error traces, we classify them into three following columns: column (Err) for potential errors, column (FP) for false positive alerts, and column (DN) for traces that are undecidable with manual inspection. The checking time of programs is given in the last column. From Table 1, we demonstrate the efficiency and the usability of our approach in detecting real errors in real-software packages. Moreover, our experiment shows that the use of data-driven mode in our framework enhances the analysis precision. Table 2 summarizes the error traces our tool detected during the experimentation. The properties, the number of reported traces, and the corresponding CERT rules are given in the table, and more details of our experimentation can be found in [TYD09].

4 FOSS Security Hardening

Software security hardening is defined in [MLD06] as *any process, methodology, product or combination that is used to add security functionalities, remove vulnerabilities or prevent their exploitation in existing software*. Security hardening practices are usually manually applied by injecting security code into software [Bis05, HL02, Sea05]. In this section, we address the problems related to the security hardening of FOSS. In this respect, we propose two aspect-oriented and pattern-based approaches for systematic security hardening. The first one is built on top of existing Aspect-Oriented Programming (AOP) technologies while the other one is based on GIMPLE. Both approaches are supported by a common structure, which is based on the full separation between the roles and duties of the security experts and the developers performing the hardening. Such proposition constitutes a bridge that allows the security experts to provide the best solutions to particular security problems with the details on why, how and where to apply them. Moreover, it allows the developers to use these solutions to harden open source software without the need to have high security expertise.

We realize the proposed structure by elaborating a programming independent and aspect-

Table 1: Results of TOCTTOU Analysis.

Package	LOC	Program	Reported Errors	Err	FP	DN	Model-checking time (Sec)
amanda-2.5.1p2	87K	chunker	1	0	1	0	71.6
		chg-scsi	3	2	1	0	119.99
		amflush	1	0	0	1	72.97
		amtrmidx	1	1	0	0	70.21
		taper	3	2	1	0	84.603
		amfetchdump	4	1	0	3	122.95
		driver	1	0	1	0	103.16
		sendsize	3	3	0	0	22.67
		amindexd	1	1	0	0	92.03
at-3.1.10	2.5K	atd	4	3	1	0	1.16
		at	4	3	1	0	1.12
bintuils-2.19.1	986K	ranlib	1	1	0	0	2.89
		strip-new	1	0	1	0	5.49
		readelf	1	1	0	0	0.23
freeradius-server-2.1.3	77K	radwho	1	1	0	0	1.29
inn-2.4.6	89K	nnrpd	1	1	0	0	4.11
		fastrm	1	1	0	0	0.37
		archive	1	0	1	0	0.95
		rnews	1	1	0	0	0.57
openSSH-5.0p1	58K	ssh-agent	2	0	0	2	22.46
		ssh	1	0	1	0	100.6
		sshd	6	3	1	2	486.02
		scp	3	2	0	1	87.95
shadow-4.1.2.2	22.7K	usermod	3	1	0	2	9.79
		useradd	1	1	0	0	11.45
		vipw	2	2	0	0	10.32
		newusers	1	1	0	0	9.2
zebra-0.95a	142K	ripd	1	1	0	0	0.46

oriented based language for security hardening called SHL, developing its corresponding parser, compiler and integrating all of them into a framework for software security hardening. In the following, we present the architecture, the design and implementation as well as the results and experiments of each of the aforementioned two approaches.

4.1 Aspect-Oriented Security Hardening

This approach is based on the Security Hardening Language (SHL) that is defined in [MLD07a, MLD07b]. We have elaborated an aspect-oriented approach to perform security hardening in a systematic way. In this approach, security experts provide security solutions using an abstract

Table 2: Summary of Analysis Results.

Experiment Property	Reported Error	Err	FP	DN	CERT Rule
Race Condition	54	33	10	11	POS35-C, FIO01-C
Temporary File Usage	23	23	0	0	FIO43-C
Chroot Jail	2	1	1	0	POS02-C, FIO16-C
Memory Leak	61	11	13	37	MEM-C
Unchecked Return value	14	14	0	0	MEM32-C, EXP34-C
Environment Variable Usage	11	10	1	0	STR31-C, STR32-C, ENV31-C
Deprecated Function	Too many	-	-	-	FIO33-C, POS33-C, MSC30-C

and a general aspect-oriented language called SHL that is expressive, human-readable, multi-language support. This will relieve developers from the burden of security issues and let them focus on the main functionality of programs. The security solutions are then applied in a systematic way eliminating the need for manual hardening. The approach provides an abstraction over the actions that are required to improve the security of programs and adopt an aspect-oriented approach to build and develop the solutions.

4.1.1 Architecture

We present in Fig. 3 the architecture of this approach. SHL is built on the top of the current AOP technologies that are based on the pointcut-advice model. The solutions elaborated in SHL are expressed by plans and patterns and can be refined into a selected AOP language. Security hardening patterns are high-level and well-defined solutions to known security problems, together with detailed information on how and where to inject each component of the solution into an application. Security hardening plans instantiate security hardening patterns with parameters regarding platforms, libraries and languages. The combination of hardening plans and patterns constitutes a bridge that allows security experts to provide the best solutions to particular security problems and allows developers to use these solutions to harden applications by developing security hardening patterns. The development implies refinement of solutions into advices using the existing AOP languages (e.g., AspectJ, AspectC++).

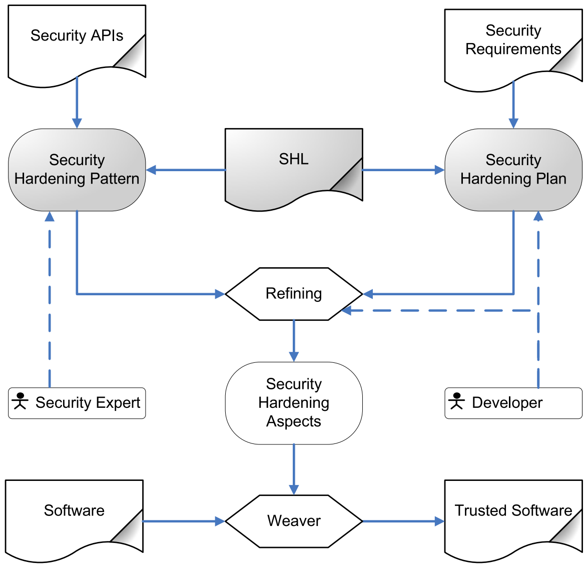


Figure 3: Framework Architecture

4.1.2 SHL Compiler and Framework Implementation

We implement the `BNF` specification of SHL using ANTLR and its associated ANTLRWorks development environment. The generated Java code allows to parse hardening plans and patterns and verify the correctness of their syntax. We build on top of it a compiler that uses the information provided by the parser to build first its data structure, then reacts upon the provided values in order to run the hardening plan and compile and run the specified pattern and its corresponding aspect. Moreover, we integrate this compiler into a development graphical user interface for security hardening. The resulting system provides the user with graphical facilities to develop, compile, debug and run security hardening plans and patterns. It allows also to visualize the software to be hardened and all the compilation and integration activities performed during the hardening. The compilation process is divided into many phases that are performed consequently and automatically. In the sequel, we present and explain these phases.

- *Plan Compilation:* This phase consists of parsing the plan, verifying its syntax correctness and building the data structure required for the other compilation phases. Any error during the execution of this phase stops the whole compilation process and provides the developer with information to correct the bug. This statement also applies on all the other phases.

- *Pattern Compilation and Matching:* A search engine is developed to find the pattern that matches the pattern instantiations requested in the hardening plan (i.e., pattern name and parameters). A naming convention composed of the pattern name and parameters is adopted to differentiate between the patterns with same name but different parameters. Once the pattern-matching the criteria is found, another check on the name and parameters specified inside the pattern is applied in order to ensure that the matching is correct and there is no error in the naming procedure. This includes automatically parsing and compiling the pattern contents to check the correctness of its syntax, verify the matching result and build the data structure required for the running process.

- *Aspect Matching:* Once the pattern is compiled successfully, a search engine similar to the aforementioned one is used to find the aspect corresponding to the matched pattern.
- *Plan Running and Weaving:* Plan running is the last phase of the compilation process. Once the corresponding aspect is matched, the execution command is constructed based on the information provided in the data structure, which is built during the previous compilation phases. Afterwards, the aspect is woven with the specified application or module and the resulted hardened software is produced.
- *Aspect Generation:* Aspect generation is an additional feature launched separately to assist the developer during the refinement of a pattern by generating automatically parts of the corresponding aspect. The generated poincuts and advices are enclosed into an aspect that has the same name as the pattern concatenated to its parameters. The developer will have to refine the advices' bodies into programming language code (i.e, C++ or Java) and then run the plan to apply the weaving.

4.2 GIMPLE-based Software Security Hardening

This approach allows applying the security hardening on the GIMPLE representation of software [Nov03]. GIMPLE is an intermediate representation of programs. It is a language-independent and a tree-based representation generated by the the GNU Compiler Collection (GCC) [GCC] during compilation. GCC is a compiler system supporting various programming languages, e.g., C, C++, Objective-C, Fortran, Java, and Ada. In transforming the source code to GIMPLE, complex expressions are split into three address codes using temporary variables. Exploiting the intermediate representation of GIMPLE enables to define language-independent weaving semantics that facilitates introducing new security-related AOP extensions. The importance of this stems from the fact that aspect-oriented languages are language dependent. Accordingly, GIMPLE weaving allows defining common weaving semantics and implementation for all programming languages supported by the GCC compiler instead of doing them for each AOP language. This approach is also based on the aforementioned Security Hardening Language (SHL).

Fig. 4 illustrates the architecture of the GIMPLE weaving approach together with the one presented in Fig. 3. The GIMPLE weaving approach bypasses the refinement step from patterns into AOP languages. The hardening tasks specified in patterns are abstract and support multiple languages, which makes the GIMPLE representation of software a relevant target to apply the hardening. This is done by passing the SHL patterns and the original software to an extended version of the GCC compiler, which at the end generates the executable of the trusted software. For this purpose, an additional pass is added to the GCC compiler in order to interrupt the compilation once the GIMPLE representation of the code is completed. In parallel, the hardening pattern is compiled and a GIMPLE tree is built for each behavior using the routines of the GCC compiler that are provided for this purpose. Afterwards, the GIMPLE trees generated from the hardening patterns are integrated in the GIMPLE tree of the original code with respect to the

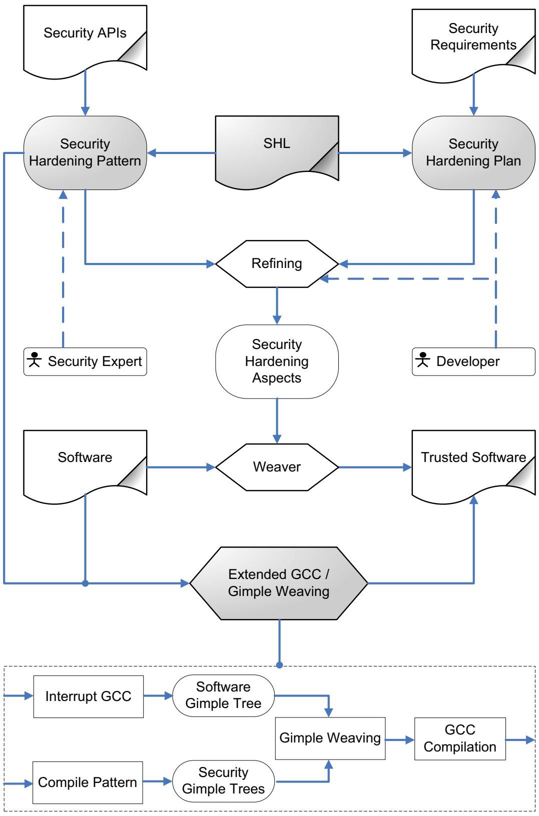


Figure 4: Approach Architecture

location(s) specified in each behavior of the hardening pattern. Finally, the resulted GIMPLE tree is passed again to the GCC compiler in order to continue the regular compilation process and produce the executable of the secure software.

4.2.1 Design and Implementation of Gimple Weaving Capabilities into GCC

This implementation allows weaving patterns into the GIMPLE representation of programs before generating the corresponding executables. We handle *before*, *after*, and *replace* behaviors. In addition, we target *call*, *set*, *get*, and *withincode* locations. The implementation methodology that is adopted consists of the following steps. First, we generate a configuration file from the SHL file. This configuration file contains all the information needed for the weaving using our extended GCC. Then, we use the name of this configuration file as an option in a specific command line of the extended GCC compiler. This compiler, which has weaving capabilities, is an extension to the GCC compiler version 4.2.0. Consequently, three input files are needed by the extended compiler to perform the weaving: a source code, a configuration file, and a library containing the subroutines to be woven. In addition to the above option, it is required to specify the library that contains the code to be woven. This is done through GCC's options `-l` and

-L. Then, a GIMPLE tree is built for the code of each behavior in a pattern. Afterwards, each generated tree is injected in the program tree depending on the insertion point and the location specified in each behavior. Once this weaving procedure is done, the GCC compiler takes over and continues the classical compilation of the modified tree to generate the executable of the hardened program.

4.2.2 Results and Experiments

The main contributions of this approach can be summarized as follows:

- Semantics and algorithms for matching and weaving in GIMPLE are formalized. For this reason, a syntax for a common aspect-oriented language that is abstract and multi-language support and a syntax for GIMPLE constructs are defined.

- Correctness and completeness of GIMPLE weaving are explored from two different views. In the first approach, we address them according to the provided formal matching and weaving rules and the defined algorithms in this paper. On the other hand, we accommodate in the second approach Kniesel's discipline to prove that GIMPLE weaving is correct and complete just in some specific cases because of behavior interactions and interferences.

- Implementation strategies of the proposed semantics are introduced. To explore the viability and the relevance of the defined approach, case studies are developed to solve the problems of unsafe creating of chroot jail, unsafe creating of temporary files, and using deprecated functions.

5 Conclusion

In this paper, we have presented an innovative framework for security evaluation and hardening of free and open-source software. For security evaluation, first a vulnerability detection approach has been proposed. This approach brings into a synergy the static analysis and the model-checking in order to leverage the advantages and overcome the shortcomings of both techniques. We have demonstrated the efficiency and the usability of our approach in detecting real errors in real-software packages. Moreover, our experiment shows that the use of data-driven mode in our framework enhances the analysis precision. It is important to mention that we have also developed a second approach to detect security vulnerabilities that is based on security testing and code instrumentation. This approach has not been detailed in this paper for the lack of space. Finally, we have presented a security hardening approach. The approach is aspect-oriented and performs security hardening in a systematic way. In this approach, security experts provide security solutions using an abstract and a general aspect-oriented language called SHL that is expressive, human-readable, multi-language support. The use of this language relieve developers from the burden of security issues and let them focus on the main functionality of programs. The approach provides an abstraction over the actions that are required to improve the security of programs and adopt an aspect-oriented approach to build and develop the solutions.

Bibliography

[AC03] D. Aspinall, A. B. Compagnoni. Heap Bounded Assembly Language. *Journal of Automated Reasoning* 31:261–302, 2003.

[BDNN01] C. Bodei, P. Degano, F. Nielson, H. R. Nielson. Static Analysis for Secrecy and Non-Interference in Networks of Processes. *Lecture Notes in Computer Science* 2127:27–41, 2001.

[Bis05] M. Bishop. How Attackers Break Programs, and How to Write More Secure Programs. 2005. Available at http://nob.cs.ucdavis.edu/~bishop/secprog/sans2002/index.html. Accessed on 2008/11/11.

[Bol03] T. Bollinger. Use of Free and Open-Source Software (FOSS) in the U.S. Department of Defense. Technical report MP 02W0000101 v1.2.04, MITRE, January 2003.

[CC77] P. Cousot, R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Pp. 238–252. ACM Press, New York, NY, Los Angeles, California, 1977.

[CC04] R. Charpentier, R. Carbone. Free and Open Source Software: Overview and Preliminary Guidelines for the Government of Canada. March 2004. Defence Research and Development Canada – Valcartier.

[cer] CERT Secure Coding Standard. http://www.securecoding.cert.org. Accessed in April 2009.

[CGG02] L. Cardelli, A. Gordon, G. Ghelli. Secrecy and Group Creation. In Hurley et al. (eds.), *Electronic Notes in Theoretical Computer Science*. Volume 40. Elsevier, 2002.

[CKFS01] Y. Coady, G. Kiczales, M. Feeley, G. Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Proceedings of Foundations of software Engineering*. Vienne, Austria, September 2001.

[CM03] J. Cheney, G. Morrisett. A Linearly Typed Assembly Language. Technical report 2003-1900, Department of Computer Science, Cornell University, 2003.

[Cov] Coverity. Coverity Prevent for C and C++. http://www.coverity.com/main.html. Accesses on June 1, 2010.

[Fag76] M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal* 15(3), 1976.

[FR03] A. Fecteau, J. P. Rodrique. Certifying Critical Software: JACC Market Survey. Technical report, Geo Alliance International Inc, June 2003.

[GCC] GCC-the GNU Compiler Collection. Available at http://gcc.gnu.org/. Accessed on 2009/6/1.

[Gre] L. Grenier. Practical Code Auditing. *OpenBSD Journal*.

[Her03] P. Herzog. Open-Source Security Testing Methodology Manual. Institute for Security and Open Methodologies (ISECOM), August 2003.

[HL02] M. Howard, D. E. LeBlanc. *Writing Secure Code*. Microsoft, Redmond, WA, USA, 2002.

[HMS03] K. W. Hamlen, G. Morrisett, F. B. Schneider. Computability Classes for Enforcement Mechanisms. Technical report TR2003-1908, Cornell University, Computing and Information Science, Ithaca, New York, August 2003.

[KHH$^+$01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold. An Overview of AspectJ. In *Proceedings of the 2001 European Conference on Object-Oriented Programming (ECOOP'01)*. 2001.

[Kim02] H. Kim. AspectC#: An AOSD implementation for C#. Technical report TCD - CS2002-55, Department of Computer Science, Trinity College, Dublin, 2002.

[KLM$^+$97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin. Aspect-Oriented Programming. In Akşit and Matsuoka (eds.), *Proceedings European Conference on Object-Oriented Programming*. Volume 1241, pp. 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[Koz98] D. Kozen. Efficient Code Certification. Technical report 98-1661, Computer Science Department, Cornell University, January 1998.

[KSS] S. Kiefer, S. Schwoon, D. Suwimonteerabuth. Moped - A Model-Checker for Pushdown Systems. http://www.fmi.uni-stuttgart.de/szs/tools/moped/. Accessed on January 20, 2009.

[Lab03] C. Labs. An Aspect-Oriented Security Assurance Solution. Technical report AFRL-IF-RS-TR-2003-254, Cigital Labs, Dulles, Virginia, USA, Oct 2003.

[MCGW98] G. Morrisett, K. Crary, N. Glew, D. Walker. Stack-Based Typed Assembly Language. In *tic*. Lecture Notes in Computer Science 1473, pp. 28–52. springer, Kyoto, Japan, March 1998.

[MLD06] A. Mourad, M.-A. Laverdière, M. Debbabi. Security Hardening of Open Source Software. In *Proceedings of the 2006 International Conference on Privacy, Security and Trust (PST 2006)*. McGraw-Hill/ACM, 2006.

[MLD07a] A. Mourad, M.-A. Laverdière, M. Debbabi. A High-Level Aspect-Oriented based Language for Software Security Hardening. In *Proceedings of the International Conference on Security and Cryptography (Secrypt)*. Barcelona, Spain, 2007.

[MLD07b] A. Mourad, M.-A. Laverdiere, M. Debbabi. Towards an Aspect Oriented Approach for the Security Hardening of Code. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops, AINAW '07*. Pp. 595–600. 2007.

[MM00] G. McGraw, G. Morrisett. Attacking Malicious Code: A Report to the Infosec Research Council. *IEEE Software* 5(17), September/October 2000.

[MWCG99] G. Morrisett, D. Walker, K. Crary, N. Glew. From System F to Typed Assembly Language. 21(3):528–569, May 1999.

[Nec97] G. Necula. Proof-Carrying Code. In *24th POPL*. Pp. 106–119. Paris, France, January 1997.

[Nov03] D. Novillo. Tree SSA: A New Optimization Infrastructure for GCC. In *Proceedings the GCC Developers Summits3*. Pp. 181–193. May 25-27 2003.

[Pol] PolySpace. Automatic Detection of Run-Time Errors at Compile Time. http://www.polyspace.com/.

[RW02] A. Rudys, D. S. Wallach. Enforcing Java Run-Time Properties Using Bytecode Rewriting. In *Proceedings of the International Symposium on Software Security*. Tokyo, Japan, November 2002.

[Sea05] R. C. Seacord. *Secure Coding in C and C++*. SEI Series. Addison-Wesley, 2005.

[SGS02] O. Spinczyk, A. Gal, W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems*. Sydney, Australia, February 2002.

[SWM00] F. Smith, D. Walker, G. Morrisett. Alias Types. *Lecture Notes in Computer Science* 1782:366–381, 2000.

[TO00] P. Tarr, H. Ossher. HyperJ User and Installation Manual. 2000. http://www.research.ibm.com/hyperspace. Accessed on June 1,2010.

[TYD09] S. Tlili, X. Yang, M. Debbabi. Verification of CERT Secure Coding Rules: Case studies. In *International Symposium on Information Security*. Springer Verlag, 2009.

[Whe] D. A. Wheeler. FlawFinder. http://www.dwheeler.com/flawfinder/. Accessed in June, 2010.

[WPJV02] B. D. Win, F. Piessens, W. Joosen, T. Verhanneman. On the Importance of the Separation-of-Concerns Principle in Secure Software Engineering. 2002. Workshop on the Application of Engineering Principles to System Security Design, Boston, MA, USA, November 6–8, 2002, Applied Computer Security Associates (ACSA).

[WTS03] J. Wack, M. Tracy, M. Souppaya. Guideline on Network Security Testing. Nist special publication 800-42, National Institute of Standards and Technology (NIST), October 2003.

[XH99] H. Xi, R. Harper. Dependently Typed Assembly Language. Technical report OGI-CSE-99-008, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, July 1999.

Security in Open Model Software with Hardware Virtualization – The Railway Control System Perspective

Johannes Feuser[1] **and Jan Peleska**[2]

[1] jfeuser@informatik.uni-bremen.de
[2] jp@informatik.uni-bremen.de
Department of Mathematics and Computer Science
University of Bremen, Germany
http://www.informatik.uni-bremen.de/agbs/index_e.html

Abstract: Using the openETCS initiative as a starting point, we describe how open software can be applied in combination with platform-specific, potentially closed-source extensions, in the development, verification, validation and certification of safety-critical railway control systems. We analyse the safety and security threats presented by this approach and discuss conventional operating system partitioning mechanisms, as well as virtualisation methods with respect to their potential to overcome these problems. Furthermore, we advocate a shift from open source to open models, in order to increase the development efficiency of combined open and proprietary solutions.

Keywords: openETCS, open source, open model, security, hardware virtualization

1 Introduction

1.1 Background

By the end of 2009 German Railways initiated a discourse on the possible benefits of using *Free/Libre Open Source Software (FLOSS)* in railway control systems, with special focus on the *European Train Control System ETCS*. This initiative was labelled *openETCS* [Has09b, Has09a]. Reviewing evidence where security threats had been purposefully integrated into closed-source commercial software products, the author argued that open source software could be useful – perhaps even mandatory in the future – to ensure safety and security of railway control systems: even though the standards applicable for safety-critical systems software development in the railway domain [CEN01a, CEN99] require independent-party verification and validation, the complexity of the source code on the one hand and the limited budget available for V&V on the other hand can only mitigate the threat of safety and security vulnerabilities, but cannot guarantee to uncover all compromising code components inadvertently or purposefully injected into the code. As a consequence, in addition to the V&V efforts required by the standards, the broad peer-review enabled by publicly available software could really increase software safety and security[1]. German Railways indicated that also *open proofs* might be necessary to complement

[1] Following [Lev95] we agree that safety and also security are *emergent properties*, that is, they can only be attributed to complete systems, and not to software alone. When we use the terms *software safety* and *software security* in this paper, we mean *absence of software malfunctions that may lead to safety or security hazards on system level.*

the open source code, but they did not comment on the necessity to publish software models, specifications and on the potential of an open certification process.

Initially, the openETCS position statement stirred considerable interest, but has become somewhat quiet recently, at least on the public level. We suspect that this is due to the fact that railway suppliers are currently evaluating the impact of these requirements on their business models which are still based on closed software and supplier-specific solutions, in order to protect their intellectual property. In parallel German Railways will still be investigating the leverage it may already have or will gain in the future on its suppliers in order to enforce the open software idea[2]. Should German Railways – potentially supported by research communities investigating the potential of open source software in the safety-critical domain – succeed in promoting openETCS, this would automatically become an international European topic: since ETCS is a European effort to provide high-speed railway transport across borders, and since suppliers in many European countries contribute to ETCS systems and software development, success or failure of the openETCS initiative will eventually be established on European level, and not just nationally in Germany.

1.2 Objectives and Overview

This contribution is a combination of a position paper and an elaboration of solution approaches to the openETCS scenario. We argue in Section 2 that the underlying development, V&V and certification approaches enforced by the standards [CEN01a, CEN99] require that not only software, proofs (or semi-formal verification arguments) and verification tools should be published, but that the open-source paradigm should be lifted to an *open-model paradigm*, in combination with open code generators and V&V tools.

Our expectation is that – due to functional extensions, adaptations to specific hardware and national rules with impact on railway control algorithms – the open source software will nearly always have to be modified and/or enhanced by platform-specific code (Section 3). These adaptations may still be closed software or – even if made publicly available – not be of sufficient general interest to stimulate a public peer reviewing process. As a consequence we envision a scenario where future railway control systems are developed as enhancements and refinements of open models where a portion of the code has been certified according to the OpenCert paradigm and will remain unchanged in most applications, but this *re-usable core* is complemented by less trustworthy additions. Analysing the remaining safety and security threats of this scenario, we show that it can be compared to the *grey-channel paradigm* where safety-critical dependable distributed applications have to communicate over potentially unsafe channels. This situation is nowadays standard practice in distributed railway control applications and the standard [CEN01c] defines how to ensure safety and security of the resulting system, at the potential risk of reducing the availability of the system, due to fail-safe blocking of further operation.

Based on the grey-channel scenario we discuss in Section 4 how conventional operating systems mechanisms may help to reduce the safety and security risks presented by this scenario. As a final step (Section 5) we advocate the utilisation of virtualisation in order to further reduce

[2] Needless to say that, due to the possibility to re-use FLOSS, German Railways also expect a decrease of software development costs by the openETCS initiative, because suppliers would not need to re-implement major portions of the publicly available railway control algorithms.

these risks: trusted core software and target-specific adaptations run in different virtual machines, communicating according to the grey channel paradigm as if distributed over a network. We discuss the impact of this approach on the future development of virtual machines, hypervisors and communication interfaces.

Section 6 contains the conclusion.

1.3 Related Work

Our work is motivated by the challenges formulated by German Railways and the openETCS initiative [Has09b, Has09a], and uses the development and railway application scenarios presented there as a starting point. Certification issues of safety-critical systems in general are described in [Sto96]; the work presented here is specialised on the railway domain where the standards [CEN01a, CEN03, CEN99, CEN01b, CEN01c] apply. While – as described in [SC09] – quality and certification issues concerning open software in general still leave many open questions to be tackled, the railway control systems scenario described in this paper relies on certification according to the rules defined in the standards listed above. The only differences to today's standard procedure are that (1) the certified code and its associated documentation are made publicly available and (2) it may be necessary to re-certify the software as soon as adaptations and extensions have been made for a concrete system implementation.

The model-driven approach advocated in this paper is based on domain-specific modelling as decribed in [KT08] because it is well-known that the utilisation of domain-specific description formalisms and associated automated code generation and mechanised model-based testing and verification has high potential in the railway domain [HP03, HPK09, Mew09]. It has to be emphasised, however, that the open-model approach and the security analyses presented in this paper only rely on the availability of an arbitrary specification formalism that is suitable for formal verification and automated code generation. Even conventional UML2 [OMG03a, OMG03b] (and potential augmentations by means of the profile mechanism) are suitable if a well-defined model-to-text (i. e. code) transformation is used to associate a transformational semantics with the semi-formal UML model [BBHP06].

2 From Open Source to Open Model Software

The terms open source software (OSS) and free/libre open source software (FLOSS) refer to source code. Certifiable train control systems software, on the other hand, has to be complemented by a collection of additional artifacts contributing to the *safety case*, that is, the comprehensive and structured evidence justifying that the resulting system will guarantee safe operation. Among others (for details see [CEN01a, CEN99]), the list of these artifacts comprises software specification and design models and complete records of all V&V measures taken to ensure software code compliance with its applicable specifications, as well as evidence showing how all functional and structural aspects of the software have been throughly tested and verified[3]. It is well known that for systems of highest assurance level[4] the effort for elaboration of the safety

[3] The term *verification* comprises formal mathematical analyses, as well as semi-formal reviews and inspections.

[4] The so-called *system integrity level SIL-4* and the associated *software safety assurance level SSAS-4*.

case is frequently higher than the proper software development effort. As a consequence, just source code without the additional artifacts mentioned above would be nearly worthless. Additionally, as soon as FLOSS code has to be adapted, this will become quite hard and often invalidate previous V&V results if these adaptations have not been guided by a systematic approach, preferably based on a software model giving indications how to modify the software in an admissible way.

Due to these considerations we are convinced that OSS/FLOSS can only be applied successfully in the railway control systems domain if code is accompanied by or – even better – embedded in free/libre open *models*. Following the principles of object-oriented modelling, the description formalism should be based on a meta-meta model that is publicly available as in the case of the OMG meta object facility [SVE07] or in the case of the *Graph, Object, Property, Role and Relationship (GOPRR)* meta-meta model introduced in [KT08] for the design of domain-specific languages, so that the model could be unambiguously interpreted and processed by various development and V&V tools. Additionally, these models should clearly indicate where platform-specific or application-specific changes are admissible by means of class inheritance, overriding and overloading of operations or by means of adding components with admissible interfaces.

As sketched in Figure 1 we suggest the terms *open meta metamodel*, *open metamodel*, and *open model* for the higher-level abstractions required "above" the open software. Figure 1.

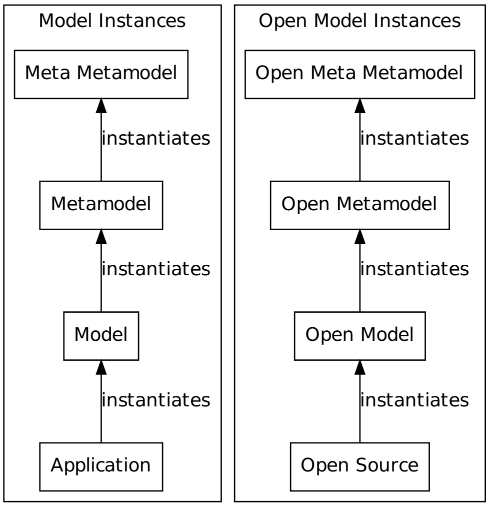


Figure 1: Denomination for open domain-specific modelling (DSM).

A typical benefit from this approach would be that certification credit for module verification could be re-used for all software methods or functions that have not been changed for the platform-specific adaptation. On the other hand, these adaptations may require extensive new V&V activities and associated re-certification for the complete system, if their impact on the re-usable components is not clearly visible. This problem will be analysed more closely in the sections below.

Re-usable certifiable open model software also requires a specification of the admissible tool chain to be used for model-to-text transformations, compilation and linking and V&V regression activities, because otherwise it could not be guaranteed that the software build process would be performed correctly and the V&V process would lead to trustworthy results. These tool aspects, however, are beyond the scope of this paper.

3 Security Analysis for The Open Model Software Scenario

Figure 2 shows a very general scenario how a platform-specific adaption of an open model and associated FLOSS could compromise the resulting system. This example has one model implementation which is directly generated from the open model, and therefore gets certification credit by means of re-use for all component-specific V&V artifacts. Suppose that sub-models 2 and 3 had to be newly developed for the platform-specific solution, resulting in supplier implementations 1 and 2. It is obvious that component-specific V&V measures have to be performed for these new implementations. We are interested in the question whether some certification credit could be re-used for model implementation 1 on software integration level, for example, the V&V measures previously taken to show that this implementation cooperates correctly with other components directly generated from the open model.

Unfortunately, this is not true without further restrictions: if implementation 2 is malicious it may compromise both model implementation 1 and supplier implementation 1, either by sending corrupted data through their designated interfaces or through covert channels which were not intended to be utilised according to the model[5], or by means of unintended resource usage creating denial of service attacks.

As a consequence no certification credit can be re-used for model implementation 1 on software integration level: All corresponding V&V artifacts have to be re-produced in order to justify that none of the platform-specific implementations can compromise the resulting system through any of the other implementations. In the two following sections we will analyse suitable measures to counter the threat presented by such malicious implementations.

4 Partitioning

As seen in the previous section, the creation of faulty or malicious supplier implementations in an open model scenario cannot be completely avoided, but their impact on other software components should be minimised. Modern operating systems offer a number of standard mechanisms to cope with these situations. All of these mechanisms may be summarised under the keyword *partitioning*, which has to be enforced in the resource domain and in the time domain.

In the resource domain partitioning means that faulty or malicious components cannot interfere with the (legal) access of another software component to the resource and cannot access any resource without proper authorisation. Typical resources in the embedded systems domain

[5] E. g., by writing to illegal memory addresses if all implementations run as operations or threads in the same address space.

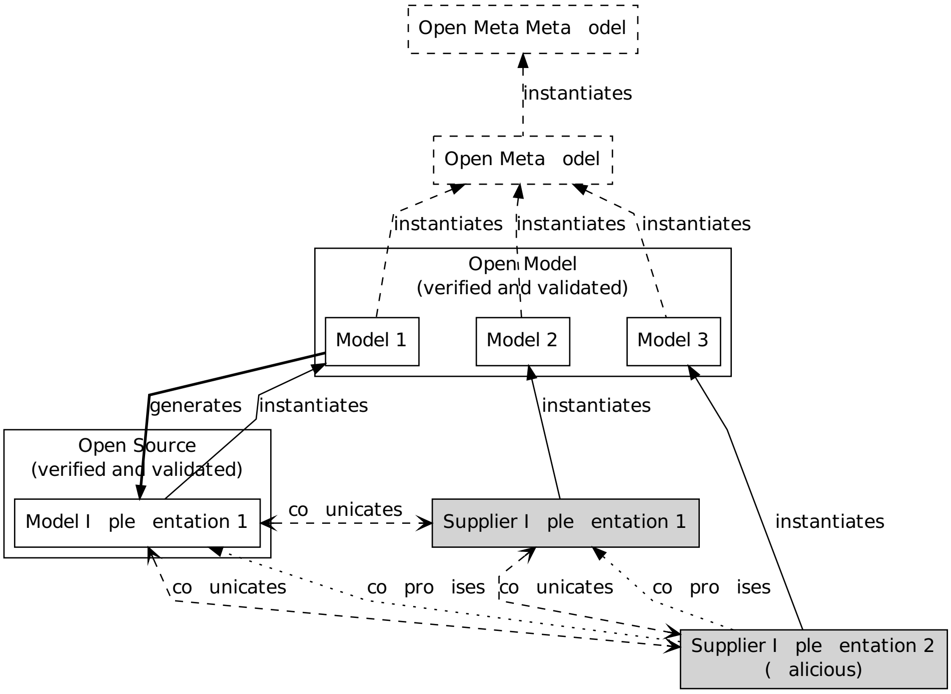


Figure 2: Possible security threats in open model software combined with platform-specific adaptations.

are CPU cores, memory, hardware and software interfaces and operating system resources like semaphores, message queues and others. The traditional way of implementing resource partitioning is through different privilege levels for application and operating system layer, virtual address spaces supported by memory management units, encapsulation of resource access by means of system calls and kernel access mechanisms and access control mechanisms enforced by the operating system [Sta08a, Sta08b]. Currently, resource partitioning is typically static for safety-critical embedded systems, since the dynamic allocation and de-allocation during system operation is hard to verify, or – as in the case of dynamic memory partitioning with paging – unsuitable for the embedded domain as long as suitable solid-state disks are not available.

In the time domain partitioning implies that corrupt components may not access any resource – in particular, the CPU and the communication interfaces – for an undue amount of time, thereby creating denial of service attacks. Time partitioning is typically enforced by means of schedulers; prominent examples are partition (process) schedulers complying with the ARINC specification 653P1-2 [ARI05] defining a distributed operating system as used in modern avionics (e. g. Airbus A380)[6]. On the interface level, the Avionics Full-Duplex Switched Ethernet Network guarantees fixed communication bandwidths for different communication links by means of an on-board scheduler for package transmission [ARI09] (also used in the Airbus A380 and in modern Boeing aircrafts). Alternatively, the Time Triggered Protocol (TTP) [TTP10] assigns temporal communication slots to processes.

5 Hardware Virtualization with Open Models

Para Virtualisation. The conventional mechanisms enforcing partitioning described in the previous section have the draw back that they require all software components to run under the regime of a single operating system. At least in the current situation, where several on-board train controllers are required in order to cope with national boundary conditions, this is disadvantageous for openETCS, because of the diversity of supplier hardware and associated operating systems. This problem also has implications on the open model approach: if the code generated from these models relies on the availability of specific operating system mechanisms (for example, a certain scheduling policy), this code may only run on platforms whose operating systems support these mechanisms. This impairs the potential re-use advantages of the open model approach in a considerable way.

As a solution to this problem we suggest *hardware virtualisation*, where – controlled by a *hypervisor* and a host operating system – several *guest* operating systems may run simultaneously in so-called *virtual machines (VM)* on the same hardware [vmw07]. A hypervisor works as a *virtual machine monitor (VMM)* which either dispatches sensitive instructions issued by a guest operating system that require kernel privileges to the hardware or emulates these instructions by means of interaction with the host operating system. In the latter case the hypervisor may have the capabilities of a micro kernel in its own right and may even render an additional host operating system superfluous. This is the case when so-called *para virtualisation* is applied:

[6] Standard [ARI05] only requires to assign guaranteed time slices to partitions in round-robin manner. This does not guarantee that applications will meet their deadlines. In [MHG^{+}09], a more sophisticated approach based on *earliest deadline first scheduling* is described

Here the sensitive actions of guest operating systems are not dealt with on machine instruction level, but instead the guest utilises a pre-defined hypervisor API providing hardware access on a higher level of abstraction, thereby considerably improving the performance of applications running in virtual machines (see [Tan08, pp. 568] for a more detailed overview).

The most important micro kernel capabilities that we suggest for hypervisors supporting para virtualisation are

- a preemptive round robin scheduler enforcing fixed execution time windows for each virtual machine, similar to the inter-partition scheduling requirements of [ARI05],
- driver management for hardware interface access with explicit assignment of interface visibility to selected virtual machines,
- control of the memory management unit to enforce memory partitioning and assign either fixed memory portions to virtual machines or limit each VM's amount of dynamically allocated memory,
- communication mechanisms supporting message-based inter-VM and remote communication.

Open Model Scenario With Virtualisation. In this virtualisation scenario, the code portions generated directly from the model without platform-specific adaptations would run in one virtual machine, and platform-specific adaptations would run in separate virtual machines. Since each virtual machine mimics a complete computer with its local operating system, platform hardware and peripherals, resource partitioning is easily enforced: hardware interfaces that should not be accessed by a group of software components are simply not visible in their "virtual computer hardware". The utilisation of main memory could be limited by the hypervisor, and the separation of memory address spaces is already enforced on virtual machine level. Communication between virtual machines can be performed, for example, by means of a socket interface.

The effect of virtualisation is similar to several distributed application programs cooperating by means of remote communication. The impact of a malicious or otherwise faulty component is reduced to corrupt communication behaviour on the intended interfaces: it is impossible to influence the outside world by other interfaces but the ones configured for the virtual machine. Since from the viewpoint of the receiver it cannot be distinguished whether the sender or the communication channel is corrupt, this situation is already well understood in today's distributed railway control applications communicating over public networks known as *grey channels*: the safety-relevant components have to be developed on the basis that any type of error may occur on the grey channel, because this is a communication medium whose hardware and software has not been developed with the same assurance level as the safety-critical application itself. As a consequence, the safety-relevant component has to cope with repetition, deletion, insertion, resequence, corruption and delay of messages and guarantee fail-safe behaviour in presence of these faults. The defence mechanisms against these types of faults or attacks have to comply with the standard [CEN01c].

Applying the concept of hardware virtualisation to the initial open model scenario in Figure 2 leads to the revised scenario depicted in Figure 3. It also contains the generated model implementation and the two supplier implementations. In contrast to Figure 2 all supplier implementations

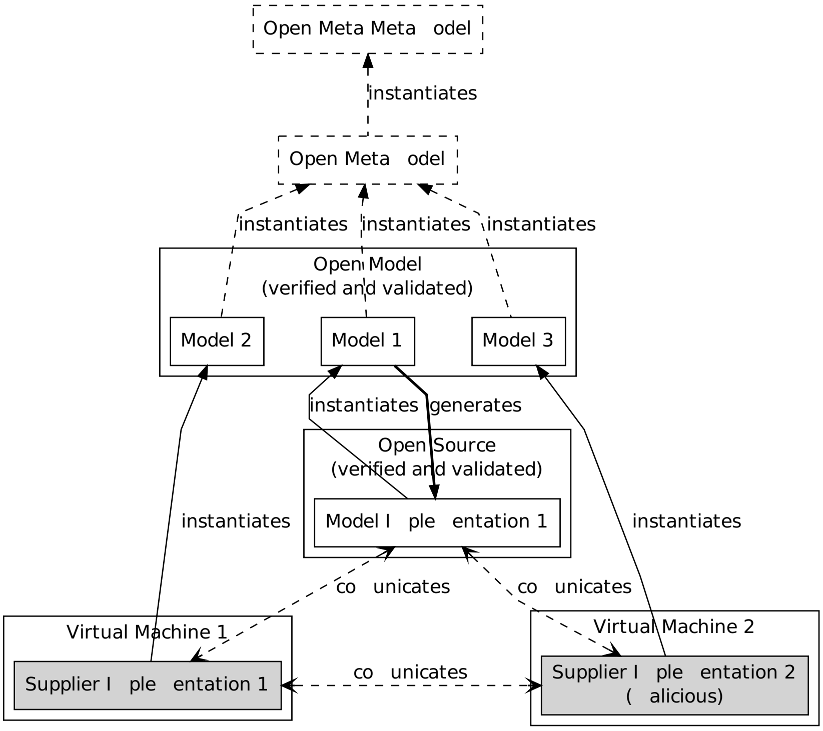


Figure 3: Hardware virtualisation for open models

are now locked in their own virtual machines. This ensures that the malicious implementation cannot compromise any other part of the software through covert channels or abuse of resources, while a communication of legal or corrupted data over the intended channels is possible.

The prevention of undue bandwidth consumption on hardware interfaces can be handled by means of scheduled I/O as described in Section 4. As a consequence, the need of certified high-integrity hypervisors or host operating systems arises. The effort to develop, verify and certify these is justified as soon as the hardware platform can be re-used in different application scenarios, so that hypervisor or host operating system would be re-used as well.

Certification Issues. We advocate the following development, validation and certification approach in the open-model scenario with virtualisation as described above:

- The hardware platforms for railway control systems should be equipped with a hypervisor possessing the micro kernel qualities listed above.
- This hypervisor should be open source and fully certified according to the aforementioned standards and according to the highest assurance level SSAS-4, because all further assurance considerations depend on the trustworthiness of this component.
- The re-usable core of the open-model software should be developed and fully validated with respect to one suitable operating system. In particular, the safe behaviour in presence of corrupt interface data received over a grey channel can be checked once and for all.
- Platform-specific or other functional adaptations should only be admissible as model derivations that may run in separate virtual machines which do not host the re-usable core software[7].
- The adaptations are again validated according to the applicable railway standards, running in an operating system possibly differing from the one hosting the re-usable core.
- Both the re-usable core and the adaptations use a remote communication paradigm to communicate with each other and integrate the required protection mechanisms for grey channel communication.
- The admissible operating systems for re-usable core and adaptations have to comply with the hypervisor API according to the para virtualisation paradigm.
- For an integrated HW/SW system consisting of several virtual machines with guest operating systems hosting the re-usable core and one or more adaptations, certification credit for the local validation activities[8] already performed can be granted.
- For certification of the integrated HW/SW system it remains to validate the following structural, functional and non-functional system properties:

[7] So, for example, simple overloading of some operations in a class belonging to the re-usable core would not be allowed.

[8] Such as module tests and SW integration tests, or partitioning properties for different processes that belong to the same adaptation, and will therefore run in the same VM on the target system.

  - Correct communication among virtual machines and between VMs and interfaces.
  - Correct functional behaviour of the integrated system: To this end, only functional requirements involving two or more virtual machines have to be tested.
  - Performance and robustness in avalanche (stress) situations.

Proof of Concept. We intend to substantiate the advantages of open model openETCS advocated so far by means of a case study. For this purpose the ETCS would be particularly well-suited because the existing open standard [ETC07] may serve as an informal specification, to be formalised as a model conforming to our domain-specific meta model which is currently under development (see Section 2). Typically, the model holds objects directly corresponding to hardware elements like sensors or actuators, e.g. a reader device for Balises [ETC06]. Such elements are often subject to supplier-specific implementations.

To compare conventional operating system methods like process scheduling and memory management with the usage of hardware virtualisation, effects on the rest of the software have to be measured for both cases, in presence of one or more malicious supplier implementations. Therefore we will purposefully generate "supplier" implementations showing the relevant types of malicious behaviours, based on a formal threat model. Examples for these threats are:

- denial of Service attacks on
  - CPU bandwidth,
  - network interface bandwidth,
  - software interfaces of other objects,
- injection of false data to software interfaces of other objects,
- infinite blocking of calls by other objects.

The results of these tests with and without hardware virtualisation could be directly compared and would lead to a conclusion about the efficiency of the virtualisation approach.

It is obvious that hardware virtualisation cannot prevent all of the above mentioned attacks from affecting other software components. Therefore the fault tolerant behaviour of software implementations is highly relevant. A possible solution would be the utilisation of a standardised interface library, e.g. CORBA [HV99], providing methods to handle time-outs and other related problems. CORBA is not needed to be included in the metamodel, but only in the code generator [KT08]. Therefore, this approach would not add additional complexity to the metamodel and its model instances.

The distribution of the software as open models is another aspect of the concepts proposed here. To attract a community of substantial size and adequate competence it is crucial to provide a comprehensive tool chain with the open models. Obviously, the editors and compilers sufficient for open source distribution have now to be complemented with meta-modelling tools, modelling tools and code generators under open source licenses. Moreover, the tool set should be extended by a simulation and visualisation platform so that different solutions could be tried out without the availability of real-world railway infrastructure.

6 Conclusion

We have described an approach for combining open source software and proprietary system-specific code for the development of certifiable railway control systems. Following the certification requirements of applicable standards in the railway control systems domain, this approach requires not only code, but also models and V&V artifacts to be made publicly available. For ensuring the safety of a mixed open/closed source system, we have analysed the support mechanisms offered by today's operating systems in order to prevent software components of minor trustworthiness to corrupt the behaviour of the trusted safety-critical core. In particular, we advocate virtualisation mechanisms to encapsulate components of different assurance levels for achieving fault containment. Virtual machines running components of different assurance levels may communicate according to the grey channel paradigm which is already well understood in today's distributed railway control applications.

Our contribution was intended as a position statement and an indication of promising solutions. The justification of these claims is currently elaborated by means of case studies based on the ETCS specification [ETC07, ETC06] and the *Positive Train Control (PTC)* system concept [PTC10]. To this end, a domain-specific description formalism specialised on the railway control system domain and following the concepts explained in [HP03, HPK09, Mew09] will be used.

The technical effort for substantiating a proof of concept should be accompanied by an open discussion about how to attract an open source community of sufficient size to the openETCS idea: only if the number of actively contributing members is big enough, the desired effect of quality improvement by peer-review, test or analysis can be expected. We believe that such numbers can be reached because – due to the complexity of control objectives on the one-hand and to their illustrative meaning on the other hand – this application domain has always attracted practitioners and researchers in the safety-critical systems and formal methods domains.

Though this paper focused on the railway domain, we expect that the approach described here will be valuable for other safety-relevant domains as well, in particular for avionic systems. Our work has been influenced by the experience of the second author with validation of safety-critical railway control and avionic systems.

Acknowledgements. The first author has been supported by Siemens AG through a research grant of the Graduate School on Embedded Systems GESy at the University of Bremen (`http://www.informatik.uni-bremen.de/gesy`).

References

[ARI05] *Avionics Application Software Interface, Part 1, Required Services*. AERONAUTICAL RADIO, INC., 2551 Riva Road, Annapolis, Maryland 21401-7435, 12 2005.

[ARI09] *Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network*. AERONAUTICAL RADIO, INC., 2551 Riva Road, Annapolis, Maryland 21401-7435, 09 2009.

[BBHP06] K. Berkenkötter, S. Bisanz, U. Hannemann, J. Peleska. The HybridUML Profile for UML 2.0. *International Journal on Software Tools for Technology Transfer (STTT)* 8(2):167–176, January 2006. Special Section on Specification and Validation of Models of Real Time and Embedded Systems with UML.

[CEN99] CENELEC. *EN 50126 - Railway applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)*. CENELEC European Committee for Electrotechnical Standardization, Central Secretariat: rue de Stassart 35, B - 1050 Brussels, 09 1999.

[CEN01a] CENELEC. *EN 50128 - Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems*. CENELEC European Committee for Electrotechnical Standardization, Central Secretariat: rue de Stassart 35, B - 1050 Brussels, 03 2001.

[CEN01b] CENELEC. EN 50159-1. Railway applications -Communication, signalling and processing systems Part 1: Safety-related communication in closed transmission systems. 2001.

[CEN01c] CENELEC. EN 50159-2. Railway applications -Communication, signalling and processing systems Part 2: Safety related communication in open transmission systems. 2001.

[CEN03] CENELEC. *EN 50129 - Railway applications - Communication, signalling and processing systems - Safety related electronic systems for signalling*. CENELEC European Committee for Electrotechnical Standardization, Central Secretariat: rue de Stassart 35, B - 1050 Brussels, 02 2003.

[ETC06] ERTMS/ETCS - Class 1 System Requirements Specification. 24-02 2006. Issue 2.3.0.

[ETC07] ERTMS/ETCS Functional Requirements Specification FRS. 21-07 2007. Version 5.0.

[Has09a] K. R. Hase. openETCS - Ein Vorschlag zur Kostensenkung und Beschleunigung der ETCS-Migration. *SIGNAL +DRAHT* 10, 10 2009.

[Has09b] K. R. Hase. openETCS - Open Source Software für ETCS-Fahrzeugausrüstung. *SIGNAL +DRAHT* 12, 12 2009.

[HP03] A. E. Haxthausen, J. Peleska. Generation of Executable Railway Control Components from Domain-Specific Descriptions. In *Proceedings of the Symposium on Formal Methods for Railway Operation and Control Systems (FORMS'2003), Budapest/Hungary*. Pp. 83–90. L'Harmattan Hongrie, May 15-16 2003.

[HPK09] A. E. Haxthausen, J. Peleska, S. Kinder. A formal approach for the construction and verification of railway control systems. *Formal Aspects of Computing* 17, December 2009. DOI: 10.1007/s00165-009-0143-6.

[HV99] M. Henning, S. Vinoski. *Advanced CORBA Programming with C++*. Addision-Wesley Publishing Company, 1999.

[KT08] S. Kelly, J.-P. Tolvanen. *Domain-Specific Modeling*. JOHN WILEY & SONS, INC., 2008.

[Lev95] N. G. Leveson. *Safeware*. Addison-Wesley, 1995.

[Mew09] K. Mewes. *Domain-specific modelling of railway control systems with integrated verification and validation*. PhD thesis, University of Bremen, 2009.

[MHG$^+$09] A. Mancina, J. Herder, B. Gras, A. Tanenbaum, G. Lipari. Enhancing a Dependable Multiserver Operating System with Temporal Protection via Resource Reservation. *Real-Time Systems* 43:177–210, 2009.

[OMG03a] OMG. UML 2.0 Infrastructure Specification, OMG Adopted Specification. http://www.omg.org/cgi-bin/apps/doc?ptc/03-09-15.pdf, September 2003.

[OMG03b] OMG. UML 2.0 Superstructure Specification, OMG Adopted Specification. http://www.omg.org/cgi-bin/apps/doc?ptc/03-08-02.pdf, August 2003.

[PTC10] Positive Train Control - Wikipedia. URL, http://en.wikipedia.org/wiki/Positive_Train_Control, 2010.

[SC09] S. A. Shaikh, A. Cerone. Towards a metric for Open Source Software Quality. In *Proceedings of the Third International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2009)*. Volume 20. 2009.

[Sta08a] W. Stallings. *Operating systems: internals and design principles*. In [Sta08b], chapter 7 - 8, pp. 353 – 453, 2008.

[Sta08b] W. Stallings. *Operating systems: internals and design principles*. Prentice Hall, 2008.

[Sto96] N. Storey. *Safety critical computer systems*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1996.

[SVE07] T. Stahl, M. Völter, S. Efftinge (eds.). *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Volume 2, chapter 5, pp. 64–71. dpunkt-Verl., 2007.

[Tan08] A. S. Tanenbaum. *Modern Operating Systems*. Pearson, 2008.

[TTP10] Real-Time Systems Research Group: The TTP Protocols. URL, http://www.vmars.tuwien.ac.at/projects/ttp/ttpmain.html, 06 2010.

[vmw07] vmware. Understanding Full Virtualization, Paravirtualization, and Hardware Assist. 08 2007. white paper.

A Deductive Verification Platform for Cryptographic Software

M. Barbosa[1], **J. Pinto**[1], **J.-C. Filliâtre**[23] **and B. Vieira**[1]

CCTC/Departamento de Informática, Universidade do Minho[1]
INRIA Saclay - Île-de-France, ProVal, Orsay[2]
LRI, Université Paris-Sud, CNRS, Orsay[3]

Abstract: In this paper we describe a deductive verification platform for the CAO language. CAO is a domain-specific language for cryptography. We show that this language presents interesting challenges for formal verification, not only in the rich mathematical type system that it introduces, but also in the cryptography-oriented language constructions that it offers. We describe how we tackle these problems, and also demonstrate that, by relying on the Jessie plug-in included in the Frama-C framework, the development time of such a complex verification tool could be greatly reduced. We base our presentation on real-world examples of CAO code, extracted from the open-source code of the NaCl cryptographic library, and illustrate how various cryptography-relevant security properties can be verified.

Keywords: formal program verification, cryptography

1 Introduction

Background. The development of cryptographic software is clearly distinct from other areas of software engineering. Cryptography is an inherently interdisciplinary subject. The design and implementation of cryptographic software draws on skills from mathematics, computer science and electrical engineering. However, there is a clear lack of domain specific languages and tools for the development of cryptographic software that can assist developers in facing these challenges. The CACE project (http://www.cace-project.eu) addresses this lack of support by pursuing the development of an open-source toolbox comprising languages, tools and libraries tailored to the implementation of cryptographic algorithms and protocols. The formal verification tool described in this work has been developed to allow the static analysis of code written in CAO, a new domain-specific language developed as a part of the CACE effort. Currently, this tool is being employed in the formal verification of an open-source library written in CAO. The results in this paper already reflect a part of this verification effort.

The CAO language[Bar09] allows practical description of cryptography-relevant programs. Unlike languages used in mathematical packages such as Magma or Maple, which allow the description of high-level mathematical constructions in their full generality, CAO is restricted to enabling the implementation of cryptography kernels (e.g. block ciphers and hash functions) and sequences of finite field arithmetics (e.g. for elliptic curve cryptography). CAO has been designed to allow the programmer to work over a syntax that is similar to that of C, whilst focusing on the implementation aspects that are most critical for security and efficiency. The memory model of CAO is extremely simple (there is no dynamic memory allocation, there are no side-effects in expressions, and it has call-by-value semantics). Furthermore, the language does not support any input/output constructions, as it is targeted at implementing the core components in cryptographic libraries. On the other hand, the native types and operators in the language are highly expressive and tuned to the specific domain of cryptography. These languages feature can then be used by the CAO compiler to provide domain-specific analysis and optimization.

Deductive program verification and Frama-C. Program Verification is the area of Formal Methods that aims to statically check software properties based on the axiomatic semantics of programming languages. In this paper we focus on techniques based on Hoare logic, brought to practice through the use of *contracts* – specifications consisting of preconditions and postconditions, annotated into the programs. Verification tools based on contracts are becoming increasingly popular, as their scope evolved from toy languages to realistic fragments of languages like C, C#, or Java. We use the expression *deductive verification* to distinguish this approach from other ways of checking properties of programs, such as *model checking*.

In this work we build on Frama-C [BFM$^+$08], an extensible framework where static analysis of C programs is provided by a series of plug-ins. Jessie [MM10] is a plug-in that can be used for deductive verification of C programs. Broadly speaking, Jessie performs the translation between an annotated C program and the input language for the Why tool. Why is a *Verification Condition Generator* (VCGen), which then produces a set of proof obligations that can be discharged using a multitude of proof tools that include the Coq proof assistant [Coq], and the Simplify [DNS05], Alt-ergo [CCK06], and Z3 [MB08] automatic theorem provers. The gwhy graphical front-end, allows monitoring individual verification conditions. This is particularly useful when used in combination with the possibility of exporting the conditions to various proof tools, allows users to first try discharging conditions with one or more automatic provers, leaving the harder conditions to be studied with the help of an interactive proof assistant.

Motivation. Experience shows [ABPV09, ABPV10] that a tool such as Frama-C has a great potential for verifying a wide variety of security-relevant properties in cryptographic software implementations. However, it is well-known that the intrinsic characteristics of the C language make it a hard target for formal verification, particularly when the goal is to maximize automation. This problem is amplified when the verification target is in the domain of cryptography, because implementations typically explore language constructions that are little used in other application areas, including bit-wise operations, unorthodox control-flow (loop unrolling, single-iteration loops, break statements, etc.), intensive use of macros, etc. The idea behind the construction of a deductive verification tool for CAO is to take advantage of the characteristics of this programming language to construct a domain-specific verification tool, allowing for the same generic verification techniques that can be applied over C implementations, simplifying the verification of security-relevant properties, and hopefully providing a higher degree of automation.

Contributions. In this paper we describe an implementation of such a deductive verification platform for the CAO language. We show that CAO presents interesting challenges for formal verification, concerning not only the rich mathematical type system that it introduces, but also the cryptography-oriented language constructions that it offers. We describe how we tackle these problems, namely by presenting what we believe is the first formalisation in first-order logic of the rich mathematical data types that are used in cryptography in the context of deductive verification. We also demonstrate that, by relying on the Jessie plug-in of the Frama-C framework, the development time of such a complex verification tool could be greatly reduced. We base our presentation on real-world examples of CAO code, extracted from the open-source code of the NaCl cryptography library (http://nacl.cr.yp.to). The development of our tool has so far focused on automating safety verification, which we have achieved for a representative set of examples.

Organisation of the Paper. The next section expands on the application scenario and functional requirements for our tool. Section 3 describes the high-level implementation choices we have made. In Section 4 we introduce the most relevant parts of the translations performed by the tool. The generation of safety proof obligations is discussed separately in Section 5. Section 6 discusses related work, and Section 7 has some concluding remarks.

2 Deductive verification of CAO programs

A detailed specification of CAO can be found in [Bar09]. Appendix A includes an example we will use throughout the paper: a partial CAO implementation of the AES block cipher. As a C-like language, CAO supports analogous definitions of conditionals and loops, as well as global variable declarations, function declarations and procedures. The syntax of expressions is also similar to that of C, although the set of types and operations are significantly different.

The CAO type system includes a set of primitive types: arbitrary precision integers int, bit strings of finite length bits[n] , rings of residue classes modulo an integer mod[n] (intuitively, arithmetic modulo an integer, or a finite field of order n if the modulus is prime) and boolean values bool. Derived types allow the programmer to define more complex abstractions. These include the product construction struct, the generic one-dimensional container vector[n] of T, the algebraic notion of matrix, denoted matrix[i,j] of T, and the construction of an extension to a finite field T using a polynomial p(X), denoted mod[T<X>/p(X)]. Algebraic operators are overloaded so that expressions can include integer, ring/finite-field and matrix operations; the natural comparison operators, extended bit-wise operators, boolean operators and a well-defined set of type conversion (cast) operators are also supported. Bit string, vector and matrix access operations are extended with range selection (also known as slicing operations).

An implementation of a type-checker for CAO programs has been derived from the CAO type system formalisation [Bar09]. Hence, for the purpose of this paper we will assume that the CAO verification tool has access to an Abstract Syntax Tree (AST) with complete type information for the input CAO program. Note that this includes the concrete sizes of all container types, the moduli and polynomials in rings and finite fields, etc. Furthermore, the CAO type checker is able to reject all programs where incompatible type parameters are passed to an operator. For example, the size restrictions associated with matrix addition and multiplication are enforced by the type system. The same happens for operations involving bit strings, rings and finite fields, where the type system checks that operator inputs have matching lengths, moduli, etc.

Safety in CAO. The verification that a program will not reach a point of execution that may result in a catastrophic failure, namely a run-time error, is commonly known as a *safety verification*. This type of verification goal is admittedly a modest one. Nevertheless, not only is safety verification a critical aspect for most practical applications, but also it is frequent that even this is a challenge for existing tools. In many cases, safety verification cannot be dealt with automatically, and it may become a labour-intensive activity. One of the requirements for the CAO verification tool is that safety verification should be feasible with minimum intervention from the end-user.

Program safety in CAO has two dimensions: memory safety and safety of arithmetic operations. A program is said to be memory safe if, at run-time, it never fails by accessing an invalid memory address. Memory safety verification is not in general a trivial problem in languages with pointers and heap-based data structures, and indeed there exist dedicated verification tools for this task. However, for CAO programs, this problem is reduced to making sure that all indices used in vector, bit string and matrix index accesses are within the proper range, which is fully determined by the type of the container and must be fully determined at compile time.

The safety of arithmetic operations is more interesting. In CAO we have four algebraic types: arbitrary precision integers, rings of residue classes modulo a non-prime, finite fields, and matrices thereof. The semantics of operators over these types is precisely given by the mathematical abstractions that they capture. This means that the concept of arithmetic overflow does not make sense in this context, and it leaves as candidate safety verification goals the possibility that such operators are not defined for some inputs. For integers, this reduces to the classic division-by-zero condition, whereas matrix addition and multiplication introduce are intrinsically safe.

Rings and finite fields pose an interesting problem, as they are not distinct CAO types. Take the following declarations:

```
def a : mod[13] := 4;            def b : mod[10] := 5;
def c : mod[13] := 1/a;          def d : mod[10] := 1/b;
```

All of these operations are safe, except for the initialization of d. The reason for this is that the multiplicative inverse modulo 10 is only defined for those integers in the range 1 to 9 that are co-prime with 10. This means that, whenever a division occurs in the mod[n] type, one must also ensure that the divisor is co-prime to the modulus.

When the modulus is a prime number, then the mod[n] type represents the finite field of size n. In this case, the previous problem reduces again to the division-by-zero case, as all non-zero elements have a multiplicative inverse. However, this observation does not help, unless there is a way to verify that the modulus is indeed a prime number. One way to do this, of course, is to allow the programmer to vouch for the primality of the modulus. We will return to this issue in Section 4. Finally, a related problem arises when one considers the construction of extension fields. In this case, not only must one ensure that the underlying type represents a finite field (which might not be the case for the mod[n] type) but also that the polynomial that is provided is irreducible in the corresponding ring of polynomials.

Extending CAO with annotations. CAO-SL is a specification language to be used in annotations added to CAO programs. These annotations are embedded in comments (so that they are ignored by the CAO compiler) using a special format that is recognised by the verification tool. CAO-SL is strongly inspired by ACSL [BFM+08] and enables the specification of behavioral properties of CAO programs. CAO-SL stands to CAO in the same way that ACSL stands to C.

The expressions used in annotations are called *logical expressions* and they correspond to CAO expressions with additional constructs. The semantics of the logical expressions is based on first-order logic. CAO-SL includes the definition of *function contracts* with pre- and postconditions, statement annotations such as assertions and loop variants and invariants, and other annotations commonly used in specification languages. CAO-SL also allows for the declaration of new logic types and functions, as well as predicates and lemmas. A complete description of CAO-SL can be found in [Bar09]. In this paper, various features of this language will be introduced gradually, as we describe the tool architecture and implementation.

3 Tool implementation

Our tool follows the same approach used in other scenarios for general-purpose languages such as Java [MPU04] and C [FM04]. Furthermore, the tool architecture itself fundamentally relies on the Jessie plug-in included in the Frama-C framework. This allowed us to significantly reduce the tool development time and effort. Jessie enables reasoning about typical imperative programs, and it is equipped with a first-order logic mechanism, which facilitates the design of new models and extensions. In particular, it is possible to use this feature to define in Jessie a model of the domain-specific types and memory model of CAO. This means that an annotated CAO program can be translated into an annotated Jessie program and, from this point on, our verification tool can rely totally on the functionality of Jessie and Why. The translation is such that correctness of the Jessie program entails the correctness of the source CAO program.

The Jessie Input Language. The Jessie input language is a simply typed imperative language with a precisely defined semantics. It is used as an intermediate language for verification of C programs in the Frama-C framework. As an intermediate language, programmers are not expected to produce Jessie source programs from scratch. Jessie was developed in parallel with ACSL, and they share many constructions. The language combines operational and logic features. The operational part refers to statements which describe the control flow and instructions that perform operations on data, including memory updates. The logical part is described through formulas of first-order logic, attached to statements and functions in the form of annotations. Jessie provides primitive types such as integers, booleans, reals and unit (the void type), abstract datatypes and also allows the definition of new datatypes.

Programs can be annotated using pre- and postconditions, loop invariants, and other intermediate assertions. The logical language is typed and includes built-in equality, booleans, arbitrary precision integers, and real numbers.

Implementation strategy. In order to better illustrate our approach to designing a VCGen for CAO taking advantage of an existing *generic* VCGen, we introduce a very simple example. Consider the definition of a VCGen for the subset of CAO that is essentially a *while* language with applicative arrays[1], and how one would deal with both *aliasing* and *safety*. The weakest precondition of the array assignment operation would resemble the following

$$\mathsf{wp}(u[e] := x, Q) \ = \ \mathit{safe}(u[e]) \wedge \mathit{unalias}(Q, e, x)$$

where $\mathit{safe}(u[e]) \ = \ \mathit{safe}(e) \ \wedge \ 0 \leq e < \mathit{length}(u)$, $\mathit{safe}(e)$ imposes that the evaluation of e will

[1] We use this denomination for typical imperative arrays with destructive update, and with opaque storage.

not produce arithmetic errors, and the function *unalias* would process Q, updating the contents of index e to x considering aliasing, e.g.

$$unalias(u[j] > 100, i, 10) \;=\; (i = j \implies 10 > 100) \land (i \neq j \implies u[j] > 100)$$

If one momentarily forgets safety considerations, an alternative possibility is to construct the VCGen on top of an existing VCGen for the base *while* language, not by adding new dedicated rules, but instead by translating the new type being introduced as a logical type. Applicative arrays can be modeled by the following well-known axioms for the *get* and *set* logical functions.

$$\forall u, e, x.\ get(set(u, e, x), e) = x \tag{1}$$

$$\forall u, e, e', x.\ e \neq e' \implies get(set(u, e, x), e') = get(u, e') \tag{2}$$

In the presence of this theory, one can use the VCGen for the core *while* language to derive verification conditions (VCs) for array lookup expressions and assignment commands, by simply translating these to use the definitions above:

$$\mathscr{T}(u[e]) \equiv get(u, \mathscr{T}(e)) \qquad\qquad \mathscr{T}(u[e] := x) \equiv u := set(u, \mathscr{T}(e), \mathscr{T}(x))$$

A powerful generic VCGen such as Jessie allows us to follow this approach for the entire CAO language. There is an overlap between the CAO language and the Jessie input language that enables a direct translation of many language constructions. Furthermore, for each CAO type that is not supported by Jessie, we are able to declare a set of logical functions and write a theory for them that creates a suitable first-order model of the type. This then enables us to translate arbitrary annotated CAO programs into suitable programs of the Jessie input language.

Let us now turn back to the example above, to see how we deal with safety conditions using Jessie's *assert* clause to force the generation of arbitrary proof obligations. Safety conditions for applicative arrays can be generated by using the following translation:

$$\begin{aligned} \mathscr{T}(u[e]) \quad &\equiv \quad \textbf{assert } 0 \leq \mathscr{T}(e) < length(u);\ get(u, \mathscr{T}(e)) \\ \mathscr{T}(u[e] := x) \quad &\equiv \quad \textbf{assert } 0 \leq \mathscr{T}(e) < length(u);\ u := set(u, \mathscr{T}(e), \mathscr{T}(x)) \end{aligned}$$

Of course, in CAO we have to deal with data types that are considerably more sophisticated than arrays. Yet, the general pattern followed in the implementation of our tool is the same. The introduction of each new type implies the introduction of a new theory. The definition of a new theory includes the definition of logic functions together with axioms to model their behavior. Some lemmas and predicates are also introduced to speed up the process of proving some goals.

At this point, it makes sense to ask which properties of those types should be included in the corresponding logical model. *Soundness* is of course the most important property that should be guaranteed by our translation process: the Jessie model should not allow proving assertions about CAO data structures that are not valid according to the language semantics. A second very desirable property is that the model should allow for as many assertions as possible to be proved automatically. More precisely, the verification conditions produced by Jessie, and exported to some external theorem prover, should as much as possible be discharged automatically.

Emphasis on Automation. The fact that Jessie relies on the Why VCGen, which is a *multi-prover* tool, means that it is possible to export verification conditions to a large number of different proof tools, from SMT-solvers to the Coq interactive proof assistant. The typical workflow is to first discharge “easy” VCs using an automatic prover, and then interactively trying to discharge the remaining conditions. Once the model of CAO is fixed, different properties of CAO code will naturally have different degrees of automation with respect to discharging VCs. As is true of VCGens for other realistic languages, safety conditions should be proved with a high degree of automation, whereas a lower degree should be expected for other functional properties. Our approach is multi-tiered in the sense that we start with high-level models tuned for automatic verification (in particular of safety properties); these models can then be refined into lower-level models that take advantage of theories supported by specific automatic provers (such as bit strings, integers, and so on). Finally, all models can be further refined to Coq models: interactive proof is the last resort for discharging VCs, and it may be mandatory for any verification tool based on first-order logic.

Our efforts have so far concentrated on maximizing the degree of automation that we can achieve in verifying the safety of CAO programs. We are able, for example, to carry out the safety verification of the entire CAO implementation of the AES block-cipher (see Appendix) without user intervention. This includes heavy use of finite field, vector and matrix operations across several dependent functions.

4 CAO to Jessie translation

We will resort to snippets of CAO code to describe the most interesting parts of the CAO to Jessie translation carried out by our verification tool, which essentially correspond to the rich cryptography-specific data types that are available in CAO. In other words, we will focus on the way in which we handle the parts of the CAO language (including the extension to CAO-SL) that do not directly map to constructions in the Jessie input language, leaving out the standard imperative constructions supported by both languages, the CAO types that directly map to Jessie native types, and the translation of annotations. In the following $\lceil e \rceil$ denotes the translation of a CAO expression e into Jessie.

4.1 Container Types

The container types in CAO include the vector[] of, matrix[] of and bits types. The get and set operations on these types are modeled in Jessie using exactly the second approach that we described in the example in the previous section. The only caveat is that they are generalized to the two dimensional case in the case of matrices, and that we fix Jessie type bool as the content type in the case of bits.

However, CAO includes elaborate operators to deal with these container types that are fine-tuned to the implementation of cryptographic algorithms, namely symmetric primitives such as block ciphers and hash functions. As an example, consider the next snippet from the AES implementation in CAO.

```
def ShiftRows( s : S ) : S {
  def r : S;
  seq i := 0 to 3 { r[i,0..3] := (Row)(((RowV)s[i,0..3]) |> i); }
  return r; }
```

$$\begin{array}{l}
\mathit{blit\_vector\_}\lceil\tau\rceil \ : \ \mathit{vector\_}\lceil\tau\rceil \rightarrow \ \mathit{vector\_}\lceil\tau\rceil \rightarrow \mathit{integer} \rightarrow \mathit{integer} \rightarrow \ \mathit{vector\_}\lceil\tau\rceil \\
\mathit{shift\_vector\_}\lceil\tau\rceil \ : \ \mathit{vector\_}\lceil\tau\rceil \rightarrow \mathit{integer} \rightarrow \mathit{vector\_}\lceil\tau\rceil
\end{array}$$

$$\begin{array}{l}
\forall v, \mathit{ofs}, i.\ \mathit{get\_vector\_}\lceil\tau\rceil(\mathit{shift\_vector\_}\lceil\tau\rceil(v,\mathit{ofs}),i) = \mathit{get\_vector\_}\lceil\tau\rceil(v,(\mathit{ofs}+i)) \\
\forall \mathit{src}, \mathit{dest}, \mathit{ofs}, \mathit{len}, i.\ \mathit{ofs} \leq i < (\mathit{ofs}+\mathit{len}) \implies \\
\qquad \mathit{get\_vector\_}\lceil\tau\rceil(\mathit{blit\_vector\_}\lceil\tau\rceil(\mathit{src},\mathit{dst},\mathit{ofs},\mathit{len}),i) = \mathit{get\_vector\_}\lceil\tau\rceil(\mathit{src}, i-\mathit{ofs}) \\
\forall \mathit{src}, \mathit{dest}, \mathit{ofs}, \mathit{len}, i.\ i < \mathit{ofs} \ \vee \ i \geq (\mathit{ofs}+\mathit{len}) \implies \\
\qquad \mathit{get\_vector\_}\lceil\tau\rceil(\mathit{blit\_vector\_}\lceil\tau\rceil(\mathit{src},\mathit{dst},\mathit{ofs},\mathit{len}),i) = \mathit{get\_vector\_}\lceil\tau\rceil(\mathit{dst}, i)
\end{array}$$

Figure 1: Declarations and axioms for vector types.

What we have here is a sequence of rotation (|>) operations applied to the i^{th} row of matrix s. The way in which this is expressed in CAO takes advantage of the range selection operator (..) that returns a value of the corresponding container type, with the same contents as the original one, but with appropriate dimensions. Here, this operator is used to select an entire row in the matrix, which is cast to the vector type in order to be rotated. The result is then cast back to the correct matrix type that can be assigned to the original row slice in matrix r.

Our first-order formalisation of container types deals with shift, rotate, range selection, range assignment and concatenation (@) operators in container types using a pattern that relies on two logic functions (shift and blit). We present the case of the vector type. The model assumes that a vector has infinite length, i.e., it has a start position, but it is represented as an unbounded (infinite length) memory block. The only exception to this rule is the extensional equality operator (==), where translation explicitly refers to the range of valid positions over which equality should hold. We emphasize that this part of the model deals only with the functionality of these operators: safety is handled separately by introducing appropriate assertions, as will be seen in Section 5.

Intuitively, the *shift* logic function takes as input a vector of arbitrary length, starting in position 0, and produces the sub-vector that starts at position i. The *blit* logic function involves two vectors, source s and destination d, an index i and a length parameter l. It produces the vector with the contents of d for indices 0 to i-1, and from i+l onwards; the l positions in between contain the region 0..l-1 of s. The behaviour of these logic functions is modeled by the declarations and axioms given in Figure 1.

Range Selection. Given a CAO variable μ of type $\mathit{vector}[n]\ \mathit{of}\ \tau$, the CAO range selection operation is modeled in Jessie as follows:

$$\begin{array}{l}
\lceil\ \mu[i..j]\ \rceil \equiv \textbf{let}\, x_1 = \lceil i \rceil\ \textbf{in}\ (\ \textbf{let}\, x_2 = \lceil j \rceil\ \textbf{in} \\
\qquad\qquad \textbf{assert}\ (0 \leq x_1 < n)\ \&\&\ (0 \leq x_2 < n)\ \&\&\ (x_1 \leq x_2);\ \mathit{shift}(\lceil\mu\rceil, x_1))
\end{array}$$

where i and j are integer expressions. We remark that although the translation disregards the upper bound j, the typechecker will ensure that the range selection operation $\mu[i..j]$ with μ of type $\mathit{vector}[n]\ \mathit{of}\ \tau$, returns type $\mathit{vector}[j-i+1]\ \mathit{of}\ \tau$, thus taking that upper bound into account.

Range assignment. Assigning to a region in a vector is modeled directly using the blit function.

$$\begin{aligned}\lceil \mu_1[i..j] := \mu_2 \rceil \ \equiv\ & \textbf{let}\ x_1 = \lceil i \rceil\ \textbf{in}\ (\ \textbf{let}\ x_2 = \lceil j \rceil\ \textbf{in} \\ & \textbf{assert}\ (0 \leq x_1 < n)\ \&\&\ (0 \leq x_2 < n)\ \&\&\ (x_1 \leq x_2); \\ & \lceil \mu_1 \rceil\ =\ blit(\lceil \mu_2 \rceil, \lceil \mu_1 \rceil, x_1, x_2 - x_1 + 1))\end{aligned}$$

Here, $=$ denotes assignment in Jessie.

Concatenation. Consider the CAO variables μ_1 and μ_2 of types $vector[n_1]\ of\ \tau$ and $vector[n_2]\ of\ \tau$ respectively. The concatenation of vectors μ_1 and μ_2 can also be captured using the blit function.

$$\lceil \mu_1\ @\ \mu_2\ \rceil \equiv blit(\lceil \mu_2 \rceil, \lceil \mu_1 \rceil, n_1, n_2)$$

The intuition behind this definition is that concatenation can be seen as a range assignment operation, where μ_2 is assigned to the region of μ_1 that starts at position n_1 (recall that in the model vectors are assumed to have infinite length).

Shift and Rotate. To present the shift and rotate operations in a more intuitive way, we will turn to the bits type. Both operations are modeled using the blit function.

The rotate operations are commonly known as circular shifts. A downwards circular shift by 1 is defined as a permutation of the entries in a tuple where the last element becomes the first element and all the other elements are down-shifted one position. Conversely, in an upwards circular shift, the first element becomes the last element and all the other are shifted up. As an example, consider the bits literal: 0b1101001. The internal representation of bits in our model stores the least significant bit (the right-most bit in the literal) in the 0-th position. This means that upwards and downwards rotate correspond to the intuitive interpretation of left and right rotations, respectively. An example of a down rotate is therefore $0b1101001 \mid> 3\ =\ 0b\underline{001}1101$ and an example of an up rotate is $0b1101001\ \ <\mid 3\ =\ 0b1001\underline{110}$. In our model, for a CAO expression e of type $vector[n]\ of\ \tau$ or $bits[n]$, we have:

$$\begin{aligned}\lceil e <\mid i \rceil &\equiv\ \lceil e[n-i\ ..\ n-1]\ @\ e[0\ ..\ n-i-1] \rceil \equiv\ blit(shift(\lceil e \rceil, 0), shift(\lceil e \rceil, n-i), i, n-i) \\ \lceil e \mid> i \rceil &\equiv\ \lceil e[i\ ..\ n-1]\ @\ e[0\ ..\ i-1] \rceil \equiv\ blit(shift(\lceil e \rceil, 0), shift(\lceil e \rceil, i), n-i, i)\end{aligned}$$

where i is a constant of type int. The intuition is that rotations can be seen as concatenations of the appropriate sub-regions, which in turn are modeled using the *blit* function.

Logical shifts are handled in a similar way, but resorting to $bits\_null\_vector$ (the all-zeroes bits value) to fill in the positions left vacant by the operation.

Matrices. Our model of matrices for the equivalent vector operators described above is essentially a direct generalization of the above strategy to the 2-dimensional case. However, our model of matrices must also account for the fact that the matrix type in CAO is an algebraic type that supports addition and multiplication operations (indeed this is why in CAO you can only define matrices whose contents are themselves algebraic types).

The formalisation of matrices in first-order logic includes the matrix addition and multiplication arithmetic operations as logic functions

$$matrix\_\lceil \tau \rceil\_add,\ matrix\_\lceil \tau \rceil\_mult\ \ :\ \ matrix\_\lceil \tau \rceil \rightarrow matrix\_\lceil \tau \rceil \rightarrow matrix\_\lceil \tau \rceil$$

The functionality of the addition operator is modeled using the following axiom:

Axiom 1 *Let A and B be matrices of dimensions $m \times n$, and a_{ij} and b_{ij} the elements in the i^{th} row and j^{th} column of A and B, respectively. Then, $\forall\ j,i.\ (A+B)_{ij} = a_{ij} + b_{ij}$.*

An equivalent axiom for matrix multiplication was not introduced because, for each possible base type, we would need the (higher-order) logic formalization of summation Σ.

The translation of expressions with arithmetic operations of type $matrix[n_1,n_2]\ of\ \tau$ is therefore the following:

$$\lceil\ \mu_1\ +\ \mu_2\ \rceil = matrix\_\lceil\tau\rceil\_add(\lceil\mu_1\rceil,\lceil\mu_2\rceil) \qquad \lceil\ \mu_1\ *\ \mu_2\ \rceil = matrix\_\lceil\tau\rceil\_mult(\lceil\mu_1\rceil,\lceil\mu_2\rceil).$$

Bitwise operations. We complete this section with a brief description of how bitwise operations are handled in our model, as these are of critical importance in cryptographic applications. Here we greatly benefit from the design of the CAO language, where the classic ambivalence between integers and their bit-level representations that exists in the C int type is eliminated by introducing the bits type. Indeed, CAO programmers can freely use bit strings of any size, and convert these to and from the type int that represents the mathematical type $\mathbb{Z}$. A very simple model of bit strings based on vectors of bits (boolean values) can be used, although things get more complicated when we need to deal with type conversions. The Jessie model of bitwise operations on bits is based on the following logic functions:

$$bits\_bitwise\_xor : bits \rightarrow bits \rightarrow bits \qquad bits\_bitwise\_and : bits \rightarrow bits \rightarrow bits$$
$$bits\_bitwise\_or : bits \rightarrow bits \rightarrow bits \qquad bits\_bitwise\_neg : bits \rightarrow bits$$

which are axiomatized in the obvious way. CAO bitwise operations are translated as:

$$\lceil e_1\ \oplus\ e_2\rceil \equiv bits\_bitwise\_\lceil\oplus\rceil(\lceil e_1\rceil,\lceil e_2\rceil) \qquad \lceil !\ e\rceil \equiv bits\_bitwise\_neg(\lceil e\rceil)$$

where $\oplus \in \{|,\&,\hat{}\}$ and μ_1 and μ_2 are expressions of type $bits[n]$.

4.2 Rings, fields and extension fields

Residue classes modulo n The mod[n] type is an algebraic type. For $n \in \mathbb{N}$, it corresponds to the algebraic ring $\mathbb{Z}_n$. Moreover if n is prime, then mod[n] permits programmers to take full advantage of the fact that $\mathbb{Z}_n$ is a field. The Jessie model for the mod[n] type is based on the theory of integers, taking advantage of optimized models supported by many automatic provers, and fully integrated into Jessie.

The model of mod[n] starts with the definition of the logic type mod_n equipped with logic functions corresponding to the two natural homomorphisms that convert to and from the Jessie integer type, as well as the mapping that results from their composition.

$$int\_of\_mod_n : mod_n \rightarrow integer \qquad mod_n\_of\_int : integer \rightarrow mod_n \qquad mod_n : integer \rightarrow integer$$

Hence, mod_n represents the mapping from $\mathbb{Z}$ to $\mathbb{Z}$ that associates to each $a \in \mathbb{Z}$ the least residue $r \in \mathbb{Z}$ of $[a]$. The model is then extended with a set of axioms for the following mathematical properties of these functions

$$\begin{array}{ll} \forall x.\ 0 \leq int\_of\_mod_n(x) \leq n-1 & \forall x.\ 0 \leq x \leq n-1 \implies mod_n(x) = x \\ \forall x.\ mod_n(int\_of\_mod_n(mod_n\_of\_int(x)) = mod_n(x) & \forall x.\ x \geq n \implies mod_n(x) = mod_n(x-n) \\ \forall x.\ x < 0 \implies mod_n(x) == mod_n(x+n) & \end{array}$$

The Jessie translation of arithmetic operations involving expressions of type mod[n] is based on the homomorphisms declared above. First, $int\_of\_mod_n$ is used to get the least residues of the equivalence classes involved in the arithmetic operation. These least residues are integers, which allows us to model the arithmetic operations using the integers theory. Finally, we apply $mod_n\_of\_int$ to the result to recover the equivalence class that represents that value. Hence, the translation of arithmetic operations on type mod[n] is given as follows for $op \in \{+,-,*,**\}$.

$$\begin{array}{rl} \lceil e_1\ op\ e_2 \rceil \equiv & mod_n\_of\_int(int\_of\_mod_n(\lceil e_1 \rceil)\ \ op_{int}\ \ int\_of\_mod_n(\lceil e_2 \rceil)) \\ \lceil e_1\ /\ e_2 \rceil \equiv & \textbf{let}\ x = int\_of\_mod_n(\lceil e_2 \rceil)\ \textbf{in} \\ & \quad \textbf{assert}\ gcd(x,n) = 1;\ mod_n\_of\_int(int\_of\_mod_n(\lceil e_1 \rceil) *_{int} inv\_mod(x,n)) \end{array}$$

Note the special case of division. This is justified because the semantics of division modulo n is not the same as integer division. Firstly, one must express the correct semantics, which we do by introducing the logical function $inv\_mod(x,n)$. Simple properties involving operations with this function, which are used to easily discharge some proof obligations, are axiomatized as follows:

$$\forall x.\ gcd(int\_of\_mod_n(x)),n) = 1 \implies mod_n(int\_of\_mod_n(x) *_{int} inv\_mod(int\_of\_mod_n(x),n)) = mod_n(1)$$

$$\forall x,y.\ mod_n(int\_of\_mod_n(x) *_{int} y) = mod_n(1) \implies inv\_mod(int\_of\_mod_n(x),n) = mod_n(y)$$

Secondly, in the division case, one must generate a proof obligation for the safety condition that CAO programs should not perform undefined divisions. This property is trivially true if the divisor is in the range $1 \ldots n$-1 and the number n is prime. Hence we add the following axiom to our model, to automatically handle these trivial cases.

$$\forall x,n.\quad is\_prime(n)\ \wedge\ (0 < x < n) \implies gcd(x,n) = 1$$

where $is\_prime : integer \rightarrow boolean$ is a predicate to check if an integer number is prime, and $gcd : integer \rightarrow integer \rightarrow integer$ is a logic function that calculates the greatest common divisor between two integer numbers. Note that is_prime and gcd are neither directly defined nor axiomatized, but the programmer can explicitly assert that some n is prime through a CAO-SL annotation. This enables automatically discharging safety assertions using gcd.

Extension Fields. Consider the following type declarations taken from the AES implementation:

```
typedef GF2  := mod[2];
typedef GF2N := mod[GF2<X> / X**8+X**4+X**3+X+1];
typedef GF2C := mod[GF2N<Y> / Y**4+1];
```

Take the first field extension type GF2N. Types of the form mod[mod[n] <X>/p(X)] are also algebraic types that model the Galois field of order n^d where n is a prime number and d is the degree of the irreducible polynomial $p(X)$. We emphasize that in CAO each such type represents a specific construction of an extension field, whose representation is fixed as elements of the polynomial ring $\mathbb{Z}_n[X]$, and the semantics of operations is defined based on polynomial arithmetics modulo $p(X)$. Furthermore this type is only valid when n is prime and $p(X)$ is irreducible.

The theory of extension fields of this form begins with the definition of a logic type $ring\_mod_n$ that represents the ring of polynomials over the base type mod[n] and logic functions to model the elements of the ring and the addition operation that permits combining them.

$$\begin{array}{r}
ring\_mod_n\_monomial : mod_n \rightarrow integer \rightarrow ring\_mod_n \\
ring\_mod_n\_add : ring\_mod_n \rightarrow ring\_mod_n \rightarrow ring\_mod_n
\end{array}$$

Our model explicitly captures the fact that elements of this ring are polynomials, which in turn can be defined as an addition of monomials. The reason for this is that the CAO literal that corresponds to the irreducible polynomial $field\_mod_n\_poly_{f(x)}\_generator$ used to construct these types can then be represented in our logical model. A monomial can be represented by its coefficient (which is an element of mod_n) and its degree (an integer). Arithmetic operations over the polynomial ring are not included in the model, as they do not exist in CAO. Indeed our model is purposefully incomplete because we do not intend to use automatic theorem provers on verification conditions involving arbitrary extension field algebra. The goal is to use specific interactive proof assistants, namely Coq, to prove these kinds of properties, relying on existing libraries (e.g. SSReflect[2]) that provide theories for abstract algebra (fields, polynomials, etc).

The model is then completed by adding definitions for the type $field\_mod_n\_poly_{f(x)}$ and the corresponding arithmetic operations. The Jessie translation of the arithmetic operations defined over mod[mod[n] <X>/p(X)] is then a direct one:

$$\begin{array}{rl}
\lceil e_1 \; op \; e_2 \rceil & \equiv \lceil e_1 \rceil \; op_{field\_mod_n\_poly_{f(x)}} \; \lceil e_2 \rceil \\
\lceil e_1 \; / \; e_2 \rceil & \equiv \textbf{let } x = \lceil e_2 \rceil \textbf{ in assert } x \neq 0_{field\_mod_n\_poly_{f(X)}} ; \; \lceil e_1 \rceil \; div_{field\_mod_n\_poly_{f(X)}} \; x
\end{array}$$

where $op \in \{+, -, *, **\}$. Note that there is also a special case for division. This ensures that a safety proof obligation is generated that checks if the divisor is different from zero.

A set of axioms that describe basic properties of these operators has been added to the model in order to increase the degree of automation provided by our tool. The goal here is that, given that there is no integrated support for this sort of mathematical construction in the automatic provers interfaced with Jessie, some simple properties can be captured in first order logic that permit dealing with trivial deduction steps, e.g. cancellation rules. The following axioms are included in our model

$$\begin{array}{cc}
\forall a,b.\; a \neq 0_F \;\wedge\; b \neq 0_F \;\Longrightarrow\; a \times_F \; b \neq 0_F & \forall a,b.\; a \neq 0_F \;\Longrightarrow\; a \; div_F \; b \neq 0_F \\
\forall a,b.\; a \neq b \;\Longrightarrow\; a \; -_F \; b \neq 0_F & \forall a,b.\; a \neq -b \;\Longrightarrow\; a \; +_F \; b \neq 0_F \\
\forall a,b.\; a \neq 0_F \;\Longrightarrow\; a \; (**)_F \; b \neq 0_F & \forall a.\; a \neq 0_F \;\Longrightarrow\; -_F a \neq 0_F
\end{array}$$

where $F = field\_mod_n\_poly_{f(X)}$. Literals of the extension field types are modeled in Jessie as vectors of polynomial coefficients. Therefore, logic functions to access and update the coefficient of a given power of some polynomial of type mod[mod[n] <X>/p(X)] are also included in the model, together with the usual two axioms for the theory of arrays.

$$field\_mod_n\_poly_{f(x)}\_get\_coef : field\_mod_n\_poly_{f(x)} \rightarrow integer \rightarrow mod_n$$

$$field\_mod_n\_poly_{f(x)}\_set\_coef : field\_mod_n\_poly_{f(x)} \rightarrow integer \rightarrow mod_n \rightarrow field\_mod_n\_poly_{f(x)}$$

[2] http://www.msr-inria.inria.fr/Projects/math-components

$$\begin{array}{rcl@{\qquad}rcl}
bits[n] & \Rightarrow & int & mod[\tau < X > / f(X)] & \rightarrow & vector[n]\ of\ \tau \\
mod[n] & \rightarrow & int & matrix[1,n]\ of\ \tau & \rightarrow & vector[n]\ of\ \tau \\
int & \rightarrow & bits[n] & matrix[n,1]\ of\ \tau & \rightarrow & vector[n]\ of\ \tau \\
\tau & \Rightarrow & mod[\tau < X > / f(X)] & vector[n]\ of\ \tau & \rightarrow & matrix[1,n]\ of\ \tau \\
vector[n]\ of\ \tau & \rightarrow & mod[\tau < X > / f(X)] & vector[n]\ of\ \tau & \rightarrow & matrix[n,1]\ of\ \tau
\end{array}$$

Figure 2: Casts and coercions

Returning to the example introduced above, it can be seen by examining the type declaration of GF2C that the base type of an extension field can actually be an extension field itself. However, our modeling approach is exactly the same for this case, taking into consideration that the base type must be adjusted when defining the ring of polynomials over the base field.

4.3 Casts and Coercions

Type conversion operations in CAO can be explicit, in which case they are called *cast* operations, or implicit, called *coercion* operations. Figure 2 presents the allowed cast (→) and coercion (⇒) operations between CAO types. The translation of CAO programs into Jessie handles these conversions in the natural way by using appropriate logical functions. We present a few examples of the simpler conversions:

$$\begin{array}{l}
e :: mod[n] \implies \lceil (int)\ e \rceil \ = \ int\_of\_mod_n(\lceil e \rceil) \\
e :: int \implies \lceil (mod[n])\ e \rceil \ = \ mod_n\_of\_int(\lceil e \rceil) \\
e :: int \implies \lceil (bits[n])\ e \rceil \ = \ bits\_of\_int(\lceil e \rceil) \\
e :: \tau \implies \lceil (mod[\tau{<}X{>}/f(X)])\ e \rceil = \ field\_\lceil \tau \rceil\_poly_{f(x)}\_set\_coef(field\_\lceil \tau \rceil\_poly_{f(x)}\_zero, 0, \lceil e \rceil_{\tau})
\end{array}$$

Conversions between matrices and column/row vectors are handled in the natural way by using get and set operations. Finally, we present the conversion between extension field types and vector types in a bit more detail, since these are very useful CAO operators that permit commuting between the abstract algebraic view of a finite field, and its concrete representation in a cryptographic implementation. Indeed, one can construct an extension field value from a vector representation that contains the coefficients of the corresponding polynomial over the base field. We model this as

$$\begin{array}{l}
\lceil (mod[\tau < X > / f(X)])\ e \rceil \ = \\
\quad \mathbf{let}\ x_1 \ = field\_\lceil \tau \rceil\_poly_{f(x)}\_zero\ \mathbf{in}\ (\ \mathbf{let}\ x_2 \ = \lceil e \rceil\ \mathbf{in} \\
\quad \mathbf{let}\ x_3 \ = field\_\lceil \tau \rceil\_poly_{f(x)}\_set\_coef(x_2, n-1, vector\_\lceil \tau \rceil\_get(x_2, n-1))\ \mathbf{in}\ \ldots \\
\quad \mathbf{let}\ x_{n+2} \ = field\_\lceil \tau \rceil\_poly_{f(x)}\_set\_coef(x_{n+1}, 0, vector\_\lceil \tau \rceil\_get(x_2, 0))\ \mathbf{in}\ x_{n+2})
\end{array}$$

The inverse conversion is also possible, and is modeled using a similar approach. This translation further justifies our modeling of extension field literals presented in the previous section.

5 Automatic safety proof obligations

Following the same approach adopted in tools such as Frama-C, the CAO to Jessie translation in our tool ensures that all statements in the input program that could potentially result in a safety

Table 1: Safety proof obligations

Type	Operation	Proof Obligation	Auto
$integer$	p_1/p_2	$p_2 \neq 0$	×
$mod[n]$	p_1/p_2	$gcd(int\_of\_mod_n(p_2),n) = 1;\ int\_of\_mod_n(p_2) \neq 0$	×
$mod[n] < X > / f(X)$	$p_1\ /\ p_2$	$p2 \neq 0$	
$vector[n]\ of\ \tau$	$v[e]$ $v \mid\!\rhd i,\ v \lhd\!\mid i$ $v[i..j]$	$0 \leq \lceil e \rceil < n$ $0 \leq \lceil i \rceil < n$ $0 \leq \lceil i \rceil < n \wedge 0 \leq \lceil j \rceil < n \wedge \lceil i \rceil < \lceil j \rceil$	
$matrix[n_1,n_2]\ of\ \tau$	$m[e_1,e_2]$ $m[i..j,k..l]$	$0 \leq \lceil e_1 \rceil < n_1\ \wedge\ 0 \leq \lceil e_2 \rceil < n_2$ $0 \leq \lceil i \rceil < n_1\ \wedge\ 0 \leq \lceil j \rceil < n_1\ \wedge\ 0 \leq \lceil k \rceil < n_2\ \wedge$ $0 \leq \lceil l \rceil < n_2\ \wedge\ \lceil i \rceil < \lceil j \rceil\ \wedge\ \lceil k \rceil < \lceil l \rceil$	
$bits[n]$	$b[e]$ $b \mid\!\rhd i, b \lhd\!\mid i, b \gg i, b \ll i$	$0 \leq \lceil e \rceil < n$ $0 \leq \lceil i \rceil < n$	

violation originate the automatic generation of a verification condition that, if proven, guarantees the safe execution of the verified code.

We have two classes of safety proof obligations: those related with memory safety, and those related with algebraic type declarations and operations. Some of the proof obligations are automatically generated by the Jessie tool, while others are explicitly introduced in the generated Jessie code as assertions, during the translation process. We have encountered examples of these assertions in the models for division operations presented above. Table 1 presents the proof obligations that are generated to ensure the safety of memory access and algebraic operations. Proof obligations automatically generated by the Jessie plug-in are signaled in the table, corresponding to those that originate from the use of the Jessie integer type.

The safety proof obligations that are generated when types are declared are limited to the declaration of extension fields of the form mod[mod[n] <X>/p(X)]. In this case, the proof depends on the two following generated lemmas.

$$is\_prime(n) \qquad\qquad ring\_mod_n\_is\_irreducible(field\_mod_n\_poly_{f(x)}\_generator)$$

These are required to allow the automatic discharge of some proof obligations, but they also ensure that the user is aware of the type declarations and their implications. Lemmas can be immediately used in proofs, so for instance the first lemma can be used as an hypothesis in all proof obligations related to division operations in mod[n], requiring that the divisor is relative prime to the modulus. We emphasise however that the presence of lemmas also originates new proof obligations corresponding to the validation of the lemmas themselves.

6 Related work

The verification infrastructure introduced in the Jessie plug-in was already used in the development of other verification tools for C and Java. Caduceus [FM04], a tool for C, and Krakatoa [MPU04], a tool for Java, are also built on the top of Why tool. The translation into Why performed by Krakatoa is similar to that performed by Frama-C and also adopted in this paper.

Boogie [BED$^+$06] is a verification condition generator very similar to Why. The input languages to Boogie and Why are both languages with imperative features and first-order propositions. In both cases, verification condition generation is based on the weakest precondition calculus. Boogie has front-ends for extensions of C# and C which enrich the languages with annotations in first-order logic, such as pre- and postconditions, assertions and loop invariants.

The C# extension is known as Spec# [BMF$^+$]. Boogie performs loop-invariant inference using abstract interpretation and then generates the verification conditions for Simplify or Z3. The core component of VCC [CDH$^+$09], a tool for low-level concurrent C programs, is also Boogie.

Esc/Java [FLL$^+$02] is another deductive verification tool for Java programs whose annotation language is a subset of JML [LRL$^+$00]. Its architecture is similar to the ones presented above and based on an earlier checker for the Modula-3 language. This tool relies on loop unrolling, and the fact that generation of verification conditions includes optimizations to avoid the exponential blow-up inherent in a naive weakest-precondition computation. It looks for run-time errors in annotated Java programs, but does not model arithmetic overflow.

Jack (Java Applet Correctness Kit) [BBC$^+$07] is a static verification tool for JML-annotated programs. It provides support for annotation generation and for interactive verification of functional specifications, as well as support for automatic verification of common security policies and for verification of byte-code programs. Its integration in the Eclipse IDE allows for proof obligation inspection, allowing users to visualize where in the code they are originated.

7 Conclusions

We have presented a model in first-order logic of certain mathematical objects, taking advantage of the theories implemented in general Satisfiability Modulo Theories (SMT) solvers. The objects that we model have specific interest for cryptographic security and for the verification of CAO programs in particular, but we believe that the model has independent interest and can be of use in other areas, whenever formal verification involving these objects is important.

Admittedly, the fact that the tool has not been proved correct is a flaw. Note however that this work is part of a bigger effort on the formalization of the CAO programming language. In related work we are working on an operational semantics for CAO, which we will later use to establish a correctness result for our VCGen. We remark however that the reliability of the VCGen is already high, since we rely on Jessie to capture the semantics of the basic language aspects of CAO, such as control structures and mutually recursive, contract-annotated procedures.

References

[ABPV09] J. B. Almeida, M. Barbosa, J. S. Pinto, B. Vieira. Verifying Cryptographic Software Correctness with Respect to Reference Implementations. In *Formal Methods for Industrial Critical Systems (FMICS)*. LNCS 5825, pp. 37–52. Springer, 2009.

[ABPV10] J. B. Almeida, M. Barbosa, J. S. Pinto, B. Vieira. Deductive Verification of Cryptographic Software. *To appear NASA Journal of Innovations in Systems and Software Engineering*, 2010.

[Bar09] M. Barbosa. CACE Deliverable D5.2: Formal Specification Language Definitions and Security Policy Extensions. 2009. Available from http://www.cace-project.eu.

[BBC$^+$07] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, A. Requet. JACK: a tool for validation of security and behaviour of Java applications. In *International Symposium on Formal Methods for Components and Objects (FMCO)*. LNCS. Springer-Verlag, 2007.

[BED$^+$06] M. Barnett, B. yuh Evan Chang, R. Deline, B. Jacobs, K. R. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO 2005)*. LNCS 4111, pp. 364–387. Springer-Verlag, 2006.

[BFM$^+$08] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto. ACSL: ANSI/ISO C Specfication Language. CEA LIST and INRIA, 2008.

[BMF$^+$] M. Barnett, P. Müller, M. Fähndrich, W. Schulte, K. Rustan, M. Leino, H. Venter. Specification and Verification: The Spec # Experience.

[CCK06] S. Conchon, E. Contejean, J. Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories. 2006. http://ergo.lri.fr/papers/ergo.ps.

[CDH$^+$09] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, S. Tobies. VCC: A practical system for verifying concurrent C. In *Conf. Theorem Proving in Higher Order Logics*. LNCS 5674. Springer, 2009.

[DNS05] D. Detlefs, G. Nelson, J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM* 52(3):365–473, 2005.

[FLL$^+$02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended static checking for Java. In *ACM SIGPLAN Conference on Programming language design and implementation (PLDI'02)*. Pp. 234–245. ACM, 2002.

[FM04] J.-C. Filliâtre, C. Marché. Multi-prover Verification of C Programs. In Davies et al. (eds.), *ICFEM*. LNCS 3308, pp. 15–29. Springer, 2004.

[LRL$^+$00] G. T. Leavens, C. Ruby, K. R. M. Leino, E. Poll, B. Jacobs. JML: notations and tools supporting detailed design in Java. In *Proceedings of OOPSLA '00 (Poster session addendum)*. Pp. 105–106. ACM, New York, NY, USA, 2000.

[MB08] L. de Moura, N. Bjørner. *Z3: An Efficient SMT Solver*. LNCS 4963/2008, pp. 337–340. Springer Berlin, April 2008.

[MM10] C. Marché, Y. Moy. Jessie Plugin Tutorial. INRIA, 2010.

[MPU04] C. Marché, C. Paulin-mohring, X. Urbain. The Krakatoa Tool for Certication of Java/JavaCard Programs annotated in JML. 2004.

[Coq] The Coq Development Team. The Coq Proof Assistant Reference Manual – Version V8.2. 2008. http://coq.inria.fr.

A CAO implementation of AES

```
typedef GF2  := mod[ 2 ];
typedef GF2N := mod[ GF2<X>/X**8 + X**4 + X**3 + X + 1];
typedef GF2V := vector[8] of GF2;
typedef S,K  := matrix[4,4] of GF2N;
typedef Row  := matrix[1,4] of GF2N;
typedef Col  := matrix[4,1] of GF2N;
typedef RowV,ColV := vector[4] of GF2N;

def M : matrix[8,8] of GF2 := { ... };
def C : vector[8]   of GF2 := { ... };
def mix : matrix[4,4] of GF2N := { ... };

def SBox( e : GF2N ) : GF2N {
  def x : GF2N;
  if (e == [0]) { x := [0]; } else {   x := [1] / e; }
  def A : matrix[8,1] of GF2:=(matrix[8,1] of GF2)(GF2V)x;
  def B : GF2V := (GF2V)(M*A);
  return ((GF2N)B) + ((GF2N)C);
}
def SubBytes( s : S ) : S {
  def r : S;
  seq i := 0 to 3
    seq j := 0 to 3 {r[i,j] := SBox( s[i,j] );}
  return r;
}
def SubWord( w : vector[4] of GF2N ) :
  vector[4] of GF2N {
  def r : vector[4] of GF2N;
  seq i := 0 to 3 { r[i] := SBox( w[i] ); }
  return r;
}
def ShiftRows( s : S ) : S
{
  def r : S;
  seq i:= 0 to 3 {r[i,0..3]:=(Row)(((RowV)s[i,0..3])|>i);}
  return r;
}
def MixColumns( s : S ) : S
{
  def r : S;
  seq i := 0 to 3 { r[0..3,i] := mix * s[0..3,i]; }
  return r;
}
def AddRoundKey( s : S, k : K ) : S
{
  def r : S;
  seq i := 0 to 3
    seq j := 0 to 3 { r[i,j] := s[i,j] + k[i,j]; }
  return r;
}
def FullRound( s : S, k : K ) : S
{
    return MixColumns( ShiftRows( SubBytes(s) ) ) + k;
}
def Aes( s : S, keys : vector[11] of K) : S
{
  seq i := 1 to 9 { s := FullRound( s,keys[i] ); }
  return ShiftRows( SubBytes(s) ) + keys[10];
}
```

Clang and Coccinelle: Synergising program analysis tools for CERT C Secure Coding Standard certification

Mads Chr. Olesen[1], René Rydhof Hansen[1], Julia L. Lawall[2], Nicolas Palix[2]

[1]rrh,mchro@cs.aau.dk, http://www.cs.aau.dk
[2]julia,npalix@diku.dk, http://www.diku.dk

Abstract: Writing correct C programs is well-known to be hard, not least due to the many language features intrinsic to C. Writing secure C programs is even harder and, at times, seemingly impossible. To improve on this situation the US CERT has developed and published a set of coding standards, the "CERT C Secure Coding Standard", that (in the current version) enumerates 118 rules and 182 recommendations with the aim of making C programs (more) secure. The large number of rules and recommendations makes automated tool support essential for certifying that a given system is in compliance with the standard.

In this paper we report on ongoing work on integrating two state of the art analysis tools, Clang and Coccinelle, into a combined tool well suited for analysing and certifying C programs according to, e.g., the CERT C Secure Coding standard or the MISRA (the Motor Industry Software Reliability Assocation) C standard. We further argue that such a tool must be highly adaptable and customisable to each software project as well as to the certification rules required by a given standard.

Clang is the C frontend for the LLVM compiler/virtual machine project which includes a comprehensive set of static analyses and code checkers. Coccinelle is a program transformation tool and bug-finder developed originally for the Linux kernel, but has been successfully used to find bugs in other Open Source projects such as WINE and OpenSSL.

Keywords: automated tool support, CERT C Secure Coding, certification, Clang, Coccinelle

1 Introduction

Writing correct C programs is well-known to be hard. This is, in large part, due to the many programming pitfalls inherent in the C language and compilers, such as low-level pointer semantics, a very forgiving type system and few, if any, run time checks. Writing a *secure* C program is even more difficult, as witnessed by the proliferation of published security vulnerabilities in C programs: even seemingly insignificant or "small" bugs may lead to a complete compromise of security.

In an effort to improve the quality of security critical C programs, the US CERT[1] organisation is maintaining and developing a set of rules and recommendations, called the *CERT C Secure*

[1] Formerly known as the US Computer Emergency Response Team (www.cert.org)

Coding Standard (CCSCS), that programmers should observe and implement in C programs in order to ensure at least a minimal level of security. The current version of the CCSCS enumerates 118 rules and 182 recommendations covering topics ranging from proper use of C preprocessor directives and array handling to memory management, error handling and concurrency. The sheer number of rules and recommendations makes it almost impossible for a human programmer to manually guarantee, or even check, compliance with the full standard. Automated tool support for compliance checking is therefore essential.

In this paper we describe work in progress on a prototype tool for automated CCSCS compliance checking. The tool is based on the open source program analysis and program transformation tool *Coccinelle* that has been successfully used to find bugs in the Linux kernel, the OpenSSL cryptographic library [LBP$^+$09, LLH$^+$10, PLM10], and other open source infrastructure software. Coccinelle is scriptable using a combination of a domain specific language, called SmPL for *Semantic Patch Language*, as well as in O'Caml and Python. The scripts specify search patterns partly based on syntax and partly on the control flow of a program. This makes Coccinelle easily adaptable to new classes of errors and new codebases with distinct API usage and code style requirements. Coccinelle does not perform program analysis in the traditional sense, e.g., data flow analysis or range analysis. However, for the purposes of program certification and compliance checking such analyses are essential, both to ensure soundness of the certification and to improve precision of the tool. For this reason we integrate the *Clang Static Analyzer* with Coccinelle in order to enable Coccinelle to use the analysis (and other) information found by Clang.

The Clang Static Analyzer is part of the C frontend for the LLVM project[2]. In addition to classic compiler support, it also provides general support for program analysis, using the monotone framework, and a framework for checking source code for (security) bugs. The emphasis in the source code checkers of the Clang project is on minimising false positives (reporting "errors" that are not really errors) and thus is likely to miss some real error cases. To further enhance the program analysis capabilities of Clang, in particular for inter-procedural program analyses, we have integrated a library, called WALi [3] for program analysis using weighted push-down systems (WPDS)[RSJM05] into Clang. To enable rapid prototyping and development of new or specialised analyses, we have implemented Python bindings for the WALi library.

The rest of the paper is organised as follows. In Section 2 we give an overview of the CERT C Secure Coding Standard including a brief description of the rule categories. Section 3 illustrates how a few of the coding rules can be automatically checked using the Coccinelle tool. Section 4 describes how the Coccinelle rules can benefit from having access to program analysis information. Section 5 discusses current work in progress, including experiments and the integration of Clang and Coccinelle. Finally Section 7 concludes.

2 The CERT C Secure Coding Standard

The CERT C Secure Coding Standard (CCSCS) is a collection of rules and recommendations for developing secure C programs. One version of the CCSCS was published in 2008 as [Sea08].

[2] Web: http://clang.llvm.org

[3] Web: http://www.cs.wisc.edu/wpis/wpds/

Code	Short name	Long name	# of Rules	# of Recomm.
01	PRE	Preprocessor	4	12
02	DCL	Declarations and Initialization	9	22
03	EXP	Expressions	11	21
04	INT	Integers	7	18
05	FLT	Floating Point	8	6
06	ARR	Arrays	8	3
07	STR	Characters and Strings	9	12
08	MEM	Memory Management	6	13
09	FIO	Input Output	15	20
10	ENV	Environment	3	5
11	SIG	Signals	6	3
12	ERR	Error Handling	4	8
13	API	Application Programming Interfaces	N/A	10
14	CON	Concurrency	6	2
49	MSC	Miscellaneous	10	23
50	POS	POSIX	12	4

Figure 1: Categories in the CERT C Secure Coding Standard

However, in this paper we focus on the version currently being developed. The development process is collaborative through the CCSCS' web site[4]. The current version[5] of the CCSCS consists of 118 rules and 182 recommendations. The rules and recommendations are divided into 16 categories covering the core aspects of the C programming language. Figure 1 shows an overview of these categories and a summary of the number of rules and recommendations in each category.

2.1 Overview of the CCSCS

Experience shows that when programming in C, certain programming practises and language features, e.g., language features with unspecified (or compiler dependent) behaviour result in insecure programs or, at the very least, in programs that are hard to understand and check for vulnerabilities. This experience is at the heart of the CCSCS. Many of the observed problems arise when programmers rely on a specific compiler's interpretation of behaviour that is undefined in the ANSI standard for the C programming language (ANSI C99). Other problems are caused, or at least facilitated, by the flexibility of the C language and the almost complete lack of run-time checks.

Based on the observed problems, the US CERT has identified a number of key issues and developed a set of *rules* that specify both how to avoid problematic features and also constructively how to use potentially dangerous constructs in a secure way, e.g., programming patterns for securely handling dynamic allocation and de-allocation of memory. The rules in the CCSCS are

[4] Web: https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Secure+Coding+Standard

[5] Last checked 6 July 2010

almost all unambiguous, universal and generally applicable in the sense that they do not depend on the specific application being developed. Furthermore the rules are, for the most part, formulated at the level of individual source files or even parts of source files and thus require little or no knowledge of the surrounding application or the context in which it is used. This makes the rules potentially ideally suited for automated checking, although see Section 3 for a more detailed discussion of this.

In addition to the above mentioned rules, the CCSCS also contains an even larger number of *recommendations*. The recommendations often represent the *best practise* for programming secure systems. In contrast to the rules, the recommendations are not limited to constructs that are local to a single file or function, but may also cover more global issues such as how to handle sensitive information, how to use and implement APIs, how to declare and access arrays, and so forth. While most of the recommendations are still amenable to automated analysis, it may take more work and, in particular, it will require configuring and specialising the automated tool to the specific project being checked, e.g., by specifying which data in the program may contain sensitive information or which macros that are considered safe or how to canonicalize file names. A programmer is not required to follow the recommendations in order to be compliant with the CCSCS.

The CCSCS is much too large to cover in detail here, instead we give a brief overview of the different categories and the kind of (potential) errors they are designed to catch. In Section 3 we focus on a few specific rules and discuss them in more detail.

2.2 Categories of the CCSCS

Preprocessor (01-PRE). The rules and recommendations in this category are concerned with proper use of the C preprocessor. Most (large) C projects use preprocessor directives, especially macro definitions, extensively. Since these can dramatically change the “look” of a program, it is very important at least to avoid the many common pitfalls enumerated in this category.

Many static analysis tools are not very good at checking these rules since they typically work on the expanded code and thus do not even see the macros. This is unfortunate since a lot semantic information can be gleaned from well-designed macros and their use.

Declarations and Initialization (02-DCL). The rules and recommendations in this category mostly cover tricky semantics of the type system and variable declarations such as implicit types, scopes, and conflicting linkage classifications.

The recommendations in this category codify good programming practises, e.g., using visually distinct identifiers (DCL02-C) and using typedefs to improve code readability (DCL05-C). While many of the recommendations can be automatically verified while others (like DCL05-C) require human interaction.

Expressions (03-EXP). The rules and recommendations in this category are concerned with issues related to expressions, including (unspecified) evaluation orders, type conversions, sizes of data types, general use of pointers, and so forth.

Below we show how rule EXP34-C (do not dereference null pointers) can be checked using the Coccinelle tool.

Integers (04-INT). The rules and recommendations in this category are concerned with issues related to proper handling of integers. The main emphasis for the rules is on avoiding overflows and wrap-around for very large or very small integer values. Automated checking for these rules can be difficult since that may require sophisticated data flow or interval analysis. Alternatively, a tool can instead check that a program includes sufficient checking in the program itself to avoid the dangerous situations. In some cases it is possible to use Coccinelle to automatically insert the proper checks. However, inserting such checks automatically would seem to violate the point of a security certification.

The recommendations are similarly concerned with conversions, limits and sizes of the integer types. Like the rules in this category, automated checking of the recommendations can be difficult and require sophisticated analysis.

Floating Point (05-FLP). the rules and recommendations in this category are concerned with issues relating to proper handling of floating point types: loss of precision, proper use of mathematical functions, and type conversion. Automated checking is at least as difficult as for the integer case.

Arrays (06-ARR). The rules and recommendations in this category focus on avoiding out of bounds array indexing and pointer access to arrays. Automated checking is likely to require pointer analysis in order to ensure correctness and to minimise false positives.

Characters and Strings (07-STR). The rules and recommendations in this category are concerned with: ensuring that strings are null terminated, proper size calculation of strings, and bounds checking for strings.

Memory Management (08-MEM). The rules and recommendations in this category cover some of the many pitfalls surrounding dynamic memory allocation, including not accessing freed memory, do not "double free" memory, only freeing dynamically allocated memory and so forth. Implementing memory management correctly is notoriously difficult and even small bugs in this category are likely to result in a security vulnerability, e.g., a buffer overflow or a null pointer dereference. Below we discuss rule MEM30-C (do not access freed memory) in more detail and show how it can be checked using Coccinelle.

Input Output (09-FIO). The rules and recommendations in this category are mainly concerned with the proper use of library functions for (file) input and output, including proper opening and closing of files, creation of temporary files, as well as secure creation of format strings.

Environment (10-ENV). The rules and recommendations in this category are concerned with proper handling of the execution environment, i.e., environment variables, and calls to external command processors are covered by the rules and recommendations in the ENV category.

Signals (11-SIG). The rules and recommendations in this category are concerned with raising and handling signals in a secure manner, including ensuring that signal handlers do not call `longjmp()` and do not modify or access shared objects.

Error Handling (12-ERR). The rules and recommendations in this category are concerned with detecting and handling errors and proper handling of the `errno` variable. Examples include not modifying the `errno` variable and not relying on indeterminate values of `errno`. Below we discuss the rule ERR33-C (detect and handle errors) in more detail and examine how Coccinelle can be used to check this rule. Note that this rule is different from most other rules in that it is actually *application dependent* since errors are detected and handled differently in different applications. Consequently, in order for an automated tool to support checking of this rule, it must be possible to customise and adapt the tool to a specific project's error handling strategy.

Application Programming Interface (13-API). In the version of CCSCS currently under development, this category has no rules, only recommendations, since proper API design is highly application specific. Similar to the error handling (ERR) category above, automated tool support requires a very adaptable tool.

Concurrency (14-CON). The rules and recommendations in this category are general observations concerning concurrent programming such as avoiding race conditions and deadlocks (by locking in a predefined order).

Miscellaneous (49-MSC). The rules and recommendations in this category are those that do not fit into any other category, e.g., it is recommended to compile cleanly at high warning levels (MSC00-C) and it is a rule that a non-void function's flow of control never reaches the end of the function (MSC37-C). Below we discuss rule MSC37-C (ensure that control never reaches the end of a non-void function) in more detail and show how this rule can be checked using Coccinelle.

POSIX (50-POS). The rules and recommendations in this category cover compliance with and proper use of POSIX. In particular things to avoid doing with POSIX, such as calling `vfork()` and not using signals to terminate threads.

3 Compliance Checking with Coccinelle

In this section we discuss how four rules, from the CCSCS categories presented in the previous section, can be checked using the Coccinelle tool. Before going into the details of the individual rules, we briefly introduce Coccinelle; for lack of space we cannot give a thorough introduction to Coccinelle and the languages used to script it, instead we refer to previous work [LBP$^+$09, BDH$^+$09, PLHM08].

The Coccinelle tool was originally developed to provide support for documenting and automating updates to Linux device drivers necessitated by a change in the underlying API, the so-called *collateral evolutions* [PLHM08]. Finding the right place to perform collateral evolutions in a large code base requires a highly configurable and efficient engine for code searching. In Coccinelle this engine is based on model checking of a specialised modal logic, called CTL-VW, over program models [BDH+09] enabling search not only for specific syntactic patterns but also for control flow patterns. Individual program searches (and transformations) are specified in a domain specific language, called SmPL (for Semantic Patch Language), designed to be similar to the unified patch format widely used by Linux kernel developers and other open source developers. Such program searches are called *semantic patches* or even Coccinelle scripts. The combination of easy configurability and efficient search capabilities makes Coccinelle an excellent tool for searching for code patterns that may lead to potential bugs or violations of coding standards. It has been successfully used to search for bugs in open source infrastructure software such as the Linux kernel and the OpenSSL cryptographic library [LBP+09, LLH+10]. The Coccinelle tool is released under the GNU GPLv2 open source license.

3.1 DCL32-C: Guarantee that mutually visible identifiers are unique

The ANSI C99 standard for the C programming language specifies that *at least* 63 initial characters are significant in an identifier. Thus, identifiers that are share a 63 character long prefix may be considered identical by the compiler. The DCL32-C rule requires that all (in scope) identifiers are unique, i.e., must differ within the first 63 characters.

Below a Coccinelle semantic patch is shown that simply searches for all variable declarations. This simple search forms the heart of the semantic patch used to search for potential violations of the DCL32-C rule:

```
@@
type T;
identifier id;
@@
  T id;
```

Observe that this is very similar to what a variable declaration looks like in a C program.

In Figure 2 the full semantic patch is shown. It simply collects all identifiers of length 63 or more and warns if there are (potential) violations of the rule. The rule does not take the scope of the declared identifiers into account and thus may give rise to unnecessary warnings (false positives). However, since identifiers of length 63 or more are rarely used this is unlikely to be a problem in practise. If, for a specific project, it turns out to be a problem, the semantic patch can be extended to take more scope information into account. The semantic patch includes a simple O'Caml script (lines 11 to 21) that collects all the found identifiers (of length 63 or more) and adds them to a hash table. Before adding an identifer to the hash table, it is checked for collisions, and thus potential violations, and a warning is printed if there are (potential) collisions (line 18).

The basic semantic patch searching for declarations has been augmented with a *position meta-variable* denoted `@pos` (line 9). The position meta-variable is bound to the position (line and column number) of each match.

```
@ initialize:ocaml @
let idhash = Hashtbl.create 128

@ decl @
type T;
identifier id;
position pos;
@@
  T id@pos;

@ script:ocaml @
p << decl.pos;
x << decl.id;
@@
if (String.length(x)) >= 63 then
  let sid = String.sub x 0 63 in
  let _ = if (Hashtbl.mem idhash sid) then
            print_endline (warn p "DCL32-C" "Found␣long␣(%d)␣identifier␣'%s'"
               (String.length(x)) x)
          else () in
  Hashtbl.add idhash sid (x,p)
```

Figure 2: Coccinelle script to find "long" identifiers.

3.2 EXP34-C: Do not dereference null pointers

In the CCSCS, the rationale for this rule is that attempts to dereference null pointers result in undefined behaviour. In recent years, attackers and vulnerability researchers have had great success at leveraging null pointer dereferences into full blown security vulnerabilities, making this rule very important for application security. The current version of the CCSCS contains an example involving the Linux kernel and the `tun` virtual network driver.

One potential source of null pointers, as noted in the CCSCS examples, is when memory allocation functions, e.g., `malloc()`, `calloc()`, and `realloc()`, fail and return null. If the return value from allocation functions is not properly checked for failure, and handled accordingly, there is a high risk that a program will eventually, or can be made to, dereference a null pointer.

Using Coccinelle to find such code patterns is straightforward. In Figure 3 the corresponding semantic patch is shown: we first look for calls to the relevant allocation functions (lines 8 to 14). The possible allocation functions are specified using the *disjunction* pattern (denoted by '(', '|', and ')') that succeeds if either of the alternatives (separated by '|') match. Following that, the script looks for a *control flow* path, represented by '`...`', where the identifier (`x`) is *not* assigned to, i.e., a path where it is not modified (line 15), and where the identifier is not tested for "nullness" (line 16). The latter is in order to cut down on the number false positives. Here the '`... WHEN != x = E`' and the '`WHEN != if(E == NULL) S1 else S2`' means along *any* control flow path where assignment to `x` does not occur, i.e., any control flow path where `x` is not modified and which contains no null test on `x`. Finally, we look for a dereference of `x` (lines 17 to 23), again using the disjunction pattern to specify three common ways to dereference a pointer: as a pointer (line 18), as an array (line 20), or for field member access (line 22).

```
 1  @@
 2  identifier x;
 3  expression E,E1;
 4  type T1;
 5  identifier fld;
 6  statement S1, S2;
 7  @@
 8  (
 9    x = (T1) malloc(...)
10  |
11    x = (T1) calloc(...)
12  |
13    x = (T1) realloc(...)
14  )
15    ... WHEN != x = E
16        WHEN != if(E == NULL) S1 else S2
17  (
18    *x
19  |
20    x[E1]
21  |
22    x->fld
23  )
```

Figure 3: Coccinelle script to find dereferencing of null pointers.

Note that, even though the semantic patch specifies that there can be no conditionals with a condition on the form ‘`E==NULL`’ (in line 16), Coccinelle will automatically also match variations of this condition such as ‘`NULL==E`’, and ‘`!E`’. This feature is called *isomorphisms* and is a general, customisable, and scriptable feature of Coccinelle designed to handle syntactic variations of the same semantic concept, in this case, comparing a variable to the `NULL` pointer. Isomorphisms, while not strictly necessary, represent a large reduction in the amount of work a programmer has to do when developing a semantic patch. Isomorphisms are also useful in developing patches that are more complete (cover more cases) since corner and special cases need only be handled once.

While the semantic patch in Figure 3 will catch many common violations of rule EXP34-C, it cannot catch all possible violations. First of all, null pointers may come from many other places than the memory allocation functions, e.g., user defined functions and library functions. In principle it is of course possible to manually extend the semantic patch with all the functions possibly returning a null pointer, however, this quickly becomes unwieldy. Another drawback of the semantic patch, as shown, is that it currently overlooks violations occuring *after* a null test. It is possible to manually refine the semantic patch to take more tests into account in a proper way. In [LBP+09] a more comprehensive Coccinelle approach to dereferencing of null pointers is described. This approach covers not only standard allocations functions, but basically any function returning null. In addition, care is taken to consider null tests and handle them properly.

Another alternative would be if the semantic patch could make use of information from a dataflow analysis. That way it would not be necessary to explicitly cover all syntactic possibilities for null testing or dereferencing. In Section 4 we describe our current work on integrating analysis

```
1  @@
2  identifier x;
3  expression E,E1;
4  function f;
5  identifier fld;
6  @@
7    free(x);
8    ... WHEN != x = E
9  (
10   f(...,x,...)
11 |
12   *x
13 |
14   x[E1]
15 |
16   x->fld
17 )
```

Figure 4: Coccinelle script to find potential access to deallocated memory.

information into Coccinelle scripts.

3.3 MEM30-C: Do not access freed memory

In the C programming language, as in most programming languages, using the value of a pointer to memory that has been deallocated, with the `free()` function, results in undefined behaviour. In practise, reading from deallocated memory may result in crashes, leaks of information, and exploitable security vulnerabilities. Rule MEM30-C ensures that deallocated memory will not be accessed. The problem underlying this rule is very similar to that described in rule EXP34-C (do not dereference null pointers): instead of focusing on null pointers, this rule covers all pointers that have been freed.

In Figure 4 a Coccinelle script covering some of the simple(r) cases of this rule is shown. The script first looks for any identifier (declared in line 2) that occurs as an argument to the `free()` function (line 7). Following that, the script looks for a *control flow* path where the identifier (`x`) is *not* assigned to, i.e., a path where it is not modified (line 8). Finally, using the *disjunction* search pattern (denoted by '(', '|', and ')') that succeeds if either of the alternatives (separated by '|') match, the script looks for a *use* of the identifier that results in the actual violation. Here four common uses are covered: used as an argument to a function (line 10), dereferenced as a pointer (line 12) or an array (line 14), and dereferenced for member field access (line 16).

3.4 ERR33-C: Detect and handle errors

The lack of proper exceptions in the C programming language means that error conditions have to be explicitly encoded and communicated to other parts of the program. Most often a run-time error in a given C function will be communicated by returning an *error value*, frequently -1 or `NULL`. Ignoring an error condition is highly likely to lead to unexpected and/or undefined behaviour, it is therefore essential that the return value is always checked for all calls to a function

```
 1  @ voidfunc @
 2  function FN;
 3  position voidpos;
 4  @@
 5    void  FN@voidpos(...) {
 6      ...
 7    }
 8
 9  @ func disable ret exists @
10  type T;
11  expression E;
12  function FN;
13  position pos != voidfunc.voidpos;
14  @@
15    T  FN@pos(...) {
16      ... WHEN != return E;
17    }
```

Figure 5: Coccinelle script to find non-void functions without a `return` statement.

that may return an error value and that any error condition is handled properly. Rule ERR33-C formalises this requirement.

This rule differs from most of the other rules in the CCSCS in that it is almost entirely application dependent, since it is up to each application or software project to decide how, specifically, error conditions are signalled, what error values are used, what they mean, and how they must be checked and handled. It is therefore impossible to come up with a single, or even a few, rules that will cover the entire spectrum of possibilities. Thus, for a tool to be useful and effective it *must* be very customisable in order to adapt it to project specific code styles and policies. We believe that the specialised semantic patch language (SmPL) used in Coccinelle provides an excellent, and highly adaptable, platform for developing project specific rule checkers.

As an example of how Coccinelle can be customised for project specific error handling standards, we show in [LLH+10] how Coccinelle was used to find several bugs in some error handling code in the OpenSSL cryptographic library. Coccinelle has also been used to find flaws in the error handling of the Linux kernel [LBP+09].

3.5 MSC37-C: Ensure that control never reaches the end of a non-void function

Non-void functions are required to return a value, using the `return` statement. It results in undefined behaviour to use the return value of a non-void function where control flow reaches the end of the function, i.e., without having explicitly returned a value. For this reason the CCSCS requires that all control flows in a non-void function *must* end in a (non-empty) return statement.

Figure 5 shows a semantic patch that finds non-void functions with a control flow path not ending in a non-empty return statement. The overall strategy for this search is to first find all `void` functions (line 1 to 7), i.e., functions that are not supposed to return a value, in order to rule them out in our search. Next, we find all function declarations *except* for the functions we have earlier identified as `void` functions (line 13). Once such a function is found, we start looking for a control flow path that does *not* contain a `return` statement (line 16).

Observe that the head of the latter search pattern (line 9) not only contains the name of the search pattern (`func`) but also a directive to Coccinelle that it should disable the use of the ‘`ret`’-isomorphism (cf. the discussion of isomorphisms in Section 3.2) in order to avoid unwanted, potential interference from the isomorphism system. The header also specifies that the current rule should look for the *existence* of a control flow path with the required property, rather than checking for the property along *all* control flow paths, since we have a potential violation if there is even a single control flow path without a `return` statement.

The problem caught by the above semantic patch is inherently syntactic and control flow based, and thus very well suited for Coccinelle searches. Furthermore, checking for violations can be done in a universal and application independent way.

4 Adding Program Analysis Information

From the discussion in the previous section of the categories and how specific rules can be checked using Coccinelle, it should be clear that while Coccinelle is useful for compliance checking it would beneift greatly from having access to proper program analysis information, e.g., for more precise and comprehensive tracking of potential null pointers. Such information could also be used to make checkers more succinct and efficient because fewer syntactic cases need to be covered. In the following we will illustrate both uses as well as how we intend to make program analysis information available for use in semantic patches. In Section 4.2 we show how such information can be obtained through the Clang tool and we discuss the current status of our integration of Clang into Coccinelle.

4.1 Pointer Analysis: Tracking NULL Pointers and Aliases

Consider the rule EXP34-C (do not dereference null pointers). Here the problem is to find all expressions that may potentially dereference a null pointer. With access to pointer analysis information, every expression that may result in a null pointer can be found and tagged. Note that this is independent of how an expression may result in a null pointer, i.e., it is no longer necessary to explicitly track information only from allocation functions in the semantic patch, since this is handled by the analysis.

Below we show how such analysis information could be incorporated into a semantic patch. The following semantic patch is intended to illustrate one possible way to make analyis information available to semantic patches:

```
@@
identifier x, fld;
expression E1;
analysis[null] NINF;
@@
( *x@NINF
| x@NINF[E1]
| x@NINF->fld
)
```

The main thing to note in the above semantic patch is the '`analysis`' declaration (line 4) that declares a meta-variable, called `NINF`. This meta-variable is then used in much the same way as position meta-variables: by "tagging" an expression with the '`NINF`' meta-variable, e.g., like '`x`' in line 7, only expressions that match the syntax (in this case an array) and that may also result in a null pointer are matched by the semantic patch.

Taking this a step further, we can also use analysis information to find all (sub-)expressions that are potential dereferences and then simply search for all expressions that are both tagged as potentially dereferencing and also as potentially resulting in a null pointer. Here a dereferencing expression is taken to mean an expression that may in any way do a pointer dereference:

```
@@
expression E;
analysis[deref] DEREF;
analysis[null] NINF;
@@
  E@DEREF@NINF
```

Since pointers in C may be *aliases* for the same location in memory, it is important that the pointer analysis not only tracks potential null pointers but also tracks all potentially aliasing pointers. This is often called a *alias analysis* or a *points-to analysis*. Such analysis information would be useful in many other situations, e.g., in the rule MEM30-C (do not access freed memory) where access may occur through an alias. The following semantic patch (with alias analysis information available) would capture this situation (see below for an explanation):

```
1  @@
2  identifier x, y;
3  expression E,E1;
4  function f;
5  identifier fld;
6  analysis[alias] xyalias;
7  @@
8    free(x@xyalias);
9    ... WHEN != y@xyalias = E
10 (
11   f(...,y@xyalias,...)
12 |
13   *y@xyalias
14 |
15   y@xyalias[E1]
16 |
17   y@xyalias->fld
18 )
```

The idea in the above semantic patch is that we first declare an analysis meta-variable in line 6 (called '`xyalias`'). Then, in line 8, we match a call to '`free()`' on an identifier '`x`' and bind the `xyalias` meta-variable to any available alias analysis information for `x`. Following that we match any assignments to and use of *any identifier* `y` that is an *alias* for `x` (represented by `y@xyalias` in lines 9, 11, 13, 15, and 17).

4.2 Integrating Clang and Coccinelle

In the following we describe how program analysis information, such as described in the above section, can be computed using the Clang tool and discuss the current status of our integration of

Clang and Coccinelle.

Clang was chosen as the main program analysis engine for Coccinelle for several reasons: it is open source, it is being (very) actively developed, it has good support for writing new analyses, it provides a robust and proven infrastructure for manipulating C programs, and so forth.

The current version of our implementation of a Coccinelle/Clang integration is a "proof of concept" where the main emphasis has been on making the two tools work together and less on adding language features to the semantic patch language. As a result, it is not possible to use the '`analysis`' declaration illustrated in the semantic patches in the last section. Instead we use positions, as implemented by the '`position`' meta-variables, to look up relevant analysis information. Below we show how this works using Python scripting in the semantic patch:

```
@ initialize:python @

  # read in analysis information generated by Clang into
  # Python dictionaries: DEREF and NINF indexed by positions

@ expr @
expression E;
position pos;
@@
  E@pos

@ script:python @
p << expr.pos
@@

  # lookup DEREF and NINF status in Clang data
  if not (DEREF[p] and NINF[p]):
    # remove the match
  else:
    # accept the match and continue
```

Currently we first run Clang on the source files in order to compute program analysis information. This information is then stored in a file that may subsequently be read by a semantic patch. However, it would be possible to start Clang from within the semantic patch, again using either O'Caml or Python scripting.

4.3 Clang and WPDSs

While Clang provides a good framework supporting the implementation of checkers and program analyses in various forms, e.g., using the monotone framework, they must be programmed directly in C++ and require recompiling the entire Clang tool. In order to make analysis development more flexible and convenient we have added a library for program analysis using weighted push-down systems (WPDS). This allows for program analyses to be specified at the more abstract level of WPDSs. We have also implemented Python bindings for the WPDS library enabling rapid prototyping of analyses without recompilation of Clang.

We have extended Clang with the analysis framework of WPDSs, using the library WALi. This enables us to model the control-flow from Clang as a push-down system, and plug-in different weight domains. Weight domains for different analyses have been presented [RLK07], such as affine-relations analysis, generalised gen-kill analysis and may-aliasing pointer analysis. We have used the gen-kill weight domain to implement a reaching definitions analysis within Clang, and plan to implement a pointer analysis as well. The analysis result can then be pre-processed in Coccinelle scripts, as illustrated above, e.g., to get maybe-null analysis information. The benefit of using Clang is that the control-flow graph of the program is readily available, with some infeasible paths automatically pruned.

The analysis is written as a special analysis pass that constructs the WPDS, assigns weights, and perform a query for each function. The analysis results (annotated weighted finite automata) are output, and subsequently interpreted by the concrete Coccinelle script when analysis information is needed. Currently the Python scripting interface is used with some additional support code for calling Clang and interpreting the output.

4.4 Current Work

The information that we have integrated at this point is the reaching definitions analysis. The output from Clang is a textual representation of the solved WFA, an example of one line of this output is:

```
( p , ( uninit_use.c , ( 5 , 9 ) ) , accept )   <\S.(S - {NULL}) U
                  {(simple:a@uninit_use.c:3:9@uninit_use.c:4:9,1)}>
```

All lines are split into their components:

From state of the WPDS, which will be the state *p* in most cases.

Symbol in this case "*(uninit_use.c , (5 , 9))*" indicating the program point.

To state which will be the accepting state *accept*.

The weight associated with this transition, which is the program analysis information associated with the program point.

The weight again needs to be parsed, in this case into its gen and kill set. In the above example the kill set is empty, and the gen set adds a definition point of the variable *simple:a@uninit_use.c:3:9*, namely that it can be defined at *uninit_use.c:4:9*.

Variables are named from: the function they are defined in, their identifier and the position they are defined at. All positions are made up of: a file name, line number and column number.

Finally, a dictionary data structure is constructed such that the reaching definitions for a variable at a program point can be looked up.

One use is to look for uninitialised variables being used, where the basic semantic patch is:

```
@ uninituse @
type T; identifier I;
position defloc, useloc;
identifier FN;
@@
// look for declarations with no assignment
T@defloc I;
... when any
//which are then used
(
    FN@useloc(...,I,...);
|
    I@useloc
)
```

Before being able to use a location from Coccinelle we have to account for small differences in how locations are presented in the CFG of Clang and Coccinelle, e.g. precisely where a variable is defined:

```
int a;
    ^ Clang define location
   ^ Coccinelle define location
```

Another example is that the use found might be part of a larger expression, so we will have to find the location of the entire expression. Currently we simply map a Coccinelle location to the closest Clang location on the same line.

We can then discard false positive matches, based on whether the data can actually flow from the found definition to the found use, in a somewhat cleaner way than specifying all possible ways the variable could have been modified. The approach of course becomes much more powerful when including analysis information from a pointer analysis.

5 Work in Progress

In this section we discuss the current status and work-in-progress for using Coccinelle to check for CCSCS compliance. In particular we we discuss compliance checking of the full standard for real world software projects.

5.1 Compliance Checking Real World Software

Coccinelle has already been used successfully to find numerous bugs in the Linux kernel, the OpenSSL library, and other open source projects used in the "real world". Especially the experience with bug finding in the Linux kernel shows that the approach scales well even to very large software projects.

One of the biggest problems when checking such large projects, is the number of *false positives*, i.e., warnings of potential violations that turn out not to be violations. Here the customisability of Coccinelle has turned out to be a great tool for reducing the number of false positives, since it enables a programmer to refine the semantic patches to take the project specific code styles into account that give rise to the most false positives.

The integration of program analysis information, e.g., obtained from Clang, will enable a code search to take (more) semantic information into account and will thus reduce the number of false positives further.

5.2 Implementing Checkers for the Full Standard

While we have only detailed the implementation of Coccinelle checkers for four of the 118 rules in the CCSCS, we have implemented checkers for approximately 25 rules and plan to implement Coccinelle checkers for all the CCSCS rules that are suitably application independent. For rules that are application dependent, such as rule ERR33-C (discussed in Section 3.4), it may be possible to provide an "abstract" semantic patch that can be instantiated with project specific details similar to the approach taken in [LLH+10].

We intend to make the complete set of checkers available for download as open source.

6 Related Work

The past decade has seen the development and release of numerous compile time tools for program navigation, bug finding and code style checking for programs written in C, as well as many other languages. These tools include the MC tool [ECCH00] (later used as basis for the commercial tool Coverity Prevent). Similar to Coccinelle, the MC tool is a bug finder that can be adapted to specific projects, however the source code for MC has never been released.

Splint [LE01] and Flawfinder [Whe06] are two examples of Open Source bug finders. Both are able to check for a relatively small set of bugs. Both are somewhat adaptable but requires either (light-weight) annotation of the source code or Python programming.

While several commercial static analysis tools support compliance checking[6] for a wide spectrum of coding standard, including the CCSCS, we are not aware of any Open Source bug finder tools working towards this goal.

7 Conclusion

In this paper we have shown that the Coccinelle tool is very well suited for checking some of the rules comprising the CERT C Secure Coding Standard. We have further argued that integrating program analysis information would facilitate even more comprehensive, more expressible, and even more flexible semantic patches to be written.

Bibliography

[BDH+09] J. Brunel, D. Doligez, R. R. Hansen, J. L. Lawall, G. Muller. A foundation for flow-based program matching: using temporal logic and model checking. In Shao and Pierce (eds.), *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*. Pp. 114–126. ACM, Savannah, GA, USA, Jan. 2009.

[ECCH00] D. R. Engler, B. Chelf, A. Chou, S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Pp. 1–16. San Diego, CA, Oct. 2000.

[LBP+09] J. L. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart, G. Muller. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009*. Pp. 43–52. IEEE, Estoril, Lisbon, Portugal, June/July 2009.

[LE01] D. Larochelle, D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proc. of the 10th USENIX Security Symposium*. USENIX, Washington D.C., USA, Aug. 2001.
http://lclint.cs.virginia.edu/

[6] See the CCSCS web page for details on tool support.

[LLH$^+$10] J. L. Lawall, B. Laurie, R. R. Hansen, N. Palix, G. Muller. Finding Error Handling Bugs in OpenSSL Using Coccinelle. In *Eighth European Dependable Computing Conference, EDCC-8*. Pp. 191–196. IEEE Computer Society, Valencia, Spain, Apr. 2010.

[PLHM08] Y. Padioleau, J. L. Lawall, R. R. Hansen, G. Muller. Documenting and automating collateral evolutions in linux device drivers. In Sventek and Hand (eds.), *Proceedings of the 2008 EuroSys Conference*. Pp. 247–260. ACM, Glasgow, Scotland, UK, Apr. 2008.

[PLM10] N. Palix, J. L. Lawall, G. Muller. Tracking code patterns over multiple software versions with Herodotos. In Jézéquel and Südholt (eds.), *Proceedings of the 9th International Conference on Aspect-Oriented Software Development, AOSD 2010*. Pp. 169–180. ACM, Rennes and Saint-Malo, France, Mar. 2010.

[RLK07] T. Reps, A. Lal, N. Kidd. Program analysis using weighted pushdown systems. In *Proceedings of the 27th international conference on Foundations of software technology and theoretical computer science, FSTTCS'07*. Pp. 23–51. Springer-Verlag, 2007.

[RSJM05] T. Reps, S. Schwoon, S. Jha, D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming* 58(1-2):206–263, 2005.

[Sea08] R. C. Seacord. *The CERT C Secure Coding Standard*. Addison-Wesley, 2008.

[Whe06] D. Wheeler. Flawfinder Home Page. Web page: http://www.dwheeler.com/flawfinder/, Oct. 2006.
http://www.dwheeler.com/flawfinder/

Safe Integration of Annotated Components in Open Source Projects

Sérgio Areias[1]*, Daniela da Cruz[2], Pedro Rangel Henriques[3] and Jorge Sousa Pinto[4]

[1] pg13381@alunos.uminho.pt
[2] danieladacruz@di.uminho.pt
[3] prh@di.uminho.pt
[4] jsp@di.uminho.pt
Departamento de Informática
Universidade do Minho

Abstract:

The decision of using existing software components versus building from scratch custom software is one of the most complex and important choices of the entire development/integration process. However, the reuse of software components raises a spectrum of issues, from requirements negotiation to product selection and integration. The correct tradeoff is reached after having analyzed advantages and issues correlated to the reuse. Despite the reuse failures in real cases, many efforts have been made to make this idea successful.

In this context of *software reuse in open source projects*, we address the problem of reusing annotated components proposing a rigorous approach to assure the quality of the application under construction. We introduce the concept of *caller-based slicing* as a way of certifying that the *integration of a component annotated with a contract* into a system will preserve the correct behavior of the former, avoiding malfunctioning after integration.

To complement the efforts done and the benefits of slicing techniques, there is also a need to find an efficient way to visualize the main program with the annotated components and the slices. To take full profit of visualization, it is crucial to combine the visualization of the control/data flow with the textual representation of source code. To attain this objective, we extend the notions of System Dependence Graph and Slicing Criterion to cope with annotations.

Keywords: Caller-based slicing, Annotated System Dependency Graph

1 Introduction

Reuse is a very simple an natural concept, however in practice is not so easy. According to the literature, selection of reusable components has proven to be a difficult task [MS93]. Sometimes this is due to the lack of maturity on supporting tools that should easily find a component on a repository or library [SV03]. Also, non experienced developers tend to reveal difficulties when

* This author is sponsored by the Foo Society under Grant Nr. 42-23.

describing the desired component in technical terms. Most of the times, this happens because they are not sure of what they want to find [SV03, SS07]. Another barrier is concerned with reasoning about component similarities in order to select the one that best fits in the problem solution; usually this is an hard mental process [MS93].

Integration of reusable components has also proven to be a difficult task, since the process of understanding and adapting components is hard, even for experienced developers [MS93]. Another challenge to component reuse is to certify that the integration of such component in a open-source software system (OSS) keeps it correct. This is, to verify that the way the component is invoked will not lead to an incorrect behavior.

A strong demand for formal methods that help programmers to develop correct programs has been present in software engineering for some time now. The Design by Contract (DbC) approach to software development [Mey92] facilitates modular verification and certified code reuse. The contract for a software component (a sub-program, or commonly, a procedure) can be regarded as a form of enriched software documentation composed of annotations (pre-conditions, post-conditions and invariants) that fully specifies the behavior of that component. So, a well-defined annotation can give us most of the information needed to integrate a reusable component in a OSS, as it contains crucial information about some constraints to obtain the correct behavior from the component.

In this context, we say that the annotations (the pre-, post-conditions and invariants that form the contract) can be used to verify that each component invocation is valid (preserves the contract); in that way, we can guarantee that a correct system will still be correct after the integration of that component. This is the motivation for our research: to find a way to help on the safety reuse of components.

This article introduces the concept of **caller-based slicing**, an algorithm that takes into account the calls to an annotated component in order to certify that it is being correctly used. To support the idea, we also introduce GamaPolarSlicer, a tool that implements such algorithm: to identify when an invocation is violating the component annotation; and to display a diagnostic or guidelines to correct it.

The remainder of paper is composed of 8 sections. Section 2 is devoted to basic concepts crucial to the understanding of the rest of the paper: the notions of *slicing* and *system dependency graph* are introduced. Section 3 formalizes the definition of *caller-based slicing* that supports our approach to safety reused of annotated components. Section 4 defines the concept of *annotated System Dependency Graph* (SDGa), used for the visual analysis of the slices and pre-conditions preservation. Section 5 illustrates the main idea through a concrete example. Section 6 gives a general overview of GamaPolarSlicer, introducing its architecture, functionalities and implementation details. Section 7 discusses related work on *slicing programs with annotated components* as it is the main idea behind our proposal. Section 8 discusses related work on *visualization of (sliced) programs*, because we strongly believe that good visual tool is crucial for software analysis. Then the paper is closed in Section 9.

2 Basic Concepts

In this section we introduce both the original concepts of slicing and system dependency graph.

2.1 Slicing

Since Weiser first proposed the notion of slicing in 1979 in his PhD thesis [Wei79], hundreds of papers have been proposed in this area. Tens of variants have been studied, as well was algorithms to compute them. Different notions of slicing have different properties and different applications. These notions vary from Weiser's syntax-preserving static slicing to amorphous slicing which is not syntax-preserving; algorithms can be based on dataflow equations, information flow relations or dependence graphs.

Slicing was first developed to facilitate program debugging [M.93, ADS93, WL86], but it is then found helpful in many aspects of the software development life cycle, including software testing [Bin98, HD95], software metrics [OT93, Lak93], software maintenance [CLM96, GL91], program comprehension [LFM96, HHF+01], component re-use [BE93, CLM95], program integration [BHR95, HPR89] and so on.

Program slicing, in its original version, is a decomposition technique that extracts from a program the statements relevant to a particular computation. A program slice consists of the parts of a program that potentially affect the values computed at some point of interest referred to as a *slicing criterion*.

Definition 1 (Slicing Criterion) *A* static slicing criterion *of a program P consists of a pair $C=(p,V_s)$, where p is a statement in P and V_s is a subset of the variables in P.*

A slicing criterion $C=(p,V_s)$ determines a projection function which selects from any state trajectory only the ordered pairs starting with p and restricts the variable-to-value mapping function σ to only the variables in V_s.

Definition 2 (State Trajectory) *Let $C=(p,V_s)$ be a static slicing criterion of a program P and $T=<(p_1,\sigma_1),(p_2,\sigma_2),...,(p_k,\sigma_k)>$ a state trajectory of P on input I. $\forall i, 1\leq i\leq k$:*

$$Proj'_C(p_i,\sigma_i)=\begin{cases}\lambda & if\, p_i\neq p\\ <(p_i,\sigma_i|V_s)> & if\, p_i=p\end{cases}$$

where $\sigma_i|V_s$ is σ_i restricted to the domain V_s, and λ is the empty string.

The extension of $Proj'$ to the entire trajectory is defined as the concatenation of the result of the application of the function to the single pairs of the trajectory:

$$Proj_C(T)=Proj'_C(p_1,\sigma_1)...Proj'_C(p_k,\sigma_k)$$

A program slice is therefore defined behaviorally as any subset of a program which preserves a specified projections in its behavior.

Definition 3 (Static Slicing) *A* static slice *of a program P on a static slicing criterion $C=(p,V_s)$ is any syntactically correct and executable program P' that is obtained from P by deleting zero or more statements, and whenever P halts, on input I, with state trajectory T, then P' also halts, with the same input I, with the trajectory T', and $Proj_C(T)=Proj_C(T')$.*

Related work of slicing programs taking into account the annotations of a program will be referred in Section 7.

2.2 System Dependency Graph

The use of dependency graphs to visualize the data and control flow of a program has been widely accepted in the last years (Section 8).

Before exploring the use Dependency Graphs for visualization and comprehension, we present below the definitions of Procedure Dependency Graph and System Dependency Graph.

Definition 4 (Procedure Dependence Graph) *Given a procedure $\mathscr{P}$, a Procedure Dependence Graph, PDG, is a graph whose vertices are the individual statements and predicates (used in the control statements) that constitute the body of $\mathscr{P}$, and the edges represent control and data dependencies among the vertices.*

In the construction of the PDG, a special node, considered as a predicate, is added to the vertex set: it is called the *entry* node and is decorated with the procedure name.

A control dependence edge goes from a predicate node to a statement node if that predicate condition the execution of the statement. A data dependence edge goes from an assignment statement node to another node if the variable assigned at the source node is used (is referred to) in the target node.

Additionally to the natural vertices defined above, some extra assignment nodes are included in the PDG linked by control edges to the entry node: we include an assignment node for each formal input parameter, another one for each formal output parameter, and another one for each returned value — these nodes are connect to all the other by data edges as stated above. Moreover, we proceed in a similar way for each call node; in that case we add assignment nodes, linked by control edges to the call node, for each actual input/output parameter (representing the value passing process associated with a procedure call) and also a node to receiving the returned values.

Definition 5 (System Dependence Graph) *A System Dependence Graph, SDG, is a collection of Procedure Dependence Graphs, PDGs, (one for the main program, and one for each component procedure) connected together by two kind of edges: control-flow edges that represent the dependence between the caller and the callee (an edge goes from the call statement into the entry node of the called procedure); and data-flow edges that represent parameter passing and return values, connecting actual$_{in,out}$ parameter assignment nodes with formal$_{in,out}$ parameter assignment nodes.*

3 Caller-based slicing

In this section, we introduce our slicing algorithm. We start by extending the notion of static slicing and slicing criterion to cope with the contract of a program.

Definition 6 (Annotated Slicing Criterion) *An* annotated slicing criterion *of a program $\mathscr{P}$ con-*

sists of a triple $C_a = (a, S_i, V_s)$, where a is an annotation of $\mathscr{P}_a$ (the annotated callee), S_i correspond to the statement of $\mathscr{P}$ calling $\mathscr{P}_a$ and V_s is a subset of the variables in $\mathscr{P}$ (the caller), that are the actual parameters used in the call and constrained by α or δ.

Definition 7 (Caller-based slicing) *A* caller-based slice *of a program $\mathscr{P}$ on an annotated slicing criterion $C_a = (\alpha, call_f, V_s)$ is any subprogram $\mathscr{P}'$ that is obtained from $\mathscr{P}$ by deleting zero or more statements in a two-pass algorithm:*

1. *a first step to execute a backward slicing with the traditional slicing criterion $C = (call_f, V_s)$ retrieved from C_a — $call_f$ corresponds to the call statement under consideration, and V_s corresponds to the set of variables present in the invocation $call_f$ and intervening in the precondition formula (α) of f*

2. *a second step to check if the statements preceding the $call_f$ statement will lead to the satisfaction of the callee precondition.*

For the second step in the two-pass algorithm, in order to check which statements are respecting or violating the precondition we are using abstract interpretation, in particular symbolic execution.

According to the original idea of *James King* in [Kin76], symbolic execution can be described as "instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values. The execution proceeds as in a normal execution except that values may he symbolic formulas over the input symbols."

Using symbolic execution we will be able to propagate the precondition of the function being called through the statements preceding the call statement. In particular, to integrate symbolic execution with our system, we are thinking to use JavaPathFinder [APV07]. JavaPathFinder is a tool than can perform program execution with symbolic values. Moreover, JavaPathFinder can mix concrete and symbolic execution, or switch between them. JavaPathFinder has been used for finding counterexamples to safety properties and for test input generation.

To sum up, the main goal of our caller-based slicing algorithm is to facilitate the use of annotated components by discovering statements that are critical for the satisfaction of the precondition, i.e., that do not verify the precondition or whose statements values can lead to its non-satisfaction (a kind of *tracing call analysis of annotated procedures*).

4 Annotated System Dependency Graph (SDGa)

In this section we present the definition of Annotated System Dependency Graph, SDG_a for short, that is the internal representation that supports our slicing-based code analysis approach.

Definition 8 (Annotated System Dependence Graph) *An Annotated System Dependency Graph, SDG_a, is a SDG in which some nodes of its constituent PDGs are annotated nodes.*

Definition 9 (Annotated Node) *Given a PDG for an annotated procedure $\mathscr{P}_a$, an Annotated Node is a pair $< S_i, a >$ where S_i is a statement or predicate (control statement or entry node) in $\mathscr{P}_a$, and a is its annotation: a pre-condition α, a post-condition ω, or an invariant δ.*

The differences between a traditional SDG and an SDG$_a$ are:

- Each procedure dependency graph (PDG) is decorated with a precondition as well as with a postcondition in the entry node;
- The *while* nodes are also decorated with the loop invariant (or true, in case of invariant absence);
- The *call* nodes include the pre- and postcondition of the procedure to be called (or true, in case of absence); these annotations are retrieved from the respective PDG and instantiated as explained below.

We can take advantage from the *call linkage dictionary* present in the SDG$_a$ (inherited from the underlying SDG) to associate the variables used in the calling statement (the actual parameters) with the formal parameters involved in the annotations.

Given a program and an annotated slicing criterion, we identify the node of the respective SDG$_a$ that corresponds to the criterion (yellow node in Figure 1). After building the respective caller-based slice, the critic statements will be highlighted in the graph, making easier to identify the statements violating the precondition (red nodes in Figure 1).

5 An illustrative example

To illustrate the previous definitions and our proposal, consider the program listed below (Example 1) that computes the maximum difference among student ages in a class.

Example 1 DiffAge

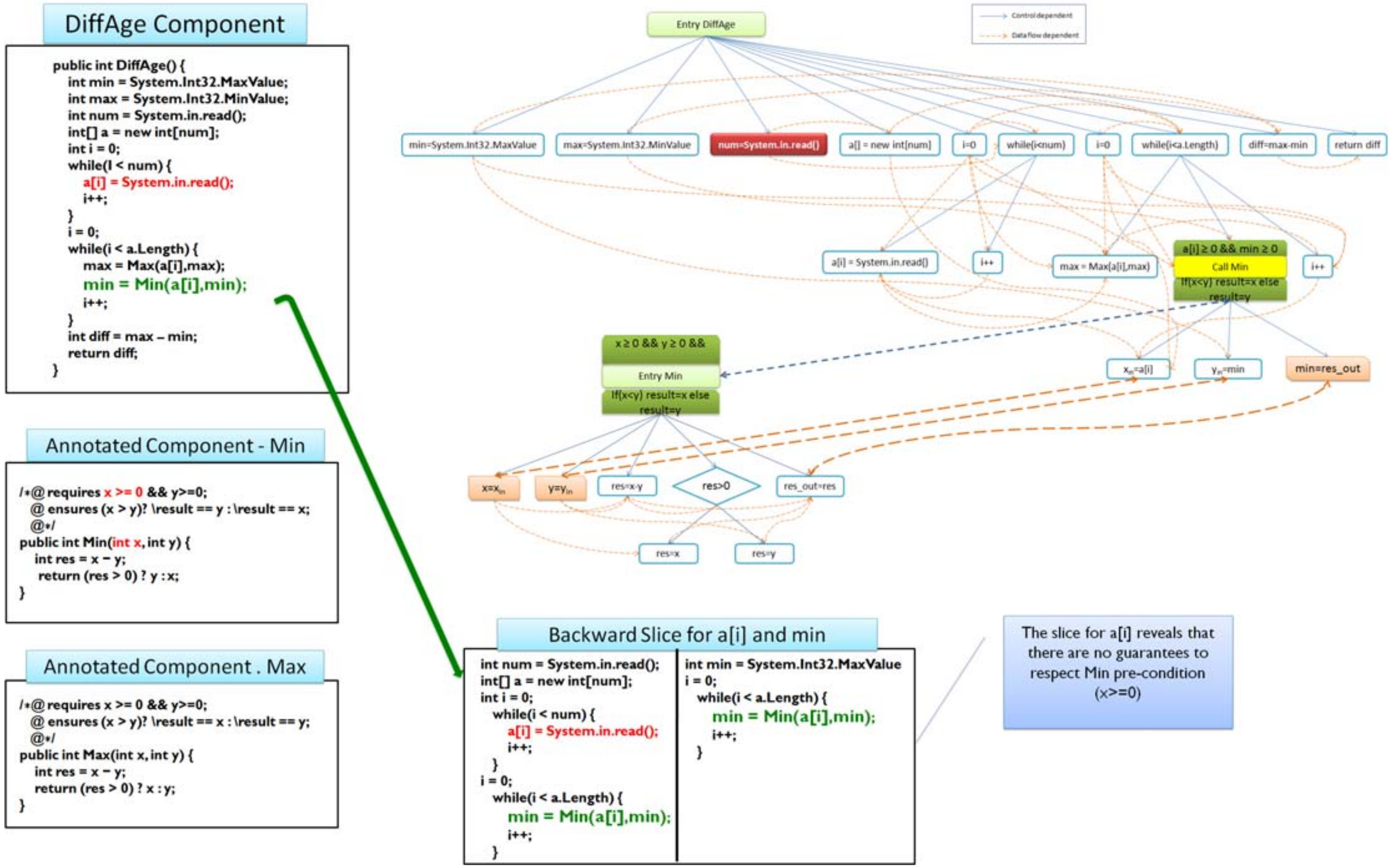
```
public int DiffAge() {
    int min = System.Int32.MaxValue, max = System.Int32.MinValue, diff;

    System.out.print(”Number of elements: ”);
    int num = System.in.read();
    int[] a = new int[num];
    for(int i=0; i<num; i++) { a[i] = System.in.read();  }

    for(int i=0; i<a.Length; i++) {
        max = Max(a[i],max);
        min = Min(a[i],min);
    }

    diff = max - min;
    System.out.println(”The difference between the greatest ” +
                       ”and the smallest ages is ” + diff);
    return diff;
}
```

This program reuses two annotated components: `Min`, defined in Example 2, that returns the smallest of two positive integers; and Max, defined in Example 3, that returns the greatest of two positive integers.



DiffAge Component

```
public int DiffAge() {
   int min = System.Int32.MaxValue;
   int max = System.Int32.MinValue;
   int num = System.in.read();
   int[] a = new int[num];
   int i = 0;
   while(I < num) {
      a[i] = System.in.read();
      i++;
   }
   i = 0;
   while(i < a.Length) {
      max = Max(a[i],max);
      min = Min(a[i],min);
      i++;
   }
   int diff = max – min;
   return diff;
}
```

Annotated Component - Min

```
/*@ requires x >= 0 && y>=0;
  @ ensures (x > y)? \result == y : \result == x;
  @*/
public int Min(int x, int y) {
   int res = x – y;
   return (res > 0) ? y : x;
}
```

Annotated Component . Max

```
/*@ requires x >= 0 && y>=0;
  @ ensures (x > y)? \result == x : \result == y;
  @*/
public int Max(int x, int y) {
   int res = x – y;
   return (res > 0) ? x : y;
}
```

Backward Slice for a[i] and min

```
int num = System.in.read();
int[] a = new int[num];
int i = 0;
  while(i < num) {
     a[i] = System.in.read();
     i++;
  }
i = 0;
  while(i < a.Length) {
     min = Min(a[i],min);
     i++;
  }
```

```
int min = System.Int32.MaxValue
i = 0;
  while(i < a.Length) {
     min = Min(a[i],min);
     i++;
  }
```

The slice for a[i] reveals that there are no guarantees to respect Min pre-condition (x>=0)

Figure 1: SDG$_a$ for a program and its role on Caller-based Slicing

Suppose that we want to study (or analyze) the call to `Min` in the context of `DiffAge` program.

For that purpose, the slicing criterion will be: $C_a = (x \geq 0 \&\& y \geq 0, Min, \{a[i], min\})$

With this criterion, a backward slicing process is performed taking into account the variables present in V_s. Then, using the obtained slices, the detection of contract violations is executed. For that, the precondition is back propagated (using symbolic execution) along the slice to verify if it is preserved after each statement. Observing the slice corresponding to the variable `a[i]` (see Example 4 below), is evident that it can not be guaranteed that all integer elements are greater than zero; so a potential precondition violation is detected.

Example 2 Min

$/*@$ requires $x \geq 0$ && $y \geq 0$
$@$ ensures $(x > y)$? \result $==x$: \result $==y$
$@*/$

```
public int Min(int x, int y) {
int res;
$res = x - y$;
return ((res > 0)? y : x);
}
```

Example 3 Max

$/*@$ requires $x \geq 0$ && $y \geq 0$
$@$ ensures $(x > y)$? \result $==y$: \result $==x$
$@*/$

```
public int Max(int x, int y) {
int res;
$res = x - y$;
return ((res > 0)? x : y);
}
```

Example 4 Backward Slice for a[i]

```
int[] a = new int[num];
for(int i=0; i<num; i++) { a[i] = System.in.read(); }
for(int i=0; i<a.Length; i++) {
    max = Max(a[i],max);
    min = Min(a[i],min);   }
```

All the contract violations detected will be reported during the next step. In the example above, the user will receive an *warning* alerting to the possibility of calling Min with negative numbers (what does not respect the precondition).

As referred, in order to visualize the contracts that are violated and the critical statements, we display the SDG$_a$ with such entities colored in red (see Figure 1). The role of the SDG$_a$ will be crucial not only to understand the data and control flow of a program as well as to understand the impact of the annotations and their violations.

6 GamaPolarSlicer

In this section, we introduce GamaPolarSlicer, a tool that we are building to implement our ideas; it will become available to open source communities, as soon as possible. This project is being developed in the context of the *CROSS project — An Infrastructure for Certification and Re-engineering of Open Source Software* at Universidade do Minho[1].

First we describe the architecture of the tool, and then we give some technical details about its implementation.

6.1 Architecture

As referred previously, our goal is to ease the incorporation process of an annotated component into an existent system. This integration should be smooth, in the sense of that it should not turn a correct system into an incorrect one.

To achieve this goal, it is necessary:

- to verify the component correctness with respect to its contract (using a traditional *Verification Condition Generator*, already incorporated in GamaSlicer [CHP10b], available at http://gamaepl.di.uminho.pt/gamaslicer);
- to verify if the actual calling context preserves the precondition;
- to verify if the component's output, specified by its postcondition, agrees with the value expected by the caller, i.e., if the returned value is properly handled in the caller context.

The chosen architecture is based on the classical structure of a language processor. Figure 2 shows GamaPolarSlicer architecture.

- **Source Code** — the input to analyze.
- **Lexical Analysis, Syntactic Analysis, Semantic Analysis** — the Lexical layer converts the input into symbols that will be later used in the identifiers table. The Syntactic layer uses the result of the Lexical layer above and analyzes it to identify the syntactic structure of it. The Semantic layer adds the semantic information to the result from the Syntactic layer. It is in this layer that the identifier table is built.
- **Invocations Repository** — is where all invocations found on the input are stored in order to be used later as support to the slicing process.
- **Annotated Components Repository** — is where all components with a formal specification (precondition and postcondition at least) are stored. It is used in the slicing process only to filter the invocations (from the invocation repository) without any annotation. Has an important role when verifying if the invocation respects component's contract.
- **Identifiers Table** — has an important role on this type of programs as always. All symbols and associated semantic found during the analysis phase are stored here. It will be one of the backbones of all structures supporting the auxiliary calculations.

[1] More details about this project can be found in http://wiki.di.uminho.pt/twiki/bin/view/Research/CROSS/WebHome

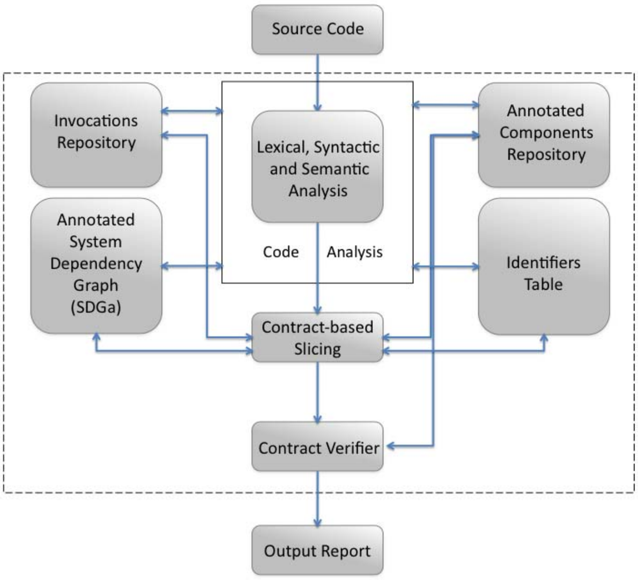


Figure 2: GammaPolarSlicer Architecture

- **Annotated System Dependency Graph** — is the intermediate structure chosen to apply the slicing.

- **Caller-based Slicing** — uses both invocations repository and annotated components repository to extract the parameters to execute the slicing for each invoked annotated component. The resulting slice is a SDG_a this a subgraph of the original SDG_a.

- **Contract Verification** — using the slice that resulted from the layer above, and using the component contract, this layer analyzes every node on the slice and verifies in all of them if and verifies in all of them if every precondition (belonging to the contract annotation) is satisfied.

- **Output Report** — presents to the user a view of all violations found during the whole process; we plan to include, in the future, a diagnosis of each violation and suggestions to overcome it. At moment the output is just textual, but we are working on a graph visualization and navigation module to display the SDGa as preview in Figure 1. Providing a visual output is, in our opinion, a fundamental feature of our analyzing tool.

6.2 Implementation

To address all the ideas, approaches and techniques presented in this paper, it was necessary to choose the most suitable technologies and environments to support the development.

To address the *design-by-contract* approach we decide to use the Java Modeling Language (JML) [2]. JML is a formal behavior interface specification language, based on design-by-contract paradigm, that allows code annotations in Java programs [LC04].

JML is quite useful as allows to describe how the code should behave when running it [LC04]. Preconditions, postconditions and invariants are examples of formal specifications that JML provides.

As the goal of the tool is not to create a development environment but to enhance an existing one, we decided to implement it as an Eclipse [3] plugin.

The major reasons that led to this decision were: the large community and support. Eclipse is one of the most popular frameworks to develop Java applications and thus a perfect tool to test our goal; the fact that it includes a great environment to develop new plugins. The Plugin Development Environment (PDE) [4] that allows a faster and intuitive way to develop Eclipse plugins; has a built-in support for JML, freeing us from checking the validity of such annotations.

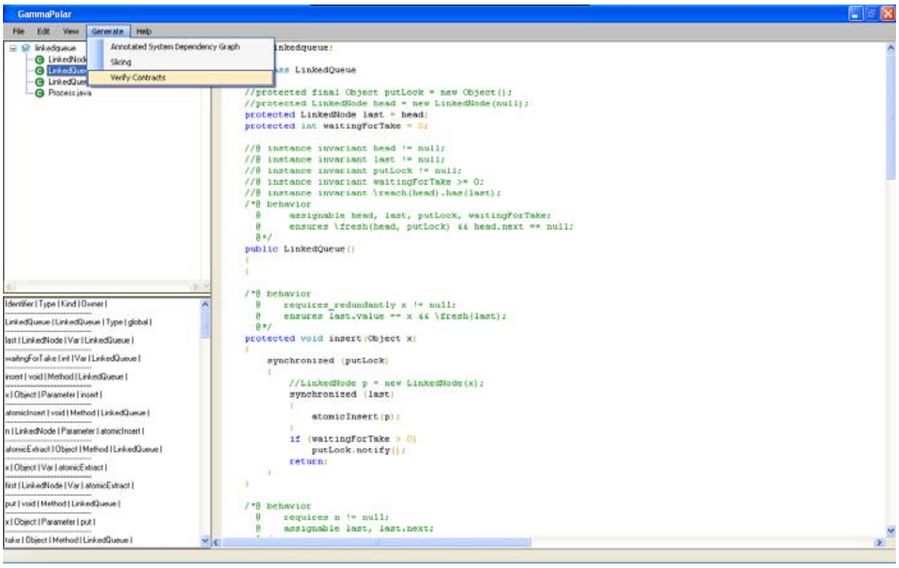
However, the parser generated for Java/JML grammar exceeded the limit of bytes allowed to a Java class (65535 bytes). Thus, this limitation led us to abandon the idea of the Eclipse plugin and implement GamaPolarSlicer using Windows Forms and C# (using the .NET framework).

A scratch of the first version of GamaPolarSlicer prototype is depicted in Figure 3.

[2] http://www.cs.ucf.edu/ leavens/JML/

[3] http://www.eclipse.org/

[4] http://www.eclipse.org/pde/



```
inkedqueue;

ass LinkedQueue

//protected final Object putLock = new Object();
//protected LinkedNode head = new LinkedNode(null);
protected LinkedNode last = head;
protected int waitingForTake = 0;

//@ instance invariant head != null;
//@ instance invariant last != null;
//@ instance invariant putLock != null;
//@ instance invariant waitingForTake >= 0;
//@ instance invariant \reach(head).has(last);
/*@ behavior
  @     assignable head, last, putLock, waitingForTake;
  @     ensures \fresh(head, putLock) && head.next == null;
  @*/
public LinkedQueue()
{
}


/*@ behavior
  @    requires_redundantly x != null;
  @    ensures last.value == x && \fresh(last);
  @*/
protected void insert(Object x)
{
    synchronized (putLock)
    {
        //LinkedNode p = new LinkedNode(x);
        synchronized (last)
        {
            atomicInsert(p);
        }
        if (waitingForTake > 0)
            putLock.notify();
        return;
    }
}

/*@ behavior
  @    requires n != null;
  @    assignable last, last.next;
```

```
Identifier | Type | Kind | Owner |
--------------------------------
LinkedQueue | LinkedQueue | Type | global |
--------------------------------
last | LinkedNode | Var | LinkedQueue |
--------------------------------
waitingForTake | int | Var | LinkedQueue |
--------------------------------
insert | void | Method | LinkedQueue |
--------------------------------
x | Object | Parameter | insert |
--------------------------------
atomicInsert | void | Method | LinkedQueue |
--------------------------------
n | LinkedNode | Parameter | atomicInsert |
--------------------------------
atomicExtract | Object | Method | LinkedQueue |
--------------------------------
x | Object | Var | atomicExtract |
--------------------------------
first | LinkedNode | Var | atomicExtract |
--------------------------------
put | void | Method | LinkedQueue |
--------------------------------
x | Object | Parameter | put |
--------------------------------
take | Object | Method | LinkedQueue |
```

Figure 3: GamaPolarSlicer prototype

7 Related Work — Slicing

In this section we review the published work on the area of slicing annotated programs, as those contribution actually motivate the present proposal. Although the works referred use the annotations to slice a program, the concrete goal of such works differs from ours. The main difference is that we do not assume that all the procedures in a program are annotated and correct with respect to these contracts. We are assuming that only the procedure being integrated is annotated.

In [CH96], *Comuzzi et al* present a variant of program slicing called *p-slice* or *predicate slice*, using Dijkstra's weakest preconditions (wp) to determine which statements will affect a specific predicate. Slicing rules for assignment, conditional, and repetition statements were developed. They presented also an algorithm to compute the minimum slice.

In [CLYK01], *Chung et al* present a slicing technique that takes the specification into account. They argue that the information present in the specification helps to produce more precise slices by removing statements that are not relevant to the specification for the slice. Their technique is based on the weakest pre-condition (the same present in *p-slice*) and strongest post-condition — they present algorithms for both slicing strategies, backward and forward.

Comuzzi et al [CH96], and *Chung et al* [CLYK01], provide algorithms for code analysis enabling to identify suspicious commands (commands that do not contribute to the postcondition validity).

In [HHF$^+$01], *Harman et al* propose a generalization of conditioned slicing called pre/post conditioned slicing. The basic idea is to use the pre-condition and the negation of the post-condition in the conditioned slicing, combining both forward and backward conditioning. This type of program slicing is based on the following rule: "Statements are removed if they cannot lead to the satisfaction of the negation of the post condition, when executed in an initial state which satisfies the pre-condition". In case of a program which correctly implements the pre- and post-condition, all statements from the program will be removed. Otherwise, those statements that do not respect the conditions are left, corresponding to statements that potentially break the conditions (are either incorrect or which are innocent but cannot be detected to be so by slicing). The result of this work can be applied as a correctness verification for the annotated procedure.

In [CHP10a], *Cruz et al* propose the contract-based slicing notion. Given any specification-based slicing algorithm (working at the level of commands), a contract-based slice can be calculated by slicing the code of each individual procedure independently with respect to its contract (called an *open slice*), or taking into consideration the calling contexts of each procedure inside a program (called a *closed slice*).

8 Related Work — Visualization of (sliced) programs

As in this paper we also focus on the visualization of programs with annotated components, and their slices that trace the calls with respect to the called preconditions, we devote this section to review the contributions on the area of slice visualization that more directly influence our proposal.

In [BE94], *Ball et al.* present SeeSlice, an interactive browsing tool to better visualize the data generated by slicing tools. The SeeSlice interface facilitates the visualization by making slices fully visible to user, even if they cross the boundaries of procedures and files.

In [GO97], *Gallagher et al.* propose an approach in order to reduce the visualization complexity by using decomposition slices. A decomposition slice is a kind of slice that depends only on a variable (or a set of variables) and does not consider the location of the statement (a traditional slice depends on both a variable and a statement in a program). The decomposition slice visualization implemented in Surgeon's Assistant [Gal96] visualizes the inclusion hierarchy as a graph using the VCG (Visualization of Compiler Graphs) [San95].

In [88101], *Deng et al* present Program Slice Browser, an interactive and integrated tool which main goal is to extract useful information from a complex program slice. Some of the features of such tool are: adaptable layout for convenient display of a slice; multi-level slice abstractions; integration with other visualization components, and capabilities to support interaction and cross-referencing within different views of the software.

In [Kri04], *Krinke* presents a declarative approach to layout Program Dependence Graphs (PDG) that generates comprehensible graphs of small to medium size procedures. The authors discussed how a layout for PDG can be generated to enable an appealing presentation. The PDG and the computed slices are shown in a graphical way. This graphical representation is combined with the textual form, as the authors argue that is much more effective than the graphical one. The authors also solved the problem of loss of locality in a slice, using a distance-limited approach; they try to answer research questions such as: 1) why a statement is included in the slice?, and 2) how strong is the influence of the statement on the criterion?

In [Bal04], *Balmas* presents an approach to decompose System Dependence Graphs in order to have graphs of manageable size: groups of nodes are collapsed into one node. The system implemented provides three possible decompositions to be browsed and analyzed through a graphical interface: nodes belonging to the same procedure; nodes belonging to the same loop;

nodes belonging to the two previous ones.

9 Conclusion

As can be seen along the paper, the motivation for our research is to apply slicing, a well known technique in the area of source code analysis, to create a tool that aids programmers to build correct open source programs reusing annotated procedures.

The tool under construction, GamaPolarSlicer, was described in Section 6. Its architecture relies upon the traditional compiler structure; on one hand, this enables the automatic generation of the tool core blocks, from the language attribute grammar; on the other hand, it follows an approach in which our research team has a large knowhow (apart from many DSL compilers, we developed a lot of Program Comprehension tools: Alma, Alma2, WebAppViewer, BORS, and SVS). The new and complementary blocs of GamaPolarSlicer implement slice and graph-traversal algorithms that have a sound basis, as described in Sections 2, 3, and 4; this allows us to be confident in there straight-forward implementation.

GamaPolarSlicer will be included in Gama project (for more details see http://gamaepl.di.uminho.pt/gama/index.html). This project aims at mixing specification-based slicing algorithms with program verification algorithms to analyze annotated programs developed under Contract-base Design approach. GamaSlicer is the first tool built under this project for intra-procedural analysis that is available at http://gamaepl.di.uminho.pt/gamaslicer/.

We believe that this set of tools will save time and make safer the process of build open-source software systems.

Bibliography

[88101] Program Slice Browser. In *IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension*. P. 50. IEEE Computer Society, Washington, DC, USA, 2001.

[ADS93] H. Agrawal, R. A. DeMillo, E. H. Spafford. Debugging with Dynamic Slicing and Backtracking. *Software - Practice and Experience* 23(6):589–616, 1993.
citeseer.ist.psu.edu/agrawal93debugging.html

[APV07] S. Anand, C. S. Păsăreanu, W. Visser. JPF-SE: a symbolic execution extension to Java PathFinder. In *TACAS'07: Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*. Pp. 134–138. Springer-Verlag, Berlin, Heidelberg, 2007.

[Bal04] F. Balmas. Displaying dependence graphs: a hierarchical approach. *J. Softw. Maint. Evol.* 16(3):151–185, 2004.
doi:http://dx.doi.org/10.1002/smr.291

[BE93] J. Beck, D. Eichmann. Program and interface slicing for reverse engineering. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*. Pp. 509–518. IEEE Computer Society Press, Los Alamitos, CA, USA, 1993.

[BE94] T. Ball, S. G. Eick. Visualizing Program Slices. In *VL*. Pp. 288–295. 1994.

[BHR95] D. Binkley, S. Horwitz, T. Reps. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.* 4(1):3–35, 1995.
doi:http://doi.acm.org/10.1145/201055.201056

[Bin98] D. Binkley. The Application of Program Slicing to Regression Testing. *Information and Software Technology* 40(11-12):583–594, 1998.
citeseer.ist.psu.edu/binkley99application.html

[CH96] J. J. Comuzzi, J. M. Hart. Program Slicing Using Weakest Preconditions. In *FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*. Pp. 557–575. Springer-Verlag, London, UK, 1996.

[CHP10a] D. da Cruz, P. R. Henriques, J. S. Pinto. Contract-Based Slicing. In *Proceedings of the Fourth Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'10)*. 2010. to appear.

[CHP10b] D. da Cruz, P. R. Henriques, J. S. Pinto. Gamaslicer: an Online Laboratory for Program Verification and Analysis. In *Proceedings of the 10th Workshop on Language Descriptions Tools and Applications (LDTA'10)*. 2010.

[CLM95] A. Cimitile, A. D. Lucia, M. Munro. Identifying reusable functions using specification driven program slicing: a case study. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*. P. 124. IEEE Computer Society, Washington, DC, USA, 1995.

[CLM96] A. Cimitile, A. D. Lucia, M. Munro. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance* 8(3):145–178, 1996.
doi:http://dx.doi.org/10.1002/(SICI)1096-908X(199605)8:3¡145::AID-SMR127¿3.3.CO;2-0

[CLYK01] I. S. Chung, W. K. Lee, G. S. Yoon, Y. R. Kwon. Program slicing based on specification. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*. Pp. 605–609. ACM, New York, NY, USA, 2001.
doi:http://doi.acm.org/10.1145/372202.372784

[Gal96] K. Gallagher. Visual Impact Analysis. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*. Pp. 52–58. IEEE Computer Society, Washington, DC, USA, 1996.

[GL91] K. B. Gallagher, J. R. Lyle. Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering* 17(8):751–761, 1991.
citeseer.ist.psu.edu/gallagher91using.html

[GO97] K. Gallagher, L. O'Brien. Reducing Visualization Complexity using Decomposition Slices. In *Proc. Software Visualisation Work.* Pp. 113–118. Department of Computer Science, Flinders University, Adelaide, Australia, 11–12 Dezembro 1997.

[HD95] M. Harman, S. Danicic. Using Program Slicing to Simplify Testing. *Software Testing, Verification & Reliability* 5(3):143–162, 1995.
citeseer.ist.psu.edu/100763.html

[HHF+01] M. Harman, R. Hierons, C. Fox, S. Danicic, J. Howroyd. Pre/Post Conditioned Slicing. *icsm* 00:138, 2001.
doi:http://doi.ieeecomputersociety.org/10.1109/ICSM.2001.972724

[HPR89] S. Horwitz, J. Prins, T. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.* 11(3):345–387, 1989.
doi:http://doi.acm.org/10.1145/65979.65980

[Kin76] J. C. King. Symbolic execution and program testing. *Commun. ACM* 19(7):385–394, 1976.
doi:http://doi.acm.org/10.1145/360248.360252

[Kri04] J. Krinke. Visualization of Program Dependence and Slices. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*. Pp. 168–177. IEEE Computer Society, Washington, DC, USA, 2004.

[Lak93] A. Lakhotia. Rule-based approach to computing module cohesion. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*. Pp. 35–44. IEEE Computer Society Press, Los Alamitos, CA, USA, 1993.

[LC04] G. T. Leavens, Y. Cheon. Design by Contract with JML. 2004.

[LFM96] A. D. Lucia, A. R. Fasolino, M. Munro. Understanding Function Behaviors through Program Slicing. In *Proceedings of the 4th Workshop on Program Comprehension*. Pp. 9–18. 1996.
citeseer.ist.psu.edu/delucia96understanding.html

[M.93] K. M. *Interprocedural dynamic slicing with applications to debugging and testing*. PhD thesis, Linkoping University, Sweden, 1993.

[Mey92] B. Meyer. Applying "Design by Contract". *Computer* 25(10):40–51, 1992.
doi:http://dx.doi.org/10.1109/2.161279

[MS93] N. A. M. Maiden, A. G. Sutcliffe. People-oriented Software Reuse: the Very Thought. In *Advances in Software Reuse - Second International Workshop on Software Reusability*. Pp. 176–185. IEEE Computer Society Press, 1993.

[OT93] L. Ottenstein, J. Thuss. Slice based metrics for estimating cohesion. 1993.
citeseer.ist.psu.edu/ott93slice.html

[San95] G. Sander. Graph Layout through the VCG Tool. In *GD '94: Proceedings of the DIMACS International Workshop on Graph Drawing*. Pp. 194–205. Springer-Verlag, London, UK, 1995.

[SS07] S. G. Shiva, L. A. Shala. Software Reuse: Research and Practice. In *ITNG*. Pp. 603–609. IEEE Computer Society, 2007.
http://dblp.uni-trier.de/db/conf/itng/itng2007.html#ShivaS07

[SV03] K. Sherif, A. Vinze. Barriers to adoption of software reuse a qualitative study. *Inf. Manage.* 41(2):159–175, 2003.
doi:http://dx.doi.org/10.1016/S0378-7206(03)00045-4

[Wei79] M. D. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, Ann Arbor, MI, USA, 1979.

[WL86] M. Weiser, J. Lyle. Experiments on slicing-based debugging aids. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*. Pp. 187–197. Ablex Publishing Corp., Norwood, NJ, USA, 1986.

GUI Inspection from Source Code Analysis

João Carlos Silva [1,2] **José Creissac** [1] **João Saraiva** [1]

[1]Departamento de Informática, Universidade do Minho, Braga, Portugal
[2]Departamento de Tecnologia, Instituto Politécnico do Cávado e do Ave, Barcelos, Portugal

Abstract: Graphical user interfaces (GUIs) are critical components of todays software. Given their increased relevance, correctness and usability of GUIs are becoming essential. This paper describes the latest results in the development of our tool to reverse engineer the GUI layer of interactive computing systems. We use static analysis techniques to generate models of the user interface behaviour from source code. Models help in graphical user interface inspection by allowing designers to concentrate on its more important aspects. One particular type of model that the tool is able to generate is state machines. The paper shows how graph theory can be useful when applied to these models. A number of metrics and algorithms are used in the analysis of aspects of the user interface's quality. The ultimate goal of the tool is to enable analysis of interactive system through GUIs source code inspection.

Keywords: Source Code, Reverse Engineering, Graphical User Interface, Metrics, Properties

1 Introduction

Typical WIMP-style (Windows, Icon, Mouse, and Pointer) user interfaces consist of a hierarchy of graphical widgets (buttons, menus, textfields, etc) creating a front-end to software systems. An event-based programming model is used to link the graphical objects to the rest of the system's implementation. Each widget has a fixed set of properties and at any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI. Users interact with the system by performing actions on the graphical user interface widgets. These, in turn, generate events at the software level, which are handled by appropriate listener methods.

In brief, and from a user's perspective, graphical user interfaces accept as input a pre-defined set of user-generated events, and produce graphical output. From the programmers perspective, as user interfaces grow in size and complexity, they become a tangle of object and listener methods, usually all having access to a common global state. Considering that the user interface layer of interactive systems is typically the one most prone to suffer changes, due to changed requirements and added features, maintaining the user interface code can become a complex and error prone task. Integrated development environments (IDEs), while helpful in that they enable the graphical definition of the interface, are limited when it comes to the definition of the behavior of the interface

A source code analysis tool can minimize the time necessary by a developer to understand and evaluate a system. In this paper we present GUISurfer, a static analysis based retargetable framework for GUI-based applications analysis from source code. In previous papers [SCS06a,

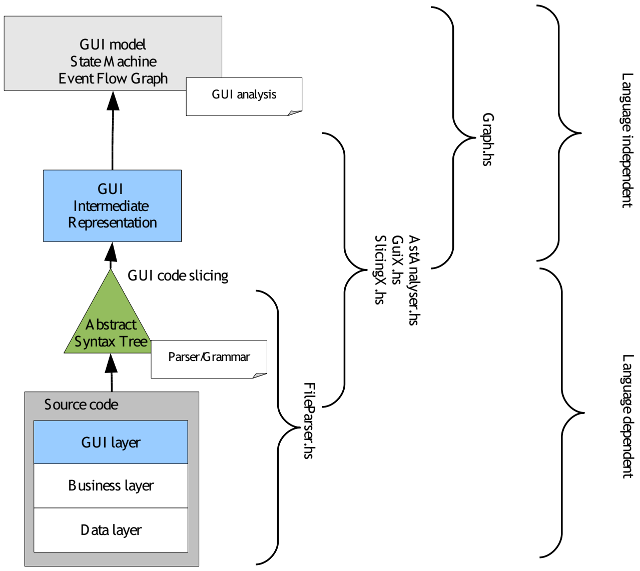


Figure 1: GUISurfer Architecture and Retargetability

SCS06b, SCS09] we have explored the applicability of slicing techniques [Tip95] to our reverse engineering needs, and developed the building blocks for the approach. In this paper we explore the integration of analysis techniques into the approach, in order to reason about the GUI models.

2 GUISurfer tool

GUISurfer's goal is to be able to extract a range of models from source code. In the present context we focus on finite state models that represent GUI behaviour. That is, when can a particular GUI event occur, which are the related conditions, which system actions are executed, or which GUI state is generated next. We choose this type of model in order to be able to reason about and test the dialogue supported by a given GUI implementation.

Figure 1 presents the architecture of the GUISurfer tool. *GUIsurfer* is composed by three tools: FileParser, AstAnalyser, and Graph. These tools are configurable through command line parameters. Below we outline some of the more important parameters for each tool.

The FileParser tool is language dependent and is used to parse a particular source code file. For example, the command *FileParser Login.java* allows us to parse a particular *Login* Java class. As a result, we obtain its AST.

The AstAnalyser tool is another language dependent tool used to slice an abstract syntax tree, considering only its graphical user interface layer. Part of this tool is easily retargetable, however most of the tool needs to be rewritten to consider another particular programming language.

The AstAnalyser tool is composed of a slicing library, containing a generic set of traversal functions that traverse any AST. This tool must be used with three arguments, i.e. the abstract syntax tree, the entry point in source code (e.g., the main method for Java source code), and a list with all widgets to consider during the GUI slicing process. The command *AstAnalyser Login.java.ast main JButton* lets us extract the GUI layer from *Login.java*'s abstract syntax tree, starting the slice process at the *main* method, and extracting only *JButton* related data. Executing the command generates two files *initState.gui* and *eventsFromInitState.gui* which contain the initial state and possible events from the initial states, respectively.

Finally, the Graph tool is language independent and receives as arguments the *initState.gui* and *eventsFromInitState.gui* files, and generates several metadata files with events, conditions, actions, and states extracted form source code. Each of these types of data is related to a particular fragment from the AST. Further important outputs generated by the Graph tool are the *GuiModel.hs* and *GuiModelFull.hs* files. These are GUI specifications written in the Haskell programming language. These specifications define the GUI layer mapping events/conditions to actions. Finally, this last tool allows us also to generate several visual models through the GraphViz tool, such as state machines, behavioral graph, etc.

3 GUI Inspection from source code

The evaluation of an user interface is a multifaceted problem. Besides the quality of the code by itself, we have to consider the user reaction to the interface. This involves issues such as satisfaction, learnability, and efficiency. The first item describes the users satisfaction with the systems. Learnability refers to the effort users make to learn how to use the application. Efficiency refers to how efficient the user can be when performing a task using the application.

Software metrics aim to measure software aspects, such as source lines of code, functions invocations, etc. By calculating metrics over the behavioral models produced by GUISurfer, we aim to acquire relevant knowledge about the dialogue induced by the interface, and, as a consequence, about how users might react to it (c.f. [TG08]). In this section we describe several kinds of inspections making use of metrics.

The analysis of source code can provide a mean to guide development and to certificate software. For that purpose adequate metrics must be specified and calculated. Metrics can be divided into two groups: internal and external [ISO99].

External metrics are defined in relation to running software. In what concerns GUIs, external metrics can be used as usability indicators. They are often associated with the following attributes [Nie93]:

- Easy to learn: The user can do desired tasks easily without previous knowledge;
- Efficient to use: The user reaches a high productivity level.
- Easy to remember: The re-utilization of the system is possible without a high level of effort.
- Few errors: Errors are made hardly by the users and the system permits to recover from them.
- Pleasant to use: The users are satisfied with the use of the system.

However, the values for these metrics are not obtainable from source code analysis, rather through users' feedback.

Internal metrics are obtained by source code analysis, and provide information to improve software development. A number of authors has looked at the relation between internal metrics and GUI quality.

Stamelos et al. [SAOB02] used the Logiscope[1] tool to calculate values of selected metrics in order to study the quality of Open Source code. Ten different metrics were used. The results enable evaluation of each function against four basic criteria: testability, simplicity, readability and self-descriptiveness. While the GUI layer was not specifically targeted in the analysis, the results indicated a negative correlation between component size and user satisfaction with the software.

Yoon and Yoon [YY07] developed quantitative metrics to support decision making during the GUI design process. Their goal was to quantify the usability attributes of interaction design. Three internal metrics were proposed and defined as numerical values: complexity, inefficiency and incongruity. The authors expect that these metrics can be used to reduce the development cost of user interaction.

While the above approaches focus on calculating metrics over the code, Thimbleby and Gow [TG08] calculate them over a model capturing the behavior of the application. Using graph theory they analyze metrics related to the users' ability to use the interface (e.g., strong connectedness ensure no part of the interface ever becomes unreachable), the cost of erroneous actions (e.g., calculating the cost of undoing an action), or the knowledge needed to use the system (e.g., the minimum cut identifies the set of actions that the user must know in order to to be locked out of parts of the interface).

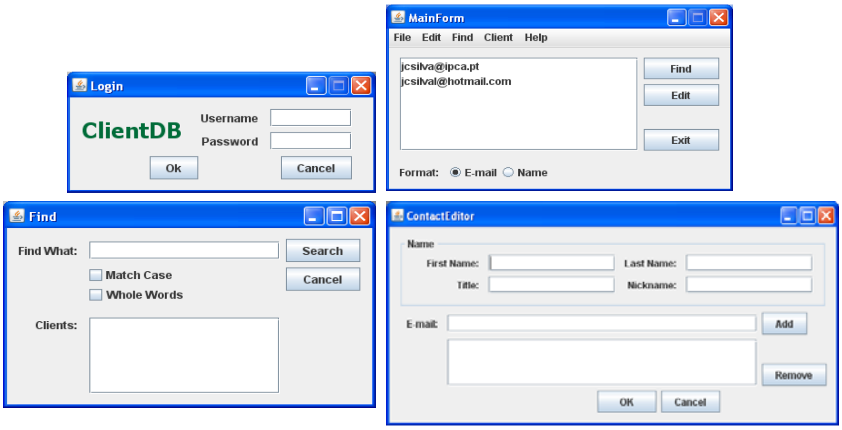
In a sense, by calculating the metrics over a model capturing GUI relevant information instead of over the code, the knowledge gained becomes closer to the type of knowledge obtained from external metrics. While Thimbleby and Gow manually develop their models from inspections of the running software/devices, an analogous approach can be carried out analyzing the models generated by GUISurfer. Indeed, by coupling this type of analysis with GUISurfer, we are able to obtain the knowledge directly from source code.

4 An Agenda application

Throughout the paper we will use a Java/Swing interactive application as a running example. This application consist of an agenda of contacts: it allows users to perform the usual actions of adding, removing and editing contacts. Furthermore, it also allows users to find a contact through its name.

The interactive application consists of four windows, namely: *Login*, *MainForm*, *Find* and *ContactEditor*, as shown in Figure 2. The initial *Login* window (Figure 2, top-left) is used to control users' access to the agenda. Thus, a login and password pair has to be introduced by the user. If the user introduces a valid login/password pair, and presses the `Ok` button, then the login window closes and the main window of the application is displayed. On the contrary, if the user introduces an invalid login/password pair, then the input fields are cleared, a warning message is

[1] http://www-01.ibm.com/software/awdtools/logiscope/



jcsilva@ipca.pt
jcsilval@hotmail.com

Format: ● E-mail ○ Name

Figure 2: A Java/Swing application

produced and the login window continues to be displayed. By pressing the *Cancel* button in the *Login* window, the user exits the application.

The Java fragment defining the action performed when the `Ok` button is pressed is as follows:

```
private void OkActionPerformed(...)
{if (isValid(user.getText(),pass.getText()))
 {new MainForm().setVisible(true);
  this.dispose();}
 else javax.swing.JOptionPane.showMessageDialog
          (this,"User/Pass not valid","Login",0);
}
```

where the method *isValid* tests the username/password pair inserted by the user.

Authorized users can use the main window (Figure 2, top-right) to find and edit contacts (c.f., *Find* and *Edit* buttons). By pressing the *Find* button in the main window, the user opens the *Find* window (Figure 2, bottom-left). This window is used to search and obtain a particular contact's data from his name. By pressing the *Edit* button in the main window, the user opens the *ContactEditor* window (Figure 2, bottom-right). This last window allows the editing of a contact's data, such as name, nickname, e-mails, etc. The *Add* and *Remove* buttons enable editing the e-mail addresses' list of the contact. If there are no e-mails in the list then the *Remove* button is automatically disabled.

Until now, we have informally described the (behavioral) model of our interactive application. Such descriptions, however, can be ambiguous and often lead to different interpretation of what the application should do. In order to unambiguously and rigorously define an application, we can use a formal model. Moreover, by using a formal model to define the interactive application, we can use techniques to manipulate and inspect such application.

Figure 3 shows a formal model to specify the behavior of our running example: a graph. A graph is a mathematical abstraction and consists of a set of vertices, and a set of edges. Each edge connects two vertices in the graph. In other words, a graph is a pair (V,E), where V is a finite set and E is a binary relation on V. V is called a vertex set whose elements are called vertices. E is a

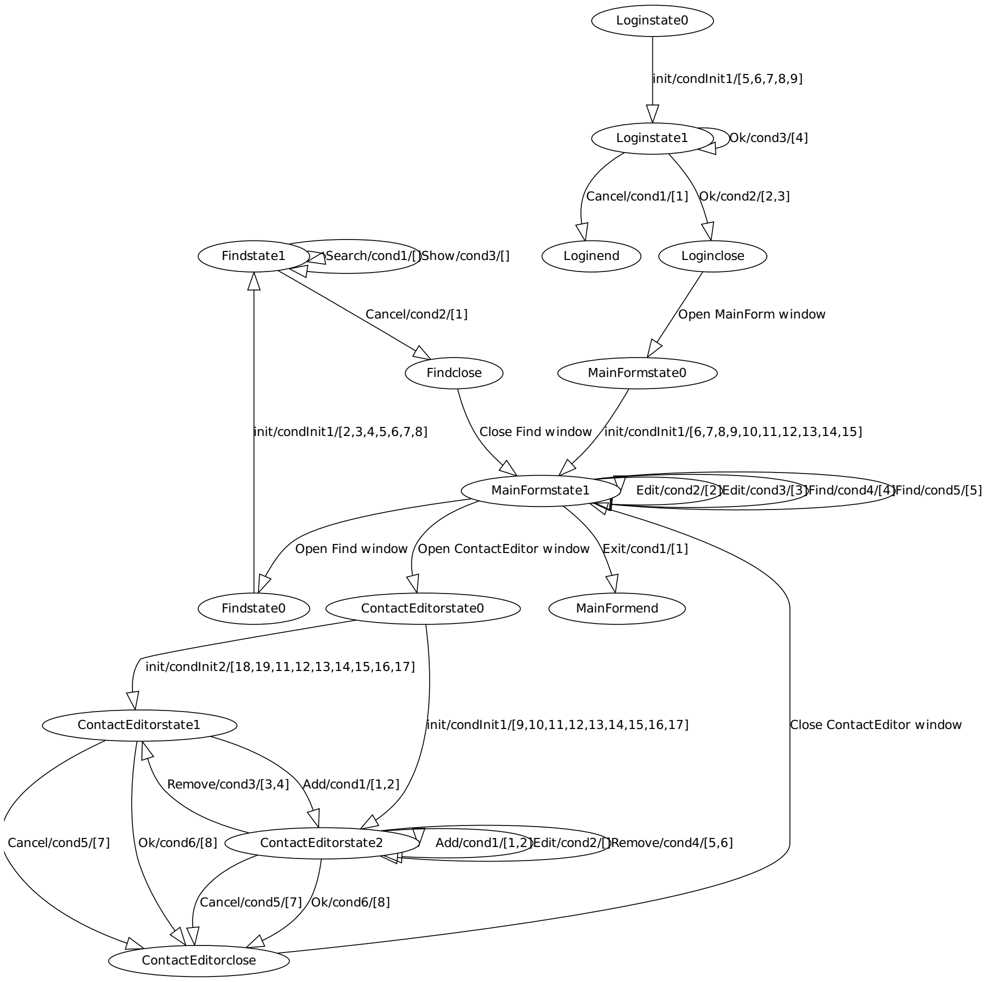


Figure 3: Agenda's behavior graph

collection of edges, where an edge is a pair (u,v) with u,v in V. Graphs are directed or undirected. In a directed graph, edges are ordered pairs, connecting a source vertex to a target vertex. In an undirected graph edges are unordered pairs of two vertices.

If some edge (u,v) is in graph, then vertex v is said to be adjacent to vertex u. In a directed graph, edge (u,v) is an out-edge of vertex u and an in-edge of vertex v. The number of out-edges of a vertex is its out-degree, and the number of in-edges is its in-degree.

A path is a sequence of edges in a graph such that the target vertex of each edge is the source vertex of the next edge in the sequence. If there is a path starting at vertex u and ending at vertex v we say that v is reachable from u.

Graphs are a commonly used to represent user interfaces. Vertices represent the possible GUI states, and the transitions between vertices (edges) define the events associated to the GUI objects.

The model in figure 3 was automatically extracted by *GUIsurfer*. Associated to each edge there is a triplet representing the event that triggers the transition, a guard on that event (here represented by a label identifying the condition being used), and a list of interactive actions executed when the event is selected (each action is represented by a unique identifier which is related to the respective source code).

Using this model it becomes possible to reason about characteristics of the interaction between users and the agenda application.

5 GUI Inspection through Graph Theory

This section describes some examples of analysis performed on the Agenda application's behavioral graph (cf. figure 3) from the previous section. We make use of Graph-Tool for the manipulation and statistical analysis of the graph.

5.1 Graph-tool

Graph-tool is an efficient python module for manipulation and statistical analysis of graphs (cf. http://projects.forked.de/graph-tool/). It allows for the easy creation and manipulation of both directed or undirected graphs. Arbitrary information can be associated to the vertices, edges or even the graph itself, by means of property maps.

Graph-tool implements all sorts of algorithms, statistics and metrics over graphs, such as degree/property histogram, combined degree/property histogram, vertex-vertex correlations, assortativity, average vertex-vertex shortest distance, isomorphism, minimum spanning tree, connected components, dominator tree, maximum flow, clustering coefficients, motif statistics, communities, centrality measures. Now we will consider the graph described in figure 4 (automatically obtained from figure 3) where all vertices and edges are labeled with unique identifiers.

5.2 GUI Metrics

To illustrate the analysis, we will consider three metrics: Shortest distance between vertices, Pagerank and Betweeness.

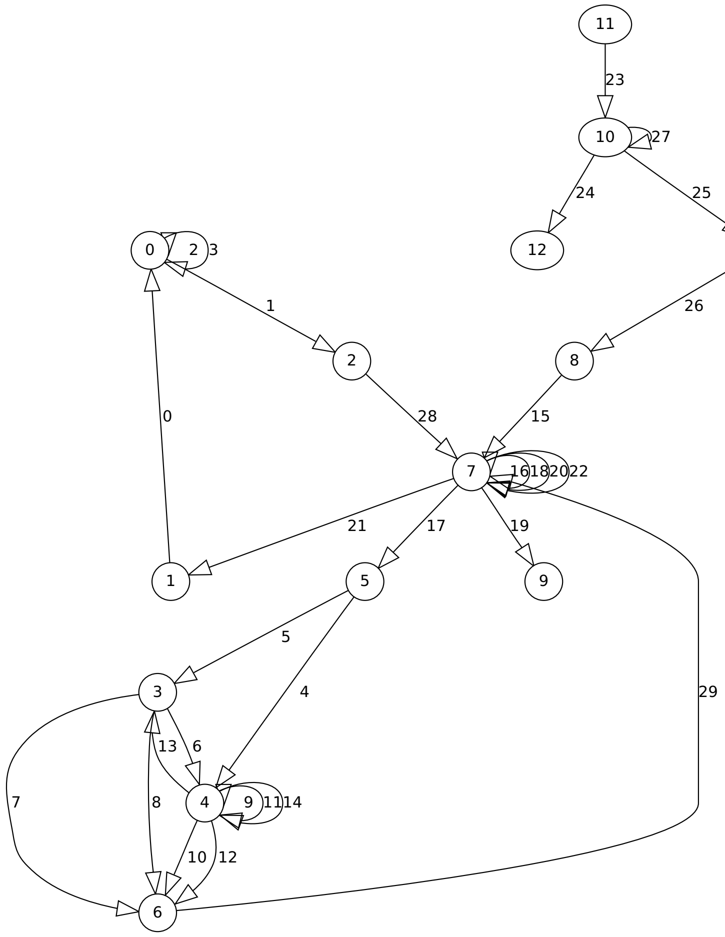


Figure 4: Agenda’s behavior graph (numbered)

5.2.1 Shortest Distance

The Graph-Tool enables us to calculate the shortest path between two vertices. As examples the obtained results for the shortest path between vertices 11 and 6 is (cf. figure 4):

```
shortest path vertices: [’11’,’10’,’13’,’8’,’7’,’5’,’4’,’6’]
shortest path edges:
 [’(11,10)’,’(10,13)’,’(13,8)’,’(8,7)’,’(7,5)’,’(5,4)’,’(4,6)’]
```

We obtain the vertices sequence from vertice 11 to vertice 6. And we have also access to the egdes sequence. This is usefull to calculate the number of steps to execute a particular task.

Now let us consider another inspection. The next result gives us the shortest distance (minimum number of egdes) from the Login window (vertice 11) to all other vertices. Each value gives the distance from vertice 11 to a particular target vertice. The index of the value in the sequence correspond to the vertice identifier. As example the first value is the shortest distance from vertice 11 to vertice 0, which is 6 edges long.

```
shortest distance from Login
[6 5 7 6 6 5 7 4 3 5 1 0 2 2]
```

Another example makes use of MainForm (vertice 7) as starting point. Negative values (-1) indicate that there are no paths from Mainform to those vertices.

```
shortest distance from MainForm
[2 1 3 2 2 1 3 0 -1 1 -1 -1 -1 -1]
```

This metrics are useful to analyse the complexity of an interactive application's user interface. Higher values represent complex tasks while lower values are applications with simple tasks. The example also shows that they can be used to detect parts of the interface that can become unavailable. In this case, there is no way to go back to the login window once the Main window is displayed. The application must be quit.

This metrics can be used to calculate the center of a graph. The center of a graph is the set of all vertices A where the greatest distance to other vertices B is minimal. The vertices in the center are called central points. Thus vertices in the center minimize the maximal distance from other points in the graph.

Finding the center of a graph is useful in GUI applications where the goal is to minimize the steps to execute a particular task (i.e. edges between two points). For example, placing the main window of an interactive system at a central point reduces the number of steps an user has to execute to accomplish tasks.

5.2.2 Pagerank

PageRank is a distribution used to represent the probability that a person randomly clicking on links will arrive at any particular page [Ber05]. A probability is expressed as a numeric value between 0 and 1. A 0.5 probability is commonly expressed as a ”50% chance” of something happening.

PageRank is a link analysis algorithm, used by the Google Internet search engine that assigns a numerical weighting to each element of a hyperlinked set of documents. The main objective is to measure their relative importance.

This same algorithm could be applied to our GUI's behavioral graphs. Figure 5 gives pagerank for each Agenda vertices. The size of a vertex corresponds to its importance within the overall application behavior. This metric is useful, for example, to analyze whether complexity is well distributed along the application behavior. In this case, the Main window is clearly a central point in the interaction.

5.2.3 Betweenness

Betweenness is a centrality measure of a vertex or a edge within a graph[Sa09]. Vertices that occur on many shortest paths between other vertices have higher betweenness than those that do not. Similar to vertices betweenness centrality, edge betweenness centrality is related to shortest

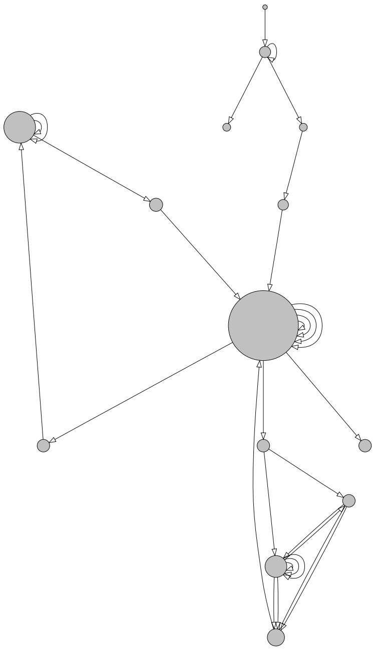


Figure 5: Agenda's pagerank results

path between two vertices. Edges that occur on many shortest paths between vertices have higher egde betweeness.

Figure 6 describes betweeness values as a visual form for each Agenda vertices and egdes. Highest betweeness edges values are related with largest egdes.

The Main window has the highest betweenness, meaning it acts as a hub from where different parts of the interface can be reached. Clearly it will be a central point in the interaction.

5.2.4 Cyclomatic Complexity

Another important metric is cyclomatic complexity which aims to measures the total number of decision logic in an application [J.76]. It is used to give the number of tests for software and to keep software reliable, testable, and manageable. Cyclomatic complexity is based entirely on the structure of software's control flow graph and is defined as $M = E - V + 2P$ (considering single exit statement) where E is the number of edges, V is the number of vertices and P is the number of connected components.

Considering the figure 5 where edges represent decision logic in the Agenda GUI layer, the GUI's overall cyclomatic complexity is 18. In other hand, each Agenda's window has a cyclomatic complexity less or equal than 10. In applications there are many good reasons to limit cyclomatic complexity. Complex structures are more prone to error, are harder to analyse, are harder to test, and are harder to maintain. The same reasons could be applied to user interfaces. McCabe proposed a limit of 10 for functions's code, but limits as high as 15 have been used

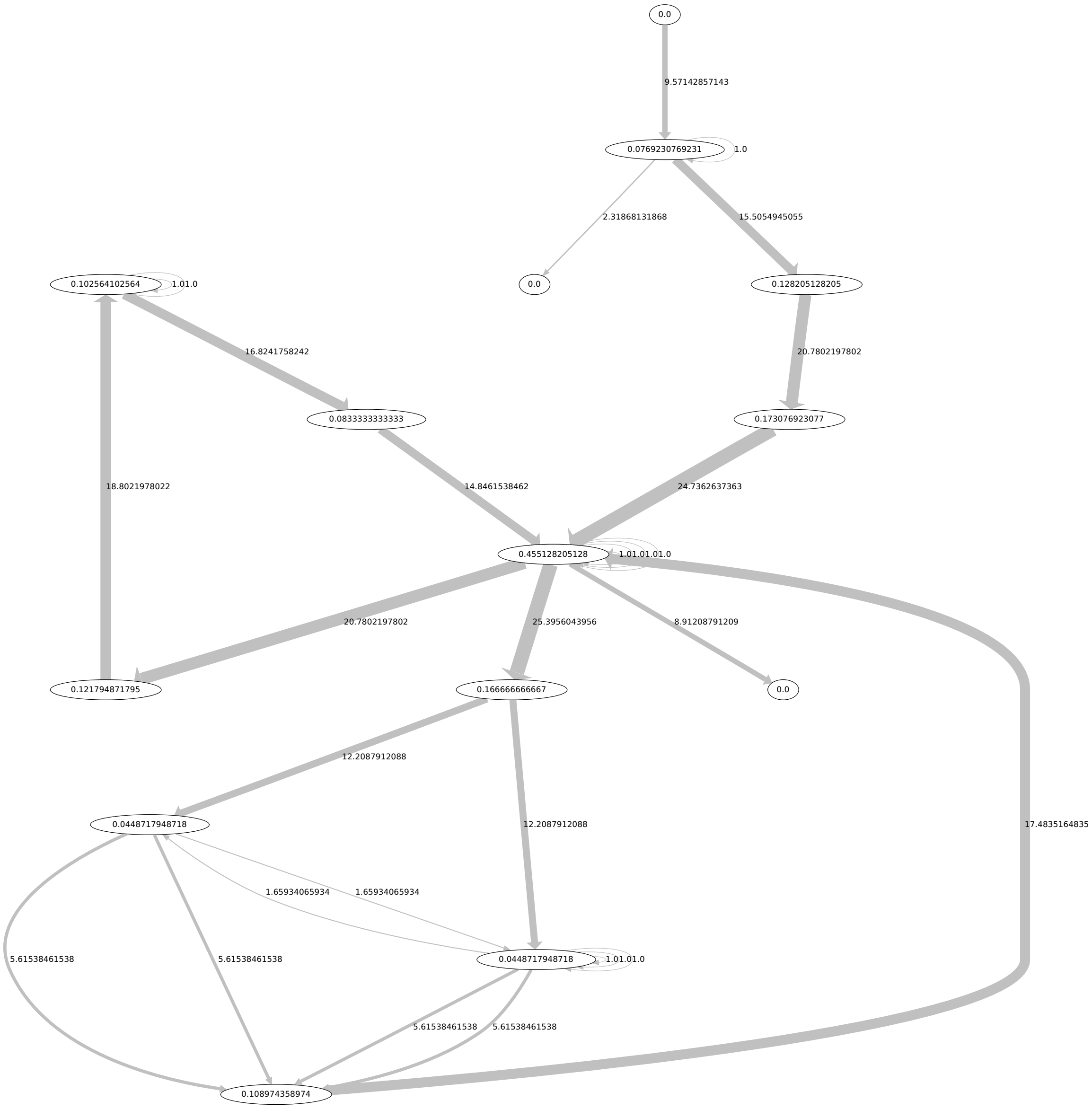


Figure 6: Agenda’s betweeness values

successfully as well. McCabe suggest limit greater than 10 for projects that have operational advantages over typical projects, for example formal design. User interfaces can apply the same limits of complexity, i.e. each window behavior complexity could be limited to a particular cyclomatic complexity.

5.3 GUI Test Cases Generation

Software testing is very important since it enables to evaluate a system by manual or automatic means and verify that it satisfies specified properties or identify differences between expected and actual results. Most approaches to software testing focus on the computational/algorithmic aspects of the systems. In this section we use the models generated by GUIsurfer like figure 3 in order to follow a model-based testing approach for GUI. Software testing is usally divided in two phases: test cases generation and properties verification. In this section we present our approach to these two tasks.

5.3.1 Related Work

The need for system reliability is the basis of research into the problem of GUI testing. The research aims to validate their correct functioning and to discover aspects of their behavior.

Having generated representations of GUI behavior, we are ready to define the coverage criteria for events and states. Considering test cases generation, some user behaviors will be more likely than other. Consequently if test cases are generated randomly then there is no guarantee that *interesting* behaviors will be tested.

To address this, several alternatives to generating test cases are proposed in the literature. As example finite state machine (FSM) are used to model system and to generate test cases [SL89, Ura92]. Test cases are generated from FSM-based specifications through several methods. These methods are: the Transition Tour (T) method; the Distinguishing Sequence (D) method; the Characterizing Set (W) method; the Unique Input/Output Sequence (UIO) method; the Single UIO (SUIO) method; and the Multiple UIO (MUIO) method.

All these methods need fully specified finite state machines, i.e. for each state and for each event, a unique transition must be defined (in some cases, null transitions). Fully specified state machines have the same set of inputs for each state. However, many graphical user interfaces have different set of inputs for their states. One *tedious* solution is to add transitions that point back to the same state with NULL output. Shehady has defined an alternative solution (VFSM - variable finite state machine) which is to have a conversion algorithm automatically add transitions and NULL ouputs when needed to fully specified finite state machine [SS97].

Another alternative is the use of graphs. Graphs have been widely used to model systems in diverse areas. Memon's approach about coverage criteria for GUI testing make use of an event flow graph for GUI's behavioral representation [MSP01]. The paper describes a methodology for generating test cases from GUI behavior graph-based specifications. Coverage criteria are presented to help determine wheter a GUI has been adequately tested.

Ping Li describes an another approach to testing GUI systems in [LHRM07]. In the proposed approach, GUI systems are divided into two abstract tiers: the component tier and the system tier. On the component tier, a flow graph is created for each GUI component, describing (relationships between the pre-conditions, event sequences and post-conditions). On the system tier, the components are integrated resulting in a view of the entire system. Finally, tests on the system tier analyse the interactions between the components.

5.3.2 Coverage Criteria

Because our GUI's model representation can be viewed as a graph, we applied the Memon approach about coverage criteria for GUI testing [MSP01]. In this section, we define several coverage criteria for events and their interactions following Memon's approach. We first formally define an event sequence, which is used to describe all the coverage criteria.

An event-sequence is a tuple $< el, e2, e3, ..., en >$ where e_i is a particular event which can be executed after event e_{i-1}, $2 \leq i \leq n$.

Next we present three coverage criterium applied to GUI behavior graph-based specifications.

- Event Coverage: The event coverage criterion enables to capture a set of event-sequences considering all possible events. The event coverage criterion is satisfied if and only if for any event e, there is at least one event sequence es such that es contains e.

- State Coverage: State coverage requires that each state is reached at least once, i.e. for any state s there is at least one event-sequence es such that state s is reached in es.

- Length-n-Event-sequence Coverage: Within GUI systems, the behavior of events may change when executed in different contexts. The length-n-event-sequence coverage criterium define the set of event-sequences which contains all event-sequences of length equal to n. As example the length-n-event-sequence coverage criterium applied to the Agenda's behavioral model in figure 3 returns the following number of test cases:

Length-n	1	2	3	4	5	6	7	8	9	10
Total	1	3	4	10	40	190	940	4690	23440	117190

Table 1: Total number of event-sequences for n event-sequence length

The result of this criterium show that the total number of event sequences grows with increasing length. The large number of event sequences turns difficult to test a GUI for all possible event sequences. Memon proposes to assign priorities to each event-sequence and first test event-sequences with higher priorities. As example, event-sequences related with the main window could have a higher priority since they may be used more times.

A test suite is a set of input sequences starting from the initial state of the machine. Intuitively, if a test suite satisfies event coverage, it also satisfy state coverage. In other hand event coverage and state coverage are special case length-n-event-sequence coverage.

In some cases, it could be interessant to consider the overall behavior of the GUI. This perspective can be achieved trough a unique path reaching all possible states (or all possible events) between a start state and a final state. These particular test cases can be generated through Chinese Postman Tour and Travelling Salesman Problem algorithms, described in next two sections.

5.3.3 Chinese Postman Tour

The background of the Chinese Postman Problem is about a chinese postman who wishes to travel along every road in a city in order to deliver letters, while traveling the least possible distance. Solving the problem corresponds to finding the shortest route in a graph in which each edge is traversed at least once [Thi03, PC05]. If the path must get back to the starting point, the problem is said to be closed. If it does not need to go back, it is called an open problem.

The algorithm to solve the open problem can be used to generate minimal sequences of user actions between pairs of states, each sequence including all possible users actions in the interface. These sequences can then be used as test cases for testing the interface against defined properties.

The length of the optimal path for the closed problem acts as a measure of the user interface's complexity [Thi03]. If we consider weighted graphs, and assign weights to the transitions that correspond to the time users are expected to take performing the corresponding actions, then the optimal path for closed problem might be used to calculate how long a user takes to explore an entire application.

5.3.4 Travelling Salesman Problem

The Travelling Salesman Problem (TSP) considers a salesman whose task is to find a shortest possible tour that visits each city in a region exactly once. Even though the problem is computationally difficult, a large number of exact methods and heuristics have been proposed, making it possible to solve instances with tens of thousands of cities.

While in the Chinese Postman Problem the goal is to traverse every edge at least once, in the Travelling Salesman Problem the goal is to visit every node. There is no need to use all edges in the graph. Paths produced as a solution to this problem will guarantee that all window states will be visited by the user, while keeping user actions to a minimum.

5.3.5 Properties Verification

The reverse engineering approach described in this paper allows us to extract GUI behaviour model as graphs. Using these graphs, we are able to test GUI properties [Bel01, Bum96, Pat95]. Previous sections define alternatives to generate particular test cases. This section describes a study enabling us to validate random GUI test cases. To test GUI properties, we make use of the *QuickCheck haskell* library tool. QuickCheck [CH00] is a tool for testing programs automatically. The programmer provides a specification of the program and properties to satisfy. Then QuickCheck tests the properties in a large number of randomly generated cases. Specifications are expressed in *Haskell*, using combinators defined in the QuickCheck library. QuickCheck provides combinators to define properties, observe the distribution of test data, and define test data generators.

Considering the Agenda application above, and its GUI behavior graph expressed as a Haskell specification, we could now generate test cases now and write some properties and test them through the *QuickCheck* tool.

Test cases could be obtained through algorithms described in above subsections 5.3.2, 5.3.3 and 5.3.4. Each algorithm defines a particular view of the user interaction with the analysed interactive system.

As example, through the Agenda's test cases, we can define a property to check that from all states its possible to reach the central state with biggest pagerank value, i.e. state number 7 in figure 4. The respective QuickCheck property could be defined as follows:

```
rule1 (N (a,b)) =
     classify ((length b)<=10) "events sequence length: <=10" $
     classify ((length b)>10) "events sequence length: >10" $
     (intersect [15,16,18,20,22,28,29] b) /= []
```

Parameters *a* and *b* defines a particular test case. The first parameter contains an events's identifiers sequence. The second parameter contains respective conditions's identifier for each event. Values 15, 16, 18, 20, 22, 28 and 29 refer to egdes identifiers from figure 4 which have central state number 7 as target. The property enables to check if all test cases contains at least one of these edges.

The number of randomly generated test cases and events length are specified by the *GUIsurfer* user. Each random case is a sequence of valid events associated with their conditions.

6 Discussion

GUISurfer makes possible high-level graphical representation of thousand of lines of code. The process is almost automatic and enables reasoning over the interactive layer of computing systems.

A particular emphasis is being placed on developing tools that are, as much as possible, language independent. Through the use of generic programming techniques, the developed tool aims at being retargetable to different user interface programming toolkits and languages. At this time, the tool supports (to varying degrees) the reverse-engineering of Java code, either with the Swing or the GWT (Google Web Toolkit) toolkits, and of Haskell code, using the wxHaskell GUI library. Originally the tool was developed for Java/Swing. The wxHaskell and GWT retargets have highlighted successes and problems with the initial approach. The amount adaptation and the time it took to code are distinct. The adaptation to GWT was easier because it exploits the same parser. The adaptation to wxHaskell was more complex as the programming paradigm is different, i.e. functional.

Results show the reverse engineering approach adopted is useful but there are still some limitations. One relates to the focus on event listeners for discrete events. This means the approach is not able to deal with continuous media and synchronization/timing constraints among objects. Another has to due with layout management issues. GUISurfer cannot extract, for example, information about overlapping windows since this must be determined at run time. Thus, we cannot find out in a static way whether important information for the user might be obscured by other parts of the interface. A third issue relates to the fact that generated models reflect what was programmed as opposed to what was designed. Hence, if the source code does the wrong thing, static analysis alone is unlikely to help because it is unable to know what the intended outcome was. For example, if an action is intended to insert a result into a text box, but input is sent to another instead. However, if the design model is available, GUISurfer can be used to extract a model of the implemented system, and a comparison between the two can be carried out.

Using GUISurfer, programmers are able to reason about the interaction between users and a given system at a higher level of abstraction than that of code. The generated graphs are amenable to analysis via model checking (c.f. [CH09]). Here however, we have explored alternative, lighter weight approaches.

Considering that the graphs generated by the reverse engineering process are representations of the interaction between users and system, we have explored how metrics defined over those graphs can be used to obtain relevant information about the interaction. This means that we are able to analyze the quality of the user interface, from the users perspective, without having to resort to external metrics which would imply testing the system with real users, with all the costs that process carries. Additionally, we are exploring the possibility of analyzing the graphs via a testing approach, and how best to generate test cases.

It must be noted that, while the approach enables us to analyze aspects of user interface quality without resorting to human test subjects, the goal is not to replace user testing. Ultimately, only user testing will provide factual evidence of the usability of an user interface. The possibility of performing the type of analysis we are describing, however, will help in gaining a deeper understanding of a given user interface. This will promote the identification of potential problems in the interface, and support the comparison of different interfaces, complementing and minimizing the need to resort to user testing.

Similarly, while the proposed metrics and analysis relate to the user interface that can be inferred from the code, the approach is not proposed as an alternative to actual code analysis. Metrics related to the quality of the code are relevant, and indeed GUISurfer is also able to generate models that capture information about the code itself. Again, we see the proposed approach as complementary to that style of analysis.

7 Conclusion

In what concerns user interface development, two perspectives on quality can be considered. Users, on the one hand, are typically interested on what can be called external quality: the quality of the interaction between users and system. Programmers, on the other hand, are typically more focused on the quality attributes of the code being produced.

This work is an approach to bridging this gap by allowing us to reason about GUI models from source code. We described GUI models extracted automatically from the code, and presented a methodology to reason about the user interface model. A number of metrics over the graphs representing the user interface were investigated. Some initial thoughts on testing the graph against desirable properties of the interface were also put forward.

A number of issues still needs addressing. In the example used throughout the paper, only one windows could be active at any given time (i.e., windows were modal). When non-modal windows are considered (i.e., when users are able to freely move between open application windows), nodes in the graph come to represents sets of open windows instead of a single active window. This creates problems with the interpretation of metrics that need further consideration. The problem is exacerbated when multiple windows of a given type are allowed (e.g., multiple editing windows).

Coverage criteria provide an objective measure of test quality. We plan to include coverage criteria to help determine whether a GUI has been adequately tested. These coverage criteria use events and event sequences to specify a measure of test adequacy. Since the total number of permutations of event and condition sequences in any GUI is extremely large, the GUI's hierarchical structure must be exploited to identify the important event sequences to be tested.

This work presents an approach to the reverse engineering of GUI applications. Models enable us to reason about both metrics of the design, and the quality of the implementation of that design. Our objective has been to investigate the feasibility of the approach. We believe this style of approach can feel a gap between the analysis of code quality via the use of metrics or other techniques, and usability analysis performed on a running system with actual users.

Acknowledgements: This work is supported by the Portuguese Research Foundation (FCT) under contracts: PTDC/EIA-CCO/108995/2008, PTDC/EIA-CCO/108613/2008, and SFRH/BSAD/782/2008.

Bibliography

[Bel01] F. Belli. Finite state testing and analysis of graphical user interfaces. In *Proceedings.of the 12th International Symposium on Software Reliability Engineering, ISSRE 2001*. Pp. 34–42. IEEE, November 2001.

[Ber05] P. Berkhin. A survey on pagerank computing. *Internet Mathematics* 2:73–120, 2005.

[Bum96] P. Bumbulis. *Combining Formal Techniques and Prototyping in User Interface Construction and Verification*. PhD thesis, University of Waterloo, 1996.

[CH00] K. Claessen, J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of International Conference on Functional Programming (ICFP), ACM SIGPLAN, 2000*. 2000.

[CH09] J. C. Campos, M. D. Harrison. Interaction engineering using the IVY tool. In *ACM Symposium on Engineering Interactive Computing Systems (EICS 2009)*. Pp. 35–44. ACM, New York, NY, USA, 2009.

[ISO99] ISO/IEC. Software Products Evaluation. 1999. DIS 14598-1.

[J.76] M. T. J. A Complexity Measure. *Intern. J. Syst. Sci.* 2(4):308, 1976.

[LHRM07] P. Li, T. Huynh, M. Reformat, J. Miller. A practical approach to testing GUI systems. *Empirical Softw. Engg.* 12(4):331–357, 2007.

[MSP01] A. M. Memon, M. L. Soffa, M. E. Pollack. Coverage criteria for GUI testing. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. Pp. 256–267. ACM Press, New York, NY, USA, 2001.

[Nie93] J. Nielsen. *Usability Engineering*. Academic Press, San Diego, CA, 1993.

[Pat95] F. D. Paternò. *A Method for Formal Specification and Verification of Interactive Systems*. PhD thesis, Department of Computer Science, University of York, 1995. Available as Technical Report YCST 96/03.

[PC05] W. L. Pearn, W. C. Chiu. Approximate solutions for the maximum benefit Chinese postman problem. *Intern. J. Syst. Sci.* 36(13):815–822, 2005.

[Sa09] S. Y. Shan, et al. Fast Centrality Approximation in Modular Networks. 2009.

[SAOB02] I. Stamelos, L. Angelis, A. Oikonomou, G. L. Bleris. Code quality analysis in open source software development. *Information Systems Journal* 12:43–60, 2002.

[SCS06a] J. Silva, J. C. Campos, J. Saraiva. Combining Formal Methods and Functional Strategies Regarding the Reverse Engineering of Interactive Applications. In *Interactive Systems, Design, Specifications and Verification, Lecture Notes in Computer Science. DSV-IS 2006, the XIII International Workshop on Design, Specification and Verification of Interactive System, Dublin, Ireland*. Pp. 137–150. Springer Berlin / Heidelberg, July 2006.

[SCS06b] J. Silva, J. C. Campos, J. Saraiva. Models for the Reverse Engineering of Java/Swing Applications. *ATEM 2006, 3rd International Workshop on Metamodels, Schemas, Grammars and Ontologies for Reverse Engineering, Genova, Italy*, October 2006.

[SCS09] J. Silva, J. C. Campos, J. Saraiva. A Generic Library for GUI Reasoning and Testing. In *In ACM Symposium on Applied Computing*. Pp. 121–128. March 2009.

[SL89] D. P. Sidhu, T.-k. Leung. Formal Methods for Protocol Testing: A Detailed Study. *IEEE Trans. Softw. Eng.* 15(4):413–426, 1989.

[SS97] R. K. Shehady, D. P. Siewiorek. A Method to Automate User Interface Testing Using Variable Finite State Machines. In *FTCS '97: Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*. P. 80. IEEE Computer Society, Washington, DC, USA, 1997.

[TG08] H. Thimbleby, J. Gow. Applying Graph Theory to Interaction Design. Pp. 501–519, 2008.

[Thi03] H. Thimbleby. The directed chinese postman problem. *In journal of Software Practice and Experience*, 2003.

[Tip95] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, september 1995.

[Ura92] H. Ural. Formal Methods For Test Sequence Generation. In *Computer Comm.* Pp. 311–325. 1992.

[YY07] Y. S. Yoon, W. C. Yoon. Development of Quantitative Metrics to Support UI Designer Decision-Making in the Design Process. In *Human-Computer Interaction. Interaction Design and Usability*. Pp. 316–324. Springer Berlin / Heidelberg, 2007.

Methodologies and Tools for OSS Software: Current State of the Practice

Zulqarnain Hashmi[1], **Siraj A. Shaikh**[2] **and Naveed Ikram**[1]

[1] zulqarnain@iiu.edu.pk, naveed.ikram@iiu.edu.pk
Department of Software Engineering,
Faculty of Basic and Applied Sciences,
International Islamic University, Islamabad, Pakistan

[2] s.shaikh@coventry.ac.uk
Department of Computing and the Digital Environment,
Faculty of Engineering and Computing,
Coventry University, Coventry, United Kingdom

Abstract: Over the years, the Open Source Software (OSS) development has matured and strengthened, building on some established methodologies and tools. An understanding of the current state of the practice, however, is still lacking. This paper presents the results of a survey of the OSS developer community with a view to gain insight of peer review, testing and release management practices, along with the current tool sets used for testing, debugging and, build and release management. Such an insight is important to appreciate the obstacles to overcome to introduce certification and more rigour into the development process. It is hoped that the results of this survey will initiate a useful discussion and allow the community to identify further process improvement opportunities for producing better quality software.

Keywords: Open Source, Testing, Debugging, Release Management, Peer Review.

1 Introduction

Open Source Software (OSS) is becoming popular both in business communities and academic sectors. The OSS movement has proved its worth with notable products such as Linux, Apache, MySQL and Mozilla, to name a few.

OSS development is typically initiated by a small group of people [SFF$^+$06] and can be distinguished from traditional development in terms of volunteers involved in the development of software dictated by their need and interest, as opposed to a dedicated team of paid developers guided by some (usually profit-making) commercial product. Such volunteers choose when and what they want to work on with typically a very loose heirarchy, as opposed to their paid counterparts. The expectation of most OSS projects is that there is less support in terms of development tools, no or very less formal design, improper project development planning, a fixed list of deliverables is not available and finally no structured testing or quality assurance of the final product. OSS projects are also less likely to be supported by project management, metrics, estimation and scheduling tools as there is no need for strict deadlines and balancing budgets [RFL05, Rob02].

OSS projects have been criticised for lack of clear and open detail of development processes. Studies on Apache and Mozilla [MFH99, MFH02, RM02] usually give an informal description of development processes which cannot be usefully replicated.

Our observation discovers that sources of information on the community, project history, work roles and task prescription provided on several OSS project websites appear mind-numbing and ambiguous. A need for standard and clear development practices has been acknowledged [Sca03]. Sharing clear and open description of development processes, with a view to further improvisation and reuse is certainly of great interest to the wider OSS community [Mic05, RM02]. Our effort is aimed at better understanding some of the development processes and behaviour within a set of OSS projects.

We present a survey of OSS developers. The survey is essentially descriptive in nature and lies in cross-sectional time dimensional category. The top 250 OSS projects from a variety of domains were selected from SourceForge [Sou09] and Launchpad [Lau09] on the basis of downloading ratings. The sampling ensured that each member of the population has an equal probability of being selected. Download rates do not convey quality or success but certainly offers a measure of fitness for purpose as users of OSS have actively downloaded it; it is essentially an objective measure independent of our influence [Mic05].

The hope is that this work will allow the wider community to identify process improvement for better quality and critical software. The importance of quality in OSS due to development practices has already been acknowledged [SC09]. Such an insight is important to appreciate the obstacles to overcome to introduce certification and more rigour into the development and testing processes. Moreover, attempts to introduce the use of formal modelling and verification within OSS developmen practices has also been suggested [CS08], though challenges have also been identified as to who and where to initiate such changes within the OSS community; addressing such challenges is of interest to us in this paper and is essentially the next step.

1.1 Rest of this paper

The rest of this paper is organised as follows. Section 2 describes some of the related work. Some past observations and the relevant trends observed are brought to attention. Section 3 discusses the methodology adopted for this effort with particular emphasis on the choice of OSS projects targeted for the survey. This is helpful in setting the results in the overall context. Section 4 presents the results of the survey. Some trends of interest are highlighted though the majority of the results serve to affirm traditional perceptions of the OSS community. Section 5 concludes the paper and promises some future work.

2 Related work

Organisational structures, technical roles and career opportunities within the OSS community have been widely studied [Sca07, YK]. Traditionally software engineers have been restricted to roles like requirements analyst, software designer, programmer or code tester. In the OSS community, roles and progression (or movement) is more sundry: volunteering roles can move up and down amongst different paths much more gracefully, with the possibility of lateral movements as well. Some of the recognised roles in OSS include *project leader*, *core developer or member*, *active developer*, *passive or peripheral developer*, *bug fixer*, *bug reporter*, *reader/active user* and *passive user*, with the likely possibility of overlap. Not all of these types of roles exist in all OSS communities, and some communities may use different names. For example, some communities refer to core members as maintainers. The difference between bug fixer and peripheral developer is also rather small as peripheral developers are likely to be engaged in fixing bugs.

Several developers and users examining the source code is one of the fundamental principles that underlies OSS. Some reports show that peer review on some OSS has been performed by millions of developers [GBBZ03]. Such code reviews are mostly done before and after any source code is committed to the repository [HS02], performed in a distributed, asynchronous manner. They are certainly more extensively acknowledged and accepted as part of the organisational culture in OSS than in traditional development. The developers more likely to perform them without any directions, which although not vital, may be a sign of commitment to quality within OSS projects. It is useful in detecting flaws, defects and quality of OSS, and is well recognised in software engineering generally for its crucial role [Ema01, HS02].

In a survey [Sta02], about 9% of OSS developers claimed the peer review of the entire source code, whereas peer review of most of the code was pointed out by 50% of the developers. Although the team members vary but the main emphasis is to maximize the ability to find bugs. The actual task is classified to be either ad-hoc or based on some checklist. The former signifies that the reviewers team has to examine in a perfect manners to dig out imperfections without any guidance. In order to guide and facilitate reviewers in examining all defect forms, standardized checklist of frequent faults is considered. There is good evidence that checklist-based techniques tend to find more defects than ad-hoc techniques [DRW03].

Testing is an essential part of the software development life cycle. Recent studies establish the uniqueness of the OSS development model with exceptionally high user involvement and structured approach for flaw/bug handling process, in the context of testing for OSS [OMK08]. Unit testing is the most frequent in OSS development. Pre-release testing on broader perspectives is less common, with the idea being that the released candidate is dealt with by the users and its flaws reported.

Pre-release testing is not commonly demonstrated and formal testing is even not implemented for most of the OSS development [HS02, GA04]. With confidence in code peer reviewed, many OSS developers are content with only minor testing [Sta02]. Some other sources go as far as to claim that over 80% of OSS developers dont have any plans of testing [ZE00].

There is no specific evidence with regards to automated tools but debuggers are widely acknowledged [ZE00]. For regression tests, about 48% of OSS projects follow baseline testing, whereas proportion is relatively higher in mega projects [ZE03]. A study conducted on the Apache project revealed that no system testing or regression was performed [MFH02]. Further analysis reports that while regression test suites were available for Apache they were not actually mandatory [Ere03]. This complements with suggestions that improvement is required in quality assurance practices, applied processes and project success criteria [OMK08].

Release management is a vital part in OSS development. Michlmayar [Mic07] presents a comparative study on release management to find that it can be categorised into three types, with respect to the concerned audience and the effort required to deliver the release: *developer release* for interested developers and experienced users requiring less or no effort, *major or stabilised* *releases* for end users requiring more effort to deliver with considerable new features and functionality, bugs fixed and tested, and *minor releases or updates* for existing users requiring a slight effort for stabilised release [MHP07].

More generally, a feature-based strategy is adopted in which certain criteria or goals have to be fulfilled, or, a time-based strategy with particular dates set for release and used as orientation for release.

3 Research Design

The research method used in this study is essentially a survey which is most common for generating primary data. This survey is descriptive in nature and lies in cross-sectional time dimensional category. The unit of analysis is essentially individuals and OSS developers are the respondents for this survey. The main objectives of the survey are to determine the development processes and developmental tools being used in OSS projects.

Our population is open source developers and targeted populations are developers of recognised OSS initiatives. The most important source to collect information about the development processes and tools used in OSS projects are OSS communities such as those accessed through *SourceForge* and *LaunchPad*, where thousands of OSS projects are hosted across several domains.

Our selected domains are *business intelligence and performance management*, *digital archiving*, *CMS systems*, *CRM*, *e-commerce*, *ERP*, *email client*, *frameworks*, *message boards*, *project management*, *scheduling*, *site management*, *social networking*, *ticketing systems* and *wiki*. We selected the top 250 OSS projects from these domains on the basis of downloading ratings from SourceForge [Sou09] and Launchpad [Lau09]. We have chosen a systematic sampling method where each member of the population has an equal probability of being selected.

Note that we use the download rate to define success. Downloads do not convey quality or success of OSS but certainly offers a measure of fitness for purpose as users of OSS have actively downloaded it. Downloads do provide the advantage as a measure as it is objective and dependent on the users [Mic05].

We designed an online questionnaire consisting of 33 questions in total. We drew inspiration from [KENU07] for questions relating to peer review and testing of software. Validity and reliability are the main priorities in surveys. There is a need of for pilot testing to assess the questionnaire clarity, understandability, comprehensiveness and acceptability. Surveys should be adequately pre-tested to check that the respondents understand the meaning of the questions or statements and to gauge whether test items are at an appropriate level of difficulty.

We validated our questionnaires by faculty members and OSS industry experts and their reliability was determined by getting few responses from the population. Participants were given an opportunity to offer comments on the structure of the questions including clarity, relevance to the objectives of the study, level of difficulty and length of the survey. Several changes were made to improve the experience as per the feedback.

A detailed search was undertaken to identify projects which existed with the same name under different domains. 250 projects were identified after eliminating duplications. Once the list of OSS projects was decided, names and contact details of the respective developers were collected (from their hosting websites). Some developers were also involved in more than one project, which were also excluded for duplication. We restricted answers in our questionnaire for a specific project.

4 Results and Analysis

This section presents the results of our survey. Section 4.1 discusses the profile of projects and individuals who responded to the survey. This sets the context for the following sections which delve into peer review practices in Section 4.2, testing strategies in Section 4.3, release management in Section 4.4 and the use of tools in Section 4.5. Section 4.6 provides a brief analysis on the results commenting on the aspects that are of particular interest.

4.1 Developer and project profile

Over 58% of the total developers surveyed have more than 5 years of experience working with OSS. Of the rest, over 18% have 3 to 5 years, over 17% have 1 to 3 years and just under 5% of the developers have less than one year of experience working with OSS. Over 3% preferred not to answer. Over 36% of the total developers who responded claimed a graduate degree, with over 25% holding a masters degree and just under 12% holding a doctoral degree. Over 31% of the total respondents identified themselves as project leader, over 25% as core developer, over 10% as active developer and over 9% as passive developer. Just over 7% claimed project management and over 3% bug reporter roles. Just under 12% fell in the *other* category, which included translator and community manager roles. Out of the total respondents, over 61% of the developers participate only part-time participation whereas over 24% are as dedicated full-time. A small percentage, just over 14%, described their participation as either in *free time*, *voluntarily* or *occassional*.

When asked about information provision and dissemination for their OSS projects, over 96% of the respondents claimed that their project has a dedicated website. Of other similar resources, over 91% identified announcements, over 87% provide some form of user documentation and just over 81% mentioned a feature list advertised for the project. Mailing lists, tutorials and to-do lists were also identified by well over 50% of the respondents. Some other avenues for communications identified included code collaborator, repositories, forums and case studies provided for the users.

For internal communication a variety of resources were identified including mailing lists (by over 76%), threaded discussion forums (60%), IRC/chat/instant messaging (over 55%), newsgroups (over 17%), community digests (just under 13%) and other resources such as XMPP, bug trackers, wiki sites and micro blogging (over 16%).

The authority to commit code varies from project to project, with the majority allowing core developers (just under 92%) to commit. Over 60% mentioned active developers and over a quarter mentioned passive developers with the ability to commit.

4.2 Peer Review

The survey reveals that software testing and release management are far more prevalent than code review, with over 87% confirming that some form of testing and release management is carried out on the OSS project they are involved in. Only over 61% of the respondents claimed any code review for the software they are involved with. This is somewhat surprising as almost 40% of the respondents did not claim any code review on their projects.

Of those who did affirm code review, over a third claimed that reviews are performed before any source code is committed to the code base. Around 30% also confirmed that some review is performed randomly and before product release.

An important element of code review is inspection of code written by others. Over a quarter of those surveyed affirmed that they regularly review other's code with just over 30% claiming occasional review of other's code. Ony under 10% said they have never reviewed source code written by others. This reflects very well highlighting a strong ethos of evaluation and self-regulation amongst the section of the OSS community.

4.3 Testing

There is strong evidence that developers have the primary responsibility for testing according to the 93% of the respondents. Testing is also left to users on over half of the projects. Dedicated individuals for quality and assurance are also identified by over 27% of the respondents. Over 42% of the projects are said to have some formal testing procedure.

The type of testing carried out is of interest here: nearly 45% of the respondents identified a black box approach to testing, with a similar 40% identifying a white box approach. For unit tests, over 35% of the developers mentioned *statement testing*, over 21% mentioned *path/branch testing*, over 17% mentioned *loop testing* and just under 6% claiming *mutation testing*. A range of testing techniques are adopted by OSS projects. When offered to identify multiple techniques, survey respondents affirmed functional testing is adopted by over 67% of the projects, with some form of system testing by over 42%, regression testing by over 42%, integration testing by just under 39% and acceptance testing by under 19%.

Over two-thirds of the projects affirmed a continuous schedule for testing, with over a third also claiming pre-release testing. Post-release testing was also highlighted by around 10% of the projects. Only under a quarter of the projects keep any form of statistical testing for future use and analysis.

4.4 Release management

Clear and consistent release procedures are important if an OSS project is to provide a coordinated and timely delivery. Our survey reveals over half of the project leaders to have complete release authority with under 20% of the projects also allowing core developers to have authority over release. Some projects also identified dedicated release or product managers having release authority.

Of the projects surveyed, just under 30% release every six months, with 11% releasing every quarter and a similar percentage every year. Nearly half of the projects release *when ready*, with just over 15% releasing on *fixed dates*, a similar number releasing *often and early* and only under 10% releasing for *fixed features*.

The decision to release is as important as the frequency: over 55% of our respondents affirmed that core team consensus is the basis for release. Almost a third also rely on single release authority's decision, with a similar number also citing market demands, committers' consensus and zero bug reporting in beta release also as contributing factors.

4.5 Tools

In this section the most commonly tools identified by the survey respondents are highlighted. This is helpful as it offers some insight into the choice of tools for OSS.

4.5.1 Version Control

Version control systems are undoubtedly crucial for OSS development as they allow management of changes to source code and documents. Our survey reveals Subversion as the most common version control system used, followed by Git and CVS. Some other choices are Mercurial, Bazaar and Darcs. Only Git and Mercurial are distributed systems as in providing no central source base and different branches holding different parts of the code.

TortoiseSVN is the most popular client for those who have affirmed the use of Subversion. RapidSVN, Textmate SVN and KDESvn are some of the other clients identified.

4.5.2 Issue Tracking

Issue tracking systems allow individual or groups of developers to keep track of outstanding bugs or issues effectively. Mantis, Bugzilla and Trac are the most popular issue tracking systems identified in our survey. Issue trackers provided by Sourceforge, Google, Codeplex and Launchpad are also identified. Other similar systems mentioned include JIRA, FogBugz, Roundup, Zentrack and YouTrack, demonstrating a very wide variety of systems in use.

4.5.3 Testing tools

A huge variety of tools supporting testing are identified in our survey including JUnit, easymock, PhpUnit, CTest, DUnit, Litmus, nosetests, Python UnitTest, QUnit, Selenium, Hudson, buildbot, NUnit, MsTests, ReSharper, TestDriven, NCover, Zope Unit testing, Ruby unit test, Squish (Froglogic), NUnit, MbUnit, GNU autotools, Pootle, scalacheck, Maven Invoker Plugin, MyTAP and GTest. There is no clear pattern for a single most popular tool, perhaps due to the nature of the activity involved.

4.5.4 Peer Review

Smart Bear Code Collaborator, Fisheye, Bugzilla, Eclipse and Pootle are some of the most popular tools identified.

4.5.5 Build System

A wide variety of build systems are identified including Ant, Make, Automake, CMake, Gnu Autotools, Tinderbox, Hudson, Bamboo, NAnt, MsBuild, Maven, SBT (Scala-based Simple Build Tool), XCode, Python setuptools, Buildout, buildbot, Module::Build, Rake (Ruby), TeamCity, PEAR (PHP Extension and Application Repository) and Eclipse.

4.5.6 Documentation System

The most common documentation system identified in the survey is Doxygen, which offers support for both on-line and off-line documentation from a set of source files. Other tools identified include Epydoc and Sphinx (for generating API documentation for Python), Javadoc (for generating API documentation in HTML format from source code), Sandcastle, DocProject, DelphiCodeToDoc, phpDocumentor (phpDoc) and RDoc.

4.5.7 Integrated Development Environment

Eclipse has been recognised as the most common development platform amongst the community surveyed. Other notable tools mentioned include CodeLite, VisualStudio, ReSharper, Quanta HTML editor, TextMate, Kate, Delphi, Lazarus, Komodo IDE, Notepad++, Qt Creator, Vim, Emacs, Xcode, NetBeans, Eclipse-Pydev and PyPaPi.

4.6 Analysis

With over half the respondents having over 5 years of experience with OSS, our survey is informed by an extensively experienced group of individuals. With nearly a two-third of the community contribution being as part-time, this reflects on the voluntary yet dedicated nature of participation by the sampled OSS community. A majority of the respondents were either project leaders or core developers with a graduate degree.

Needless to say, most projects claim to have some procedure in place for controlling changes to software and supporting document. Most of them allow core developers to commit code with nearly two-third also allowing active developers to commit. This is critical because it implies that any significant changes that need to be brought in to improve developmental processes, would not only require a consent on behalf of the core developers of the project but also depend on their adoption of new practice as well.

When it comes to testing, unit testing is most common with a strong focus on functional testing. Nearly 50% of the projects are also using some form of documented test cases. This sends a strong hint as to where more rigour and assurance measures could be incorporated in OSS development in general. Strict and specific testing for critical functionality could be the key here to associate any standard evaluation of the software and any certification that may follow.

Note that projects that have adopted some formal testing procedure are also the ones where release management is an integral part of the project.

It is interesting to observe that nearly all OSS projects use an wide range of communication tools and strategies with nearly all having a dedicated website. Feature lists, mailing lists, user and developer documentation are some of the other most common mechanisms in use. This demonstrates the need for effective and efficient communication that the disparate set of users employ to contribute to the success of OSS.

For the purposes of change of developmental practices and adoption of more rigorous means, our survey results offers to identify a starting point. It is the experienced members of the community that are best placed to bring about this change. This may appear counterintuitive as developers who are in set their ways are least likely to be agents of change. The results, however, reveal that it is indeed the most experienced (those with over 5 years of experience) of developers who perform peer review, and are responsible for testing on their projects, and have the ability to commit code and authority to release. Of those with lesser experience, a very small proportion fall in this category.

5 Conclusion

The work presented in this paper came out of a desire to understand the OSS developer community better and the state of current development practices. The accuracy of the survey results presented in this paper is undoubtedly subject to the survey design and the target population. It serves however to provide a snapshot which is both useful and indicative of further inquiry.

The motivation behind this work follows from earlier work [CS08] that encourages a more rigorous approach to software development and testing within the OSS community. Any such change therefore has to be brought about carefully. The results of this paper serve to highlight a prevailing structure of OSS projects, which should be taken advantage of. The leadership for any such initiative should also ideally come from within the community. This will facilitate adoption and better stands to influence the younger and future generations of developers who are to follow.

5.1 Future work

The target population for this survey has provided with a rich sample of the community some of whom could be targeted for further inquiry. Following the survey, we are currently in the process of setting up a shorter follow-up survey to explore the perceptions of formal methods and more rigorous methods alike for adoption by the community. Aspects of software modelling and verification, assurance and certification will be explored. We hope to report on the results of this follow-up survey soon. These results will undoubtedly provide us with a platform for more concrete proposals for change.

Acknowledgements: The authors would like to thank Shahida Bibi at International Islamic University for her assistance with data collection for this paper.

Bibliography

[CS08] A. Cerone, S. A. Shaikh. Incorporating Formal Methods in the Open Source Software Development Process. In *International Workshops 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 Software Certification (OpenCert 2008). UNU-IIST Research Report 398, pp. 26–34. 2008.

[DRW03] A. Dunsmore, M. Roper, M. Wood. The development and evaluation of three diverse techniques for object-oriented code inspection. *IEEE transactions on software engineering* 29(8):677–686, 2003.

[Ema01] K. E. Emam. Software Inspection Best Practices. *Agile Project Management Advisory Service* 2(9), 2001.

[Ere03] J. R. Erenkrantz. Release management within open source projects. *Proceedings of the Third Workshop on Open Source Software Engineering, Portland, Oregon*, 2003.

[GA04] C. Gacek, B. Arief. The Many Meanings of Open Source. *IEEE Software* 21(1):34–40, 2004.

[GBBZ03] S. Greiner, B. Boskovic, J. Brest, V. Zumer. Security issues in information systems based on open source technologies. *EUROCON*, 2003.

[HS02] T. Halloran, W. Scherlis. High quality and open source software practices. *2nd Workshop on Open Source Software Engineering, International Conference on Software Engineering*, pp. 19–25, 2002.

[KENU07] G. Koru, K. E. Emam, A. Neisa, M. Umarji. A Survey of Quality Assurance Practices in Biomedical Open Source Software Projects. *Journal of Medical Internet Research* 9(2):e8, May 2007.

[Lau09] LaunchPad Home page. https://launchpad.net/, 2009.
https://launchpad.net/

[MFH99] A. Mockus, R. Fielding, J. Herbsleb. A Case Study of Open Source Software Development: The Apache Server. *Proceedings of the 22nd International Conference on Software Engineering (ICSE), Los Angeles, CA*, pp. 263–272, 1999.

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

[MHP07] M. Michlmayr, F. Hunt, D. Probert. Release management in free software projects: Practices and problems. *IFIP International Federation for Information Processing, Open Source Development, Adoption and Innovation* 234:295–300, 2007.

[Mic05] M. Michlmayr. Software Process Maturity and the Success of Free Software Projects. *Software Engineering: Evolution and Emerging Technologies* 130:3–14, 2005.

[Mic07] M. Michlmayr. *Quality Improvement in Volunteer Free and Open Source Software Projects – Exploring the Impact of Release Managemen*. PhD thesis, University of Cambridge, UK, 2007.

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

[RFL05] J. Robbins, H. Fitzgerald, S. Lakhani. Adopting Open Source Software Engineering (OSSE) Practices by Adopting OSSE Tools. *Perspectives on Free and Open Source Software*, pp. 245–264, 2005.

[RM02] C. R. Reis, R. P. de Mattos Fortes. An overview of the software engineering process and tools in the mozilla project. *Workshop on OSS Development, Newcastle upon Tyne, UK*, pp. 162–182, 2002.

[Rob02] J. E. Robbins. Adopting OSS methods by adopting OSS tools. *2nd Workshop on Open Source Software Engineering (co-located with 24th International Conference on Software Engineering) Orlando, Florida*, 2002.

[SC09] S. A. Shaikh, A. Cerone. Towards a metric for Open Source Software Quality. In Barbosa et al. (eds.), *Foundations and Techniques for Open Source Certification 2009*. Electronic Communication of the European Association of Software Science and Technology (ECEASST) 20. 2009.

[Sca03] W. Scacchi. Issues and Experiences in Modeling Open Source Software Development Processes. In *In Proceedings of the 3rd ICSE workshop on Open Source Software Engineering*. Pp. 121–125. 2003.

[Sca07] W. Scacchi. Free/Open Source Software Development : Recent Research Results and Emerging Oppurtunities. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007.

[SFF$^+$06] W. Scacchi, J. Feller, B. Fitzgerald, S. Hissam, K. Lakhani. Understanding Free/Open Source Software Development Processes. *Software Process: Improvement and Practice* 11(2):95–105, 2006.

[Sou09] SourceForge Home page. http://sourceforge.net/, 2009.
http://sourceforge.net/

[Sta02] J. Stark. Peer reviews as a quality management technique in open-source software development projects. *European Conference on Software Quality*, pp. 340–350, 2002.

[YK] Y. Ye, K. Kishida. Toward an understanding of the motivation Open Source Software developers. *Proceedings of the 25th International Conference on Software Engineering*, pp. 364–374.

[ZE00] L. Zhao, S. Elbaum. A survey on quality related activities in open source. *ACM SIGSOFT Software Engineering Notes*, pp. 53–57, 2000.

[ZE03] L. Zhao, S. Elbaum. Quality assurance under the open source development model. *The Journal of Systems and Software* 66:65–75, 2003.

Integrating Data from Multiple Repositories to Analyze Patterns of Contribution in FOSS Projects

Sulayman K. Sowe[1*] **and Antonio Cerone**[2]

[1] sowe@merit.unu.edu,
UNU-MERIT. 6211 TC, Maastricht, The Netherlands.

[2] antonio@iist.unu.edu
UNU-IIST, Macao, China.

Abstract: The majority of Free and Open Source Software (FOSS) developers are mobile and often use different identities in the projects or communities they participate in. These characteristics not only poses challenges for researchers studying the presence (where) and contributions (how much) of developers across multiple repositories, but may also require special attention when formulating appropriate metrics or indicators for the certification of both the FOSS product and process. In this paper, we present a methodology to study the patterns of contribution of 502 developers in both SVN and mailing lists in 20 GNOME projects. Our findings shows that only a small percentage of developers are contributing to both repositories and this cohort are making more commits than they are posting messages to mailing lists. The implications of these findings for our understanding of the patterns of contribution in FOSS projects and on the quality of the final product are discussed.

Keywords: Open Source Software developers, Open Source Software projects, Software repositories, Concurrent Versions System, Mailing lists, Linking data, Software Quality.

1 Introduction

Free and Open Source Software (FOSS) developers are like nomads; freely moving from one project to another. They commit bits and pieces of code, report and fix bugs, take part in discussions in various mailing lists, forums, and IRC channels, document coding ethics and guidelines, and help new entrants. Along the way they create and archive a wealth of knowledge and experience associated with their art [SAS06]. Participants in various projects use tools (Versioning Systems, mailing lists, bug tracking systems, etc.) to enable the distributed and collaborative software development process to proceed. These tools serve as repositories which can be data mined to understand *who* is involved, who is talking to *whom*, *what* is talked about, *how much* someone contributes in terms of code commits or email postings. Thus, by applying cyber-archeology [SII07] to these repositories, we can learn and better understand the patterns of contribution [SFF$^+$06, GKS08] of FOSS developers in the projects concerned.

[*] *Correspondence Author*: Sulayman K. Sowe. Email:{sowe@merit.unu.edu}. *Address*: Keizer Karelplein 19. 6211 TC Maastricht, The Netherlands. Tel: +31(0)43 388 4432 Fax: +31(0)43 388 4499

An important aspect of software engineering research, and the certification of FOSS products in particular, is understanding and measuring the contribution of individuals, particularly developers, who work on a project [SO09a, SAS06]. A host of factors which have both empirical and industrial implications motivates this kind of research. Factors include, but not limited to, (i) helping practitioners understand and monitor the rate of project development, (ii) characterizing FOSS projects in terms of developers turnover and extent of contribution [CLM03, GKS08, SSSA08] (iii) identifying bottlenecks and isolate exceptional cases in terms of projects and individuals contributions [SO09a], (iv) using the research results to develop new metrics or evaluate an existing taxonomy [SO09b] of metrics (Process, Product, and Resources) for FOSS quality attributes [SAOB02] and the certification process. Furthermore, as argued by [SC09], communication and patterns of contribution are factors that contribute to measure the efficiency of the development process, a measure that the authors called “quality by development”. Indeed, the patterns of [code] contribution in FOSS projects has emerged as an important measure in assessing the quality of FOSS products [SO09b, SAOB02].

A lot of research utilizes data from a single repository to analyze code contribution of developers [RG06, GKS08], trends and inequality in posting and replying activities in Apache and Mozilla [MFH02], KDE [Kuk06], Debian [SSL08], and FreeBSD [DB05]. Most of these researches use data from CVS or mailing lists as these are *de facto* repositories in FOSS projects. Source configuration management (SCM), of which CVS or SVN[1] is part, is mainly used to coordinate the coding activities of software developers and manage software builds and releases. Mailing lists, on the other hand, are the main communication channels [SSL08]. Many important aspects of a project are negotiated in [developer] lists: software configuration details, the way forward and how to deal with future requests, how tasks are distributed, issues concerning package dependencies, scheduling online and off-line meetings, etc. Thus, for a developer to keep abreast with developments in a project, committing code to SVN alone is not sufficient. S/he needs to participate in the respective lists, communicate his ideas, and engage with colleagues. To bolster this view, [Bro75] pointed out the essence of communication as a means to foster long term success of software projects. This may take the form of a bi-directional developer to developer, developer to user, and developer to community communication.

Even though a strong linkage exist between the information in FOSS repositories (e.g. bug reports and source code repositories [DB07, ZPZ07]), few researchers strive to understand how developers’ contributions varies across repositories. In this research we tired to fill this niche by establishing links between SVN and mailing lists to locate developers who are present in both repositories and quantify their contribution in terms of commits and posts.

The rest of the paper is organized as follows. First, in section 2, we discuss the rationale behind this research and construct two hypothesis which will guide us for the rest of the paper. In section 3, we outline the methodology and data used in this research and present our algorithm for identifying and quantifying developers contributions to both SVN and mailing lists. This is followed by an analysis and discussion of our results in section 4. Our concluding remarks and future work are presented in 5 section.

[1] Note: SVN is our software code repository (see Subsection 3.1). Reference is made to CVS when other researchers mentioned using data from that repository.

2 Research Rationale and Hypothesis

For software projects to evolve, it is by design that developers must continuously commit and review the codebase. In the eyes of the developer, user, and business community an active mailing list is a proxy of project success. The presence of project's leads, core and active developers in mailing lists has a profound effect on the way individuals within and outside the project see the commitment of the most influential members in the project. For software companies and private enterprises, developers presence in lists may indicate that software support activities are not only available from ordinary users, but also comes form individuals behind the software and project. Thus, developers should strive to balance their coding activity with their involvement in mailing lists. This raises a number of questions which may be of great interest to both FOSS project administrators and researchers. For instance: How many developers are willing to commit code and patches and at the same time participate in discussions in mailing lists and other project's fora? If developers are coding more than they are participating in mailing lists, what does this tell us about the maintenance and dynamics of the software and project? How much effort can a developer allocate to one activity and at what stage in the project's life-cycle? If attaining a balance activity is much required in a project, how can project administrators schedule and assign or dedicate one activity at the expense of another? What is the impact on the performance the project of having developers specializing in on activity?

In this research, we used data provided by the FLOSSMetrics project (http://flossmetrics.org/) to proposed a methodology to help us answer some of the above questions. FOSS researchers (e.g. [MFH02, Kuk06, DB05]) study and report developers coding activities separately from their mailing lists activities. However, research on the FOSS development process [Mas05, SFF$^+$06] informs us that in many projects, a small number of talented core developers or "cod gods" [RG06] are busily (as if in a *software beehive*) submitting patches and tinkering with code to produce good and usable software for the rest of the community. This cohort also contribute to discussions in mailing lists; interacting with other software developers and users, keeping abreast with project activities and monitoring what goes on in there projects or packages [SSL08]. Nevertheless, we conjecture that not all the developers who commit or make changes to a project's source repository also participate in [developer] mailing lists. This study investigates the contributions of FOSS developers to both SVN and developer mailing lists and presents a methodology to overcome the empirical research challenges associated with integrating or linking data from multiple repositories. That is, we find out if developers are coding through commits in SVN as much as they are participating in mailing lists. This involves correlating developers commits activities with their corresponding mailing lists activities within the same project.

Research Hypothesis. Hypothesis put forward in this research are the following;

- **Hypothesis** [H1]: Since developers must code and commit, *ad infinitum*, for the software and project to evolve, we hypothesize that *FOSS developers make more commits to a project's code (SVN) repository than they are posting messages to mailing lists.*
- **Hypothesis** [H2]: However, we posit that developers must strike a balance between their coding and mailing lists activities. Thus, *FOSS developers contribute equally to code repository and mailing lists.*

3 Research Methodology

The methodology employed in this research investigates the simultaneous occurrence of developers in SVN and mailing lists. That is, identifying developers who make both commits and postings, and ensuring that the developer making the SVN commit is the same individual posting to the developer mailing list(s) of the same project. The methodology also ensures that developers with multiple identities are only counted once. The methodology as represented in figure 1, shows the FLOSSMetrics database as the data source from which we extracted SVN and mailing lists data for the 20 projects in our study. In our data acquisition, a fundamental question is always asked; 'is this the same developer we have in both repositories?' Figure 1 also shows the MYSQL tables and fields from which we extracted commits and posts which are used in our analysis to identify developers (see subsection 3.3) in the projects. The links between the tables as indicated by the arrows (with "IS") shows the path taken to locate a developer and counting his contribution to both SVN and mailing lists.

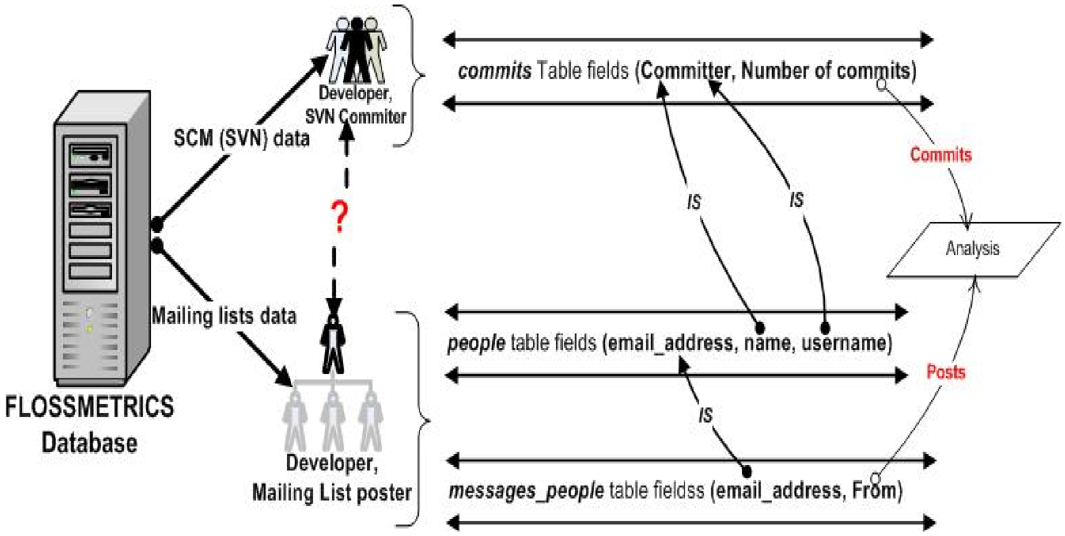

FLOSSMETRICS
Database
SCM (SVN) data
Mailing lists data
Developer,
SVN Commiter
?
Developer,
Mailing List poster
commits Table fields (Committer, Number of commits)
people table fields (email_address, name, username)
messages_people table fieldss (email_address, From)
IS
IS
IS
Commits
Posts
Analysis

Figure 1: Methodology to Identify developers from multiple repositories.

3.1 Data

The data for this research consists of the 20 GNOME projects shown in table 1. The FLOSSMetrics database retrieval system uses a combination of tools (http://tools.libresoft.es/) to retrieve data from projects (e.g. GNOME and Apache) and forges (e.g. SourceForge) and computes various code and community metrics. The *CVSAnalY2* [ARG06, RGCH09, SSSA08] tool retrieves Source Content Management systems (SCM) data and stores committers attributes into various tables. The *MLStats* [SSSA08] tool extracts one or more mailing lists archives of a particular project. For each of the 20 projects, committers SVN identifications (commit ID) and the total number of commits each committer made is extracted. For the mailing list data, for each project, data was extracted from two FLOSSMetrics database tables: Two fields (type_of_recipient and

Table 1: List of GNOME projects studied

No.	Projects	No.	Projects
1	Balsa	11	GNOME Control Center
2	Brasero	12	GNOME Games
3	Deskbar Applet	13	GNOME Media
4	Ekiga	14	GNOME Power Manager
5	Eog	15	GNOME Screensaver
6	Epiphany	16	GNOME System Tools
7	Evince	17	Libsoup
8	Evolution	18	Metacity
9	GDM	19	Nautilus
10	gedit	20	Seahorse

email_address) from the *"messages_people"* table. The type_of_recipient field has the format "**From**", "To", and "Cc". The "From" email header is used to identify lists posters [QJ04] and counting their contribution to mailing lists [SAS06]. And three fields (**email_address**, **name**, and **username**) from the *"people"* table.

3.2 Data Cleaning

Having identified fields needed to analyze developers participation in SVN and mailing lists, we proceeded with data cleaning. For the mailing lists data, since we need both the "name" and "username", all posters without recognizable names and/or usernames were removed. Some of the names contained unrecognizable characters such as "*=?ISO88591?Q?g=FCrkan_g=FCr?=*". Some of the posts with null posters/developere were also removed. Furthermore, since the full name (first +last) is needed to identify a developer, all posters with a single name were deleted from the mailing lists data. That is, delete developer "*Foo*" but retain developer "*Foo Bar*". For the SVN data, all commits without committers or authors were removed. Aggregate number of items deleted in each of the above categories were; Unrecognizable characters = 28, Posts with null posters = 30, Posters with a single name = 14, and Commits without authors = 5093.

3.3 Identification of developers across repositories

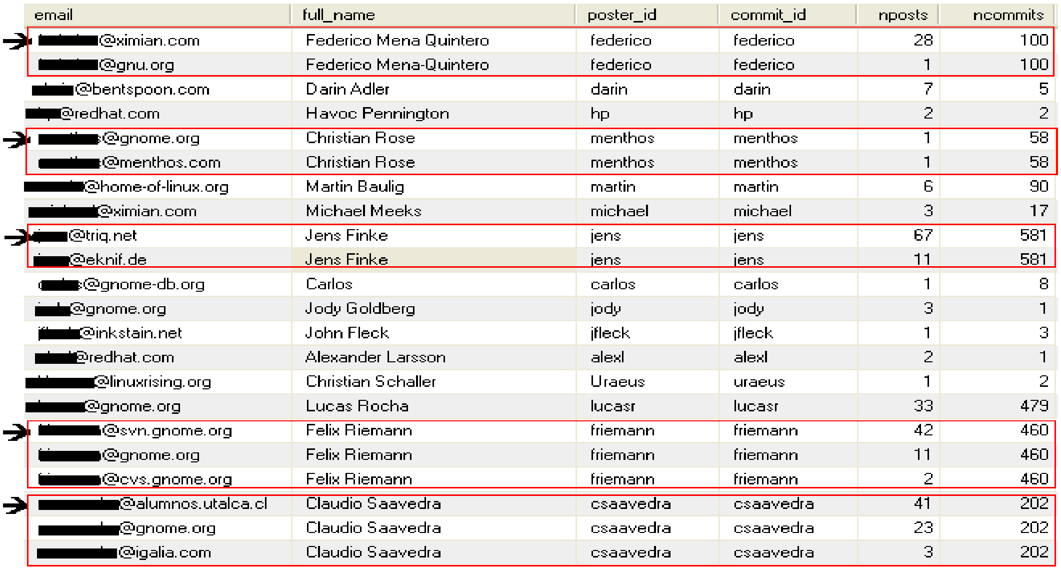
As depicted in figure 1, a poster in the mailing lists can be identified in two ways. In the *messages_people* table, a poster is identified by his email address. By using the "*From*" field, all the emails posted by a particular person can be aggregated . The *people* table is used to identify a poster through his *"email address"*, poster *"name"* in the form of first name + last name (eg. Pawel Salek), and *"username"* (eg. pawsa). For the SVN data, the committer field from the *commits* table was used to identify a committer or author of a commit. In SVN, an individual is simply identified as a "Committer" or an "Author" of one or more commits. Mailing lists participants, on the other hand, can be identified by means of message identifiers like "From:" in email headers [SAS06]. The identification process proceeds thus;

1. For each project in the *commits* table, LIST all the committers and for each committer (unique commit ID or *commit_id*) SUM all his commits and store the value as *ncommits* variable.

2. For each project in the *people* table, LIST ("email address" + "name" + "username" or *poster_id*) WHERE both name and username is the same for this committer as in the *commits* table. And

   - From the *messages_people* table, LIST developers "email address", WHERE *people*.email address = *messages_people*.email address. For each developer, COUNT all the posts and store the value as *nposts* variable.

The results of a typical query is shown in figure 2, with developers emails anonymized. From the query, it can be seen that a developer may appear many times. This is because, while a developer has only one identification in SVN, his commit id, the same developer may use many email addresses when posting messages to developer mailing lists.



email	full_name	poster_id	commit_id	nposts	ncommits
[redacted]@ximian.com	Federico Mena Quintero	federico	federico	28	100
[redacted]@gnu.org	Federico Mena-Quintero	federico	federico	1	100
[redacted]@bentspoon.com	Darin Adler	darin	darin	7	5
[redacted]@redhat.com	Havoc Pennington	hp	hp	2	2
[redacted]@gnome.org	Christian Rose	menthos	menthos	1	58
[redacted]@menthos.com	Christian Rose	menthos	menthos	1	58
[redacted]@home-of-linux.org	Martin Baulig	martin	martin	6	90
[redacted]@ximian.com	Michael Meeks	michael	michael	3	17
[redacted]@triq.net	Jens Finke	jens	jens	67	581
[redacted]@eknif.de	Jens Finke	jens	jens	11	581
[redacted]@gnome-db.org	Carlos	carlos	carlos	1	8
[redacted]@gnome.org	Jody Goldberg	jody	jody	3	1
[redacted]@inkstain.net	John Fleck	jfleck	jfleck	1	3
[redacted]@redhat.com	Alexander Larsson	alexl	alexl	2	1
[redacted]@linuxrising.org	Christian Schaller	Uraeus	uraeus	1	2
[redacted]@gnome.org	Lucas Rocha	lucasr	lucasr	33	479
[redacted]@svn.gnome.org	Felix Riemann	friemann	friemann	42	460
[redacted]@gnome.org	Felix Riemann	friemann	friemann	11	460
[redacted]@cvs.gnome.org	Felix Riemann	friemann	friemann	2	460
[redacted]@alumnos.utalca.cl	Claudio Saavedra	csaavedra	csaavedra	41	202
[redacted]@gnome.org	Claudio Saavedra	csaavedra	csaavedra	23	202
[redacted]@igalia.com	Claudio Saavedra	csaavedra	csaavedra	3	202

Figure 2: Query showing the identification of FOSS developers from SVN and Mailing lists.

3.3.1 Unmasking Aliases and removing duplicates

The volunteering nature of the FOSS development process and participation in public repositories means that participants may use different emails. For example, as shown in figure 2, a developer (e.g. friemann) has his identity masked using three email aliases; foo@svn.gnome.org, bar@gnome.org, foo.bar@cvs.gnome.org. The fundamental problem in email alias unmasking [BGD+06, SSL08] is finding out that those aliases all belong to one developer. The algorithm for checking duplicate records and unmasking aliases in the mailing lists data proceeded thus;

```
For ALL records in project X
IF ncommits  > 1 for THIS developer
AND poster_id = commit_id
RETURN ''THIS is a duplicate''
RECORD ONLY 1 value of ncommits for THIS developer
THEN SUM nposts for this developer
```

The query scenario in figure 3 shows the result when the algorithm is applied to the dataset. This literally means; a developer (e.g. federico in figure 2) with a unique commit id made 100 commits to the project's SVN. However, he contributed to mailing lists using two emails (foo@ximian.com and foo.bar@gnu.org). He posted 28 messages using the first email and 1 message using the second email. The developer's overall email postings is the sum of the two posts he made using the different emails, i.e. 28 + 1 = 29. All duplicate records are identified and developers *nposts* and *ncommits* are calculated in a similar manner. There were 115 duplicate records of this nature per project in our dataset. This means that many developers are using multiple email addresses. Generally, as shown on the right hand side of figure 3, a developer contribution to mailing list (nposts) will be counted as X + Y + Z, whilst his SVN contribution (ncommits) will be counted as α.

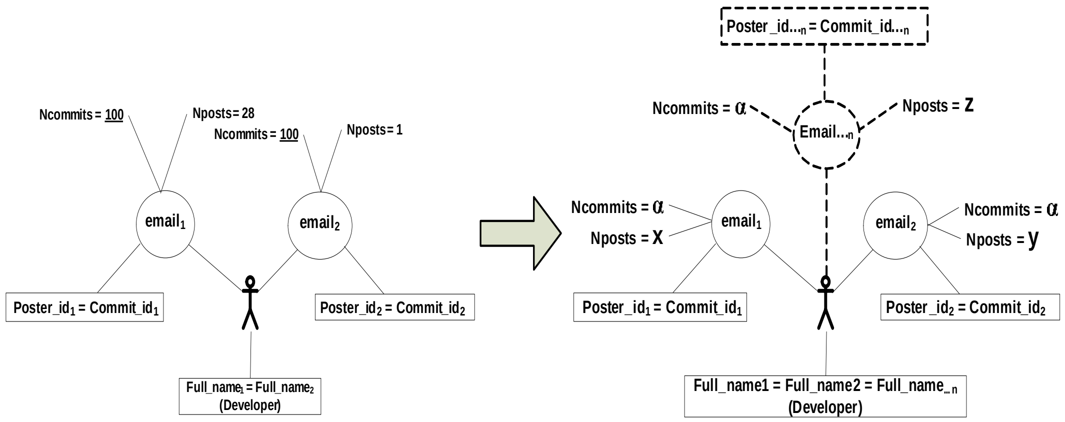

Ncommits = 100
Nposts= 28
Ncommits = 100
Nposts= 1
email$_1$
email$_2$
Poster_id$_1$ = Commit_id$_1$
Poster_id$_2$ = Commit_id$_2$
Full_name$_1$ = Full_name$_2$ (Developer)
Poster_id...$_n$ = Commit_id...$_n$
Ncommits = α
Email...$_n$
Nposts = Z
Ncommits = α
Nposts = X
email$_1$
email$_2$
Ncommits = α
Nposts = y
Poster_id$_1$ = Commit_id$_1$
Poster_id$_2$ = Commit_id$_2$
Full_name1 = Full_name2 = Full_name$_{...n}$ (Developer)

Figure 3: Query scenario to identify developers in SVN and mailing lists

4 Analysis and discussion

According to [Sek06], an exploratory study is undertaken "when not much is known about the situation at hand or no information is available on how similar problem or research issues have been solved in the past". Thus, we begin our analysis using what we call an *exploratory data analysis (*EDA*) technique* to help us examine the distribution, the nature of the commits and posts, and prepare the ground for what may be the appropriate analysis technique to be used to answer the research hypothesis. Tables 2 and 3 shows the descriptive statistics of the developers posting and committing activities after data cleaning.

Table 2: Descriptive statistics of Posts

Projects	N Posters	Mean	Median	Std. Dev.	Skewness	Std. Err. of Skewness	Max.	Sum
Balsa **	1088	12.98	2.00	67.273	13.942	.074	1465	14125
Brasero	63	4.13	1.00	8.071	3.498	.302	45	260
Deskbar Applet	97	7.00	2.00	19.187	5.200	.245	137	679
Ekiga **	729	9.24	3.00	59.999	22.103	.091	1509	6734
Eog	134	4.17	1.50	8.914	4.533	.209	67	559
Epiphany **	889	5.91	1.00	23.795	12.657	.082	470	5250
Evince **	451	3.46	1.00	13.093	14.013	.115	238	1562
Evolution **	4769	7.44	2.00	46.274	25.619	.035	1760	35478
GDM **	658	3.99	1.00	25.597	20.006	.095	595	2628
gedit **	571	3.95	1.00	15.653	14.252	.102	306	2253
GNOME Power Manager **	203	5.58	2.00	33.046	13.881	.171	470	1133
GNOME Control Center	174	8.36	2.00	20.936	5.261	.184	186	1455
GNOME Games	173	8.79	2.00	25.146	5.909	.185	224	1521
GNOME Media	289	5.39	2.00	12.270	5.884	.143	115	1557
GNOME Screensaver	27	5.59	3.00	7.846	3.322	.448	39	151
GNOME System Tools **	297	4.51	1.00	11.019	6.076	.141	112	1339
Libsoup **	52	3.73	1.00	8.761	6.326	.330	63	194
Metacity	60	4.82	2.00	11.029	5.301	.309	77	289
Nautilus **	2065	8.61	2.00	61.402	32.822	.054	2475	17782
Seahorse	62	6.16	2.00	18.382	5.390	.304	122	382

Table 3: Descriptive statistics of Commits

Projects	N Committers	Mean	Median	Std. Dev.	Skewness	Std. Err. of Skewness	Max.	Sum
Balsa	181	44.09	4.00	241.309	9.233	.181	2688	7981
Brasero ++	86	26.05	5.00	137.976	8.869	.260	1271	2240
Deskbar Applet ++	133	19.67	5.00	84.751	8.413	.210	834	2616
Ekiga	186	41.99	5.00	286.865	12.130	.178	3757	7810
Eog	298	16.59	4.00	53.660	8.231	.141	581	4944
Epiphany	252	34.84	6.00	217.618	14.340	.153	3352	8780
Evince	203	17.30	4.00	59.881	7.494	.171	535	3511
Evolution	430	81.11	10.00	309.253	8.099	.118	4061	34877
gdm	282	23.63	5.00	103.297	9.653	.145	1266	6663
gedit	329	20.68	5.00	81.699	10.704	.134	1153	6804
GNOME Power Manager	148	22.75	5.00	161.952	12.060	.199	1974	3367
GNOME Control Center ++	423	21.39	6.00	55.908	6.917	.119	634	9049
GNOME Games ++	321	27.68	7.00	89.618	8.559	.136	1164	8884
GNOME Media ++	324	13.05	4.00	31.803	6.804	.135	345	4228
GNOME Screensaver ++	126	12.79	4.00	74.388	11.097	.216	838	1611
GNOME System Tools ++	207	20.55	5.00	82.615	10.079	.169	1043	4254
Libsoup	37	32.49	1.00	111.261	5.067	.388	647	1202
Metacity ++	264	15.50	4.00	61.675	8.547	.150	600	4091
Nautilus	395	37.52	7.00	126.202	8.529	.123	1712	14822
Seahorse ++	137	21.39	5.00	99.481	9.603	.207	1087	2931

As shown in table 2, for each project the total number of posters (N posters), the mean post per poster, the median, standard deviation, skewness, the maximum posts made by one individual, and the total or sum of postings for that project are shown. For all the projects, the mode and minimum numbers of posts made equals 1. A total of 12,851 posters contributed 95,331 email messages. Table 3 shows the same descriptive statistics for the committers (N Committers) in each project. A total of 4,762 developers made 140,665 commits. Evident from the statistics is that each project has its unique characteristics [CLM03] in terms of developers' postings and committing activities, as well as the number of developers involved in each activity. For instance, 45% (N = 9) of the projects (marked with ++ in table 3) have more committers than posters. The other 55% (N = 11) of the projects (marked with ** in table 2) have more posters than committers.

Furthermore, figures 4 and 5 (both Y-axis in logarithmic scale) shows the distribution of posts and commits in the respective projects. From the boxplots it can be seen that the contributions of the developers to mailing lists is characterised by smaller means (post per poster). However, the posting data has many outliers; with many developers posting few emails and a few making large numbers of posts. On the contrary, the commits are characterised by larger means (commits per committer). These characteristics are reminiscent of power distributions observed in FOSS participants' contributions to mailing lists [SSL08] and CVS [MFT02] activities.

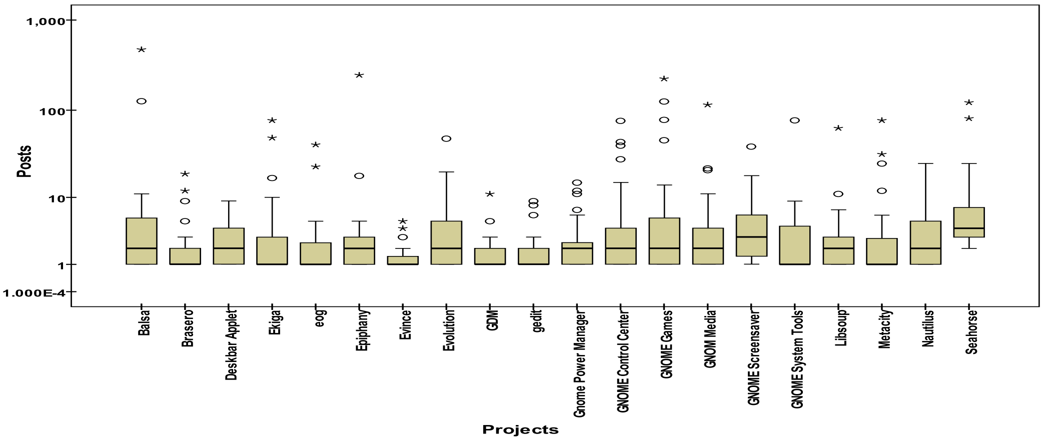


Figure 4: Box-plots showing the distributions of Posts.

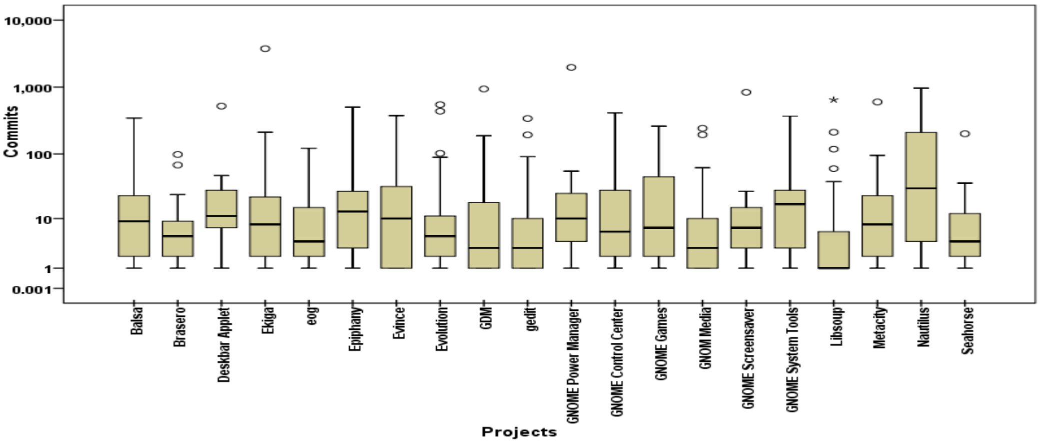


Figure 5: Box-plots showing the distributions of Commits.

4.1 Developers in both SVN and Mailing lists

In order to analyze the simultaneous occurrence of the developers in both repositories, we queried the SVN and mailing lists data for each project and computed developers contributions in terms of the *ncommits* and *nposts* variables discussed in subsection 3.3. Table 4 shows the number of developers (N_dev) in each project who contributed to both SVN and mailing lists. For the 20 projects, 502 developers made more commits (mean = 152.1; Std. deviation = 431.171) than posts (mean = 43.19; Std. deviation = 164.353). Furthermore, as shown in figure 6, our identification technique and algorithm revealed a relatively small, but varying, percentage of developers who are involved in both activities[2]. The percentage of developers in each activity varies across the projects. For example, the Ekiga, Gnome Power Manager, and Seahorse projects having less than 5% of their developers committing to SVN and at the same time posting messages to their respective projects' mailing lists. Projects such as Balsa and Nautilus have few poster (3.68% and 3.23%), but higher percentage of committers (22.1% and 34.43%).

Table 4: Developers contribution to both SVN and Mailing Lists

		nposts					ncommits				
Projects	*N_dev.*	Mean	Median	Std.dev.	Max.	Sum	Mean	Median	Std.dev.	Max.	Sum
balsa	40	37.23	5.5	133.76	851	1489	112.53	25	206.33	751	4501
brasero	6	19.33	2	30.936	77	116	69.17	8.5	98.3	196	415
deskbar_applet	8	20.13	6	35.64	106	161	120.25	5.5	289.5	834	962
ekiga	4	438.25	121.50	722.49	1509	1753	2170.25	2417	1876.01	3757	8681
eog	16	18.81	4.5	25.95	78	301	129.38	37.5	196.16	581	2070
epiphany	40	55.73	7	116.69	470	2229	146.82	16.5	536.03	3352	5873
evince	18	27.17	2	58.61	238	489	100.89	10.5	180.31	535	1816
evolution	92	56.47	4.5	172.68	1481	5195	283.29	46	622.01	4061	26063
gdm	21	26.38	2	56.63	227	554	112.9	17	242.76	939	2371
gedit	19	20.84	2	69.34	306	396	103	4	267.13	1153	1957
gnome_control_center	35	21	4.00	51.753	296	735	69.54	19	125.13	527	2434
gnome_games	14	43.57	7	83.15	304	610	178.93	13.5	341.49	1164	2505
gnome_media	23	21.87	5	36.67	130	503	39.22	6	84.03	345	902
gnome_power_manager	7	3.43	4	1.51	6	24	8	2	11.4	32	56
gnome_screensaver	4	15.75	10	15.84	39	63	211.5	3.5	417.67	838	846
gnome_system_tools	22	17.14	3	33.42	154	377	92.59	24	228.27	1043	2037
ibsoup	3	28.33	8	39.63	74	85	219.67	8	370.09	647	659
metacity	10	14.8	6	23.85	77	148	184.2	7	270.12	600	1842
nautilus	136	49.95	8.5	225.14	2475	6793	86.63	13	220.1	1712	11782
seahorse	2	73.5	73.5	101.12	145	147	198	198	275.77	393	396

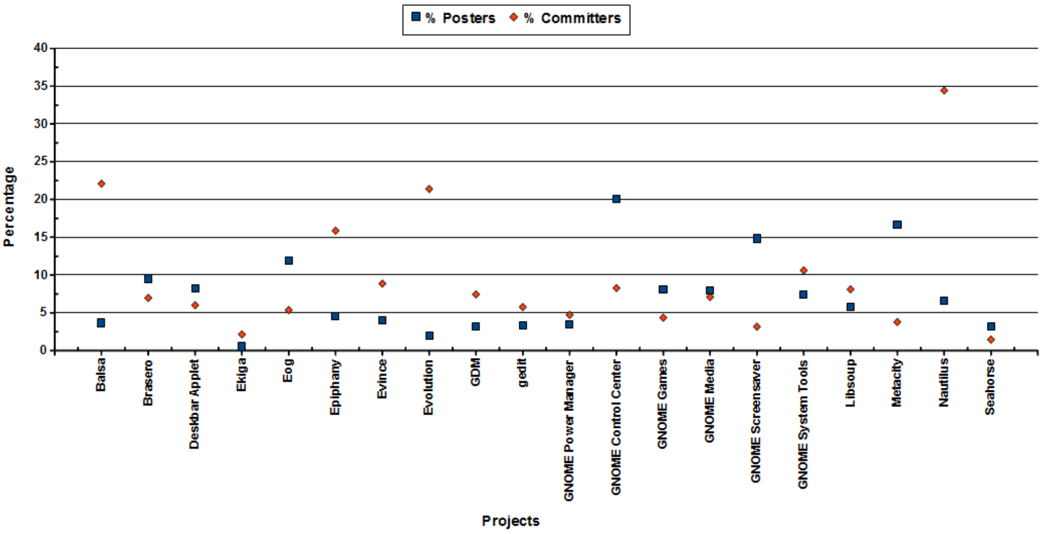


Figure 6: Percentage of developers involved in posting and committing

[2] Calculated as: *% committers = (N_dev/N committers)*100* ; *% posters = (N_dev/N Poster)*100*

4.1.1 Are developers making more commits than posts?

Hypothesis [H1]: *FOSS developers make more commits to a project's code (SVN) repository than they are posting messages to mailing lists.*
In our investigation of **H1**, for each project, we compared the total number of commits made to SVN with the total number of messages developers posted to the mailing lists. The pattern of contribution for all the 502 developers in the 20 projects is shown in the boxplots in figure 7. In the boxplots, the median line and error T-bar widths for each set of project data (nposts and ncommits) are shown. The domination of SVN commits, with larger means of commits per developer (mean = 150.32, Std. deviation = 424.986) over posts (mean = 42.63, Std. deviation = 161.852) is evident in all the projects.

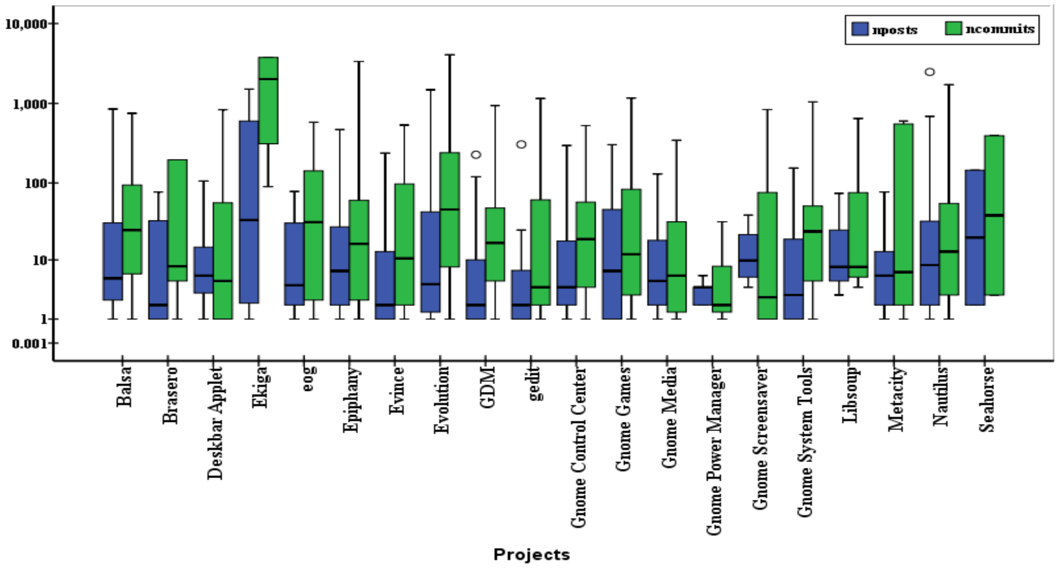


Figure 7: Distributions of commits and posts for all projects (y-axis in log scale).

4.1.2 Are developers contributing equally to SVN and mailing lists?

Hypothesis [H2]: *FOSS developers contribute equally to code repository and mailing lists.*
We used correlation between commits and posts to study how developers activities in SVN and mailing lists are related. The scatter plot in figure 8 shows the correlation between commits and posts in all projects. In the plot, data points are fitted to a line to show the trend in the commits and posting activities of the developers. Previous research ([SSL08]; page 414) showed that FOSS developers and users mailing lists activities have *fractal* or self-similarities properties and could best be explained by a polynomial model of third order, i.e. a cubic relation of the type

$$LogN = b0 + b1 * logr + b2 * (logr)^2 + b3 * (logr)^3 \tag{1}$$

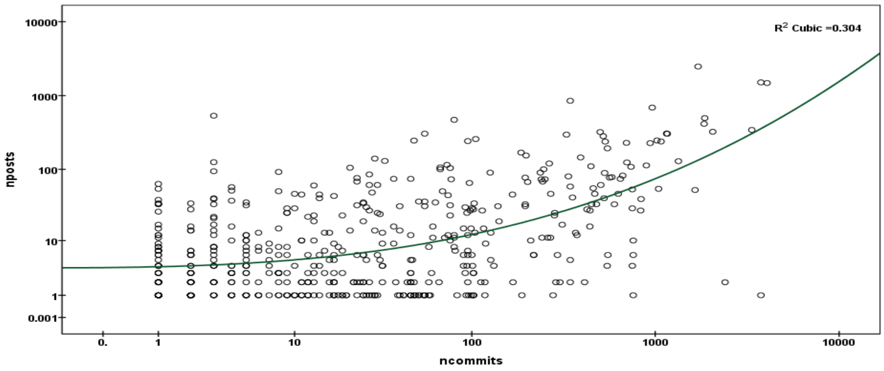


Figure 8: Relationship between commits and posts. *both axis in a log scale*.

As shown in figure 8, our fit method could explain 30.4% (r^3 = 0.304) of the variability in commits and posting activities. This translates to 26.5% or r^2 = 0.265 in linear terms. The linear association between *nposts* and *ncommits* as measured by Pearson correlation = 0.594, and this is significant at the 0.01 level (2-tailed) with ρ = 0.000. However, the *nposts* and *ncommits* data are not normally distributed and have outliers. Thus, nonparametric correlations using Spearman's *rho* and Kendall's *tau_b* statistics, which work regardless of the distribution of the variables [Nor04], are used to report the association between posts and commits. Table 5 shows that, overall, there is a low correlation between commits and posts, with Spearman's coefficient (ρ) = 0.426 (p =1.000). Furthermore, Wilcoxon signed ranks for the Two-Related-Samples Tests procedure was used to compare the distributions of two variables. The results of the test in table 6 shows that for the 502 developers in the 20 projects; for 140 = ncommits $<$ nposts, for 327 developer ncommits $>$ nposts, and 35 developers had a balanced activity with ncommits = nposts.

Table 5: Correlations between posts and commits

			nposts	ncommits
Kendall's tau_b	nposts	Correlation Coefficient	1.000	.308
		Sig. (2-tailed)	.	.000
		N	502	502
	ncommits	Correlation Coefficient	**.308**	1.000
		Sig. (2-tailed)	.000	.
		N	502	502
Spearman's rho	nposts	Correlation Coefficient	1.000	**.426**
		Sig. (2-tailed)	.	.000
		N	502	502
	ncommits	Correlation Coefficient	.426	1.000
		Sig. (2-tailed)	.000	.
		N	502	502

Table 6: Ranks of developers contribution

Variable		$N_{dev.}$	Mean Rank	Sum of Ranks
ncommits -nposts	Negative Ranks	140	175.46	24565.00
	Positive Ranks	327	259.06	84713.00
	Ties	35		
	Total	502		

5 Concluding remarks

In this paper we have put forward research questions to investigate whether FOSS developers are making more commits to code repositories (SVN) than they are posting messages to mailing lists, and whether developers should aim at a balanced activity by contributing equally there repositories. Despite the fact that FOSS data is widely available and can be easily extracted [SASM07], this kind of research is made difficult because of the problem associated with integrating data from and the subsequent identification of developers from multiple repositories (SVN and mailing lists). We have presented and discuss a methodology which alleviates this empirical research obstacle. The methodology and algorithm enabled us to locate and count the quantitative contribution of FOSS developers in 20 GNOME projects.

An exploratory data analysis or EDA technique was used to show that each project has its unique characteristics and developers contribution to either coding or mailing lists can vary tremendously. In our data consisting of 12,851 posters and 4,762 committers who, respectively posted 95,331 email messages and made 140,665 commits, we found out that in 55% (N = 11) of the projects there are more developers as posters, with smaller means (post per poster), than committers, with larger means (commits per committer). From this sample of posters and committers we are able to extract 502 developers who simultaneously contribute to both SVN and mailing lists. This cohort made more commits (mean = 152.1; Std. deviation = 431.171) than posts (mean = 43.19; Std. deviation = 164.353). However, this group accounts for a relatively small percentage of the overall developer community in each project. But a close examination of the percentage of developers involved in posting and committing shows that projects with small number of posters will also have a small number of committers. This is valid in 60% (N = 12) of the projects studied. There is a 50-50 split (20%; N=4 on either side) between projects with small percentage of posters but large percentage of committers (Balsa, Epiphany, Evolution, and Nautilus) and those with large percentage of posters but a smaller percentage of committers.

The analysis supports our first hypothesis (H1) that developers are making more commits to SVN (mean = 150.32, Std. deviation = 424.986) than they are posting messages to the developers mailing lists (mean = 42.63, Std. deviation = 161.852). Furthermore, a low but significant correlation (ρ = .0426; p = 1.000) between developers commits and posting patters supporting our second hypothesis (H2) that developers are contributing equally to code repositories and mailing lists. Wilcoxon signed ranks for the Two-Related-Samples Tests revealed that only 35 developers (less than 10%) had a balanced or tie activity.

The implications of these findings may provide assurance that FOSS developers, apart from coding and committing bits of code to a project's SCM, they are also involved in knowledge brokage [SAS06] in mailing lists. We can conjecture from earlier findings [SAS06, ARG06, Lon06] and our experience in both the FLOSSMetrics[3] and SQO-OSS[4] projects that this serendipity has implications for the quality of code since a large number of developers are externalizing and discussing their coding activities with other community members in the mailing lists. This kind of engagement may enable the developers to improve the quality of their code base, do more refactoring and learn about how the quality of the produced code may be improved.

[3] http://www.flossmetrics.org/

[4] http://www.sqo-oss.org/home

Future work and research directions: As a follow up to this research, our future work aims at consolidating understanding developer dynamics and the development of appropriate community metrics [SC09] or indicators for the certification of both the FOSS product and process. Thus, narrowing the gap which exist in FOSS certification and formal methods [CS08]. Specifically, we plan to add a qualitative element to our research by interviewing some of the α [VTG$^+$06] or star [SSL08] or key developers. This future work may also incorporate content analysis of the postings, new metrics like posts/commits and how such metrics vary overtime. This kind of data, metrics and commits analysis may help us better understand the quality of developers contribution, reveal any bottlenecks which may hinder the incorporation of developers code into the release product, and further reveal what kinds of metrics may be most appropriate when characterizing FOSS developers and projects.

Furthermore, in addition to SVN and mailing lists, developers also contribute intensively to the bug reporting and fixing process. Therefore, there exist an avenue of extending the methodology presented in this research to incorporate data from bug tracking systems data. This will provide a more comprehensive view of the pattern of developers contribution in open source projects. While the conclusion drawn from this study points out certain trends in Gnome projects, we are working on extracting a more heterogenous sample of projects and apply the same methodology to see if the patterns observed here can be generalized to other FOSS projects, not specifically Gnome based.

Acknowledgements: We are grateful to all partners in the FLOSSMETRICS project for providing access to the data and tools used in this study.

Bibliography

[ARG06] J. Amor, G. Robles, J. Gonzalez-Barahona. Discriminating Development Activities in Versioning Systems: A Case Study. In *Proceedings PROMISE 2006: 2nd. International Workshop on Predictor Models in Software Engineering co-located at the 22th International Conference on Software Maintenance (Philadelphia, Pennsilvanya, USA)*. 2006.

[BGD$^+$06] C. Bird, A. Gourley, P. Devanbu, M. Gertz, A. Swaminathan. Mining email social networks. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*. Pp. 137–143. ACM Press, New York, NY, USA, 2006.

[Bro75] F. Brooks. *The Mythical Man-Month. Essays on Software Engineering.* Addison-Welsey Publishing, 1975.

[CLM03] A. Capiluppi, P. Lago, M. Morisio. Characteristics of Open Source Projects. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*. P. 317. IEEE Computer Society, Washington, DC, USA, 2003.

[CS08] A. Cerone, S. A. Shaikh. Incorporating Formal Methods in the Open Source Software Development Process. In *2nd International Workshop on Foundations and Techniques for Open Source Software Certification*. Milan, Italy, 10 September 2008 2008.

[DB05] T. T. Dinh-Trong, J. M. Bieman. The FreeBSD Project: A Replication Case Study of Open Source Development. *IEEE Transactions on Software Engineering* 31(6):481–494, 2005.

[DB07] J. M. Dalle, M. den Besten. Different Bug Fixing Regimes? A Preliminary Case for Superbugs. In Feller et al. (eds.), *Open Source Development, Adoption and Innovation*. IFIP International Federation for Information Processing 234, pp. 247–252. Springer, September 7-10 2007.

[GKS08] G. Gousios, E. Kalliamvakou, D. Spinellis. Measuring developer contribution from software repository data. In *MSR '08: Proceedings of the 2008 international workshop on Mining software repositories*. Pp. 129–132. ACM, 2008.

[Kuk06] G. Kuk. Strategic Interaction and Knowledge Sharing in the KDE Developer Mailing List. *MANAGEMENT SCIENCE 2006 52: 1031-1042.* 52:1031–1042, 2006.

[Lon06] J. Long. Understanding the Role of Core Developers in Open Source Development. *Journal of Information, Information Technology, and Organizations* 1:75–85, 2006.

[Mas05] B. Massey. Longitudinal analysis of long-timescale open source repository data. In *PROMISE '05: Proceedings of the 2005 workshop on Predictor models in software engineering*. Pp. 1–5. ACM, New York, NY, USA, 2005.

[MFH02] A. Mockus, R. Fielding, J. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *Transactions on Software Engineering and Methodology.* 11(3):1–38, 2002.

[MFT02] G. Madey, V. Freeh, R. Tynan. The open source software development phenomenon: An analysis based on social network theory. In *Americas conf. on Information Systems (AMCIS2002)*. Pp. 1806–1813. 2002.

[Nor04] M. Norusis. *Statistical Procedures Companion*. Prentice Hall, Inc., 2004.

[QJ04] S. R. Q. Jones, G. Ravid. Information overload and the message dynamics of online interaction spaces: a theoretical model and empirical exploration. *Information System Research* 15 (2):194210, 2004.

[RG06] G. Robles, J. Gonzalez-Barahona. Contributor Turnover in Libre Software Projects. In Damiani et al. (eds.), *IFIP International Federation for Information Processing, Open Source Systems*. Volume 203, pp. 273–286. Springer,Boston, 2006.

[RGCH09] G. Robles, J. Gonzalez-Barahona, D. Cortazar, I. Herraiz. Tools for the study of the usual data sources found in libre software projects. *International Journal of Open Source Software and Processes* 1(1):24–45, Jan-March 2009.

[SAOB02] I. Stamelos, L. Angelis, A. Oikonomou, G. Bleris. Code Quality Analysis in Open-Source Software Development. *Information Systems Journal, 2nd Special Issue on Open-Source, Blackwell Science* 12 (1):43–60, 2002.

[SAS06] S. K. Sowe, L. Angelis, I. Stamelos. Identifying Knowledge Brokers that Yield Software Engineering Knowledge in OSS Projects. *Information and Software Technology* 48:1025–1033., 2006.

[SASM07] S. K. Sowe, L. Angelis, I. Stamelos, Y. Manolopoulos. Using Repository of Repositories (RoRs) to Study the Growth of F/OSS Projects: A Meta-Analysis Research Approach. In *Open Source Development, Adoption and Innovation*. IFIP International Federation for Information Processing 234/2007(978-0-387-72485-0), pp. 147–160. Springer Boston, August 2007.

[SC09] S. A. Shaikh, A. Cerone. Towards a metric for Open Source Software Quality. *Electronic Communications of the EASST* Volume 20: Foundations and Techniques for Open Source Certification 2009(ISSN 1863-2122), 2009.

[Sek06] U. Sekaran. *Research Methods for Business: A Skill Building Approach*. Wiley, 4th edition, 2006.

[SFF$^+$06] W. Scacchi, J. Feller, B. Fitzgerald, S. A. Hissam, K. Lakhani. Understanding Free/Open Source Software Development Processes. *Software Process: Improvement and Practice* 11(2):95–105, 2006.

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

[SO09a] SQO-OSS. Novel Quality Assessment Techniques. Deliverable Report-D7. Technical report, Software Quality Observatory for Open Source Software. Project Number: IST-2005-33331, 29 June 2009.
http://www.sqo-oss.eu/research/reports/SQO-OSS_D_7_final.pdf

[SO09b] SQO-OSS. Overview of the state of the art. Deliverable Report-D2. Technical report, Software Quality Observatory for Open Source Software. Project Number: IST-2005-33331, 29 June 2009.
http://www.sqo-oss.eu/research/reports/SQO-OSS_D_2_final.pdf

[SSL08] S. K. Sowe, I. Stamelos, A. Lefteris. Understanding Knowledge Sharing Activities in Free/Open Source Software Projects: An Empirical Study. *Journal of Systems and Software* 81(3):431–446., 2008.
doi:10.1016/j.jss.2007.03.086

[SSSA08] S. K. Sowe, I. Samoladas, I. Stamelos, L. Angelis. Are FLOSS developers committing to CVS/SVN as much as they are talking in mailing lists? Challenges for Integrating data from Multiple Repositories. In *3rd International Workshop on Public Data about Software Development (WoPDaSD). September 7th - 10th 2008, Milan, Italy.* 2008.
http://www.slideshare.net/sksowe/implications-of-dual-participation-of-floss-developer

[VTG+06] S. Valverde, G. Theraulaz, J. Gautrais, V. Fourcassie, R. V. Sole. Self-Organization Patterns in Wasp and Open Source Communities. *IEEE Intelligent Systems* 21(2):36–40, 2006.
doi:http://doi.ieeecomputersociety.org/10.1109/MIS.2006.34

[ZPZ07] T. Zimmermann, R. Premraj, A. Zeller. Predicting Defects for Eclipse. In *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. P. 9. IEEE Computer Society, Washington, DC, USA, 2007.

Open Source Verification under a Cloud

Peter T. Breuer[1] **and Simon Pickin**[2]

[1] ptb@cs.bham.ac.uk
Dept. Comp. Sci., University of Birmingham, Birmingham, UK

[2] spickin@it.uc3m.es
Depto. Ing. Telemática, Universidad Carlos III, Leganés, Madrid, SPAIN

Abstract: An experiment in providing volunteer cloud computing support for automated audits of open source code is described here, along with the supporting theory. Certification and the distributed and piecewise nature of the underlying verification computation are among the areas formalised in the theory part.

The eventual aim of this research is to provide a means for open source developers who seek formally backed certification for their project to run fully automated analyses on their own source code. In order to ensure that the results are not tampered with, the computation is anonymized and shared with an ad-hoc network of volunteer CPUs for incremental completion. Each individual computation is repeated many times at different sites, and sufficient accounting data is generated to allow each computation to be refuted.

Keywords: Formal Methods, Software Verification, Static Analysis, Open Source, Cloud Computing, Distributed Computation

1 Introduction

We have developed a fledgling volunteer cloud computing system for the formal verification and static analysis of large open source software code bases, and performed experiments on some millions of lines of C code with it. What has motivated this development is the vision of a future in which a formal verification problem can be sent out to a cloud of volunteer solvers somewhere on the Internet for completion. Hopefully, those supporters of an open source project who do not have the skills to provide first-hand help to the developers will contribute by lending their CPU cycles to the task of certifying their favourite new release free from certain classes of semantic errors – or detecting them if they exist. They might also contribute extra regression tests or novel verification procedures.

In this kind of framework, the bottleneck presented by the certification authority in a traditional approach is removed. Moreover, the abundance of available CPU cycles allows the calculation to be duplicated many times over for reliablity, while enough intermediate results are stored for accounting processes to check the computation as may be required. Our prototype software provides a skeleton for a possible ‘open certification’ method, in other words. While neither the design nor the implementation is perfect and complete, it is to be hoped that the initiative stimulates better efforts and further progress in this direction.

An ‘open verification cloud’ as prototyped here physically consists of a database back-end and its servers plus volunteer clients running the bespoke verification solver software. The clients have volunteered to help perform the computations that resolve the verification problems stored on the cloud’s database. The cloud-computation nature of the process is manifested in the fact that no client knows about the other clients currently helping the computation and none knows where the servers are physically located.

The code treated in the work reported here is ANSI C [11, 12] with embedded assembler, and no significant restrictions. There is no inherent limitation to a particular language, however. It is of course universally realized that (unrestricted) C is an inherently intractable candidate for verification because of its indirections via pointers and other infelicitous language features, and those obstacles are overcome in this approach by using deliberately approximate (but sound) verification logic [7, 8, 9].

1.1 Context and related work

The verification technology used in the work reported here falls in the class of ‘lightweight’ verification technologies. It is based at the top level on a symbolic programming logic [9] and at the very bottom level on decision procedures using mixed integer linear programming implemented using the GNU Linear Programming Kit (GLPK). The GLPK is intended for solving large-scale linear programming, mixed integer programming, and other related problems. It is a set of routines itself written in ANSI C and organised as a library. It is available as part of the GNU Project and is released under the GNU General Public License [5, 19]. That is particularly appropriate here because the principal target for our technology has historically been the open source C code of the Linux operating system kernel (see for example [7]).

Other lightweight verification technologies in the same class include Splint [17] (derived from Larch [20]), also ESC/Java and Spec# [6]. All these tools make some sacrifices in the area of completeness or precision in order to be useful on the undecorated original source codes, and some require expert annotations to be added to the source. And while the C language is always a particularly difficult target for such technologies, some notable attempts at it have been made.

David Wagner and collaborators in particular have been active in the area (see for example [26], where Linux user space and kernel space memory pointers are given different types, so that their use can be distinguished, and [27], where C strings are abstracted to a minimal and maximal length pair and operations on them abstracted to produce linear constraints on these numbers). That research group often uses model-checking [13].

Their approach in [27, 26] makes use of both model-checking and abstract interpretation [14] (abstraction is used in general in order to ‘airbrush out’ the more unsavoury aspects of C from the formal view of it), and therefore contrasts with contributions like Jeffrey Foster’s work with CQual [18], which seek to extend the type system of C in a more controllable direction. In particular, CQual has been used to detect “spinlock-under-spinlock”, a sub-case of one of the analyses routinely performed by the tools used in the experiment reported here.

The SLAM project [4] originating at Microsoft also analyses C programs using a mixture of model-checking, abstract interpretation and deduction. That technology is intrinsically an order or more of magnitude slower than the basic technology used in the experiment described here, but it also works by creating an abstraction of the program code, and it also generates intermediate state descriptions mechanically.

The Coverity checker [16] has also come to be used In the context of the Linux kernel source code. Coverity is a commercial tool based on an user-extensible version (a *meta-compiler*) of the GNU C compiler, *gcc* [21]. Coverity itself is proprietary, and its innards are not accessible to review, but it may be guessed that the staff of the company can configure into the compiler framework any finite state machine-based computation for the purposes of a custom analysis that they have in mind. It is a less abstract technological solution than the one used here, but shares with it the characteristic of customizability.

Efforts to distribute verification computations to a large number of solvers organised in a well-defined topology are made regularly – see [1], for example – and it is a recognised conference topic. Researchers have particularly sought to distribute model-checking problems onto grid-based machinery. Holzmann defines the notion of 'swarm verification' to describe the technique [23], adapting the SPIN [22] model-checking tool to the paradigm. In passing, it may be noted that the verification technology used here seems to be part of a recent trend observed by Holzmann in [24] towards verification of an abstraction of the actual code rather than of a design model. However, the work reported here aims to accommodate the lower performance targets obtainable from zero-cost volunteer CPU cycles available out on the Internet.

Existing infra-structure projects support so-called 'volunteer computing' -type projects. See for example the BOINC software [2, 3] from Berkeley. It is not clear at the time of writing if that software would have been a significant aid to our exploratory project, because BOINC clients expect a single data file and return a single result file to the database server, rather than engaging in a substantially continuous interchange, as is the case here. Nevertheless, it may in the future be very helpful in the organisation of the architecture in a full-scale project, particularly in terms of the organisation of the permissions for access and the classification of the provenance and reliability of the data returned.

Peter Lee [25] has approached the problem of automatically checking the trustworhiness of machine code to be executed by an operating system. The idea in *proof carrying* approaches is that incoming code snippets carry a proof that a desired security property is satisfied, and the operating system automatically checks the syntactic relationship of the machine code to the proof. Our approach is to check the source code instead.

1.2 Contents

This article is organised as follows. Section 2 formally describes the process of certification from the top down. After describing certification properties, the section describes in more detail the process of analysis that produces the certification here, showing in particular how the calculation is adapted to the exigencies of a part-time volunteer cloud computing context. Section 3 succinctly presents the programming logic used in order to provide a self-contained account of the technology here, and readers may wish to skip that section if they are not interested in formal logic. Section 4 describes an experiment performed on about a million lines of C source code. That experiment was previously conducted using monolithic analysis tools [8] and it has been repeated in the volunteer cloud computing trial [10].

2 Certification

In this section a global view of the certification process is set out and related to the procedure implemented.

2.1 Certification in the abstract

This section describes formally what certification means as implemented in the prototype project. Firstly, it is a process and it produces a result and a certificate of the result. Secondly, the certificate has the property that it can be checked to have been generated by following the purported process applied to the purported source code, generating the purported result. Thirdly, the process and its result accords certain guarantees to the certified code.

Consider then the certification process in the abstract. An automated procedure M takes a software code base C and, in the presence of a list L of known defects, produces a certificate X that says that the list is complete. That is, the process takes code C and (sometimes – the alternative is that the certification process fails) produces a certificate X:

$$CM = X$$

Moreover, if L_d is the sub-list of L of defects of kind d, and we write d_p to mean that there is a defect of kind d at point p in the code, then L_d contains all the points p in the code at which a defect of kind d arises. That is:

$$\{p \in C \mid d_p\} \subseteq L_d$$

Putting those two together, one gets a fundamental description of what certification means:

$$X = CM \;\Rightarrow\; \forall p \in C - L_d : \neg d_p \tag{1}$$

I.e. code that is has more defects than stated does not get a certificate.

2.2 What is a defect?

In our implementation, a defect d_p is defined by a condition expressed in symbolic logic as $D_p(\mathbf{x})$ that is deduced to be possibly reachable after p. That is, logical analysis deduces a post-condition

$$\ldots\; p \;\{\; \text{post}_p \;\}$$

for p and checking the formula using a model-based technique shows there is a non-empty intersection of the post-condition with $D_p(\mathbf{x})$. That is:

$$d_p \;\Leftrightarrow\; \exists \mathbf{x}.\, D_p(\mathbf{x}) \wedge \text{post}_p \tag{2}$$

for some values of the logical variables $\mathbf{x}$. An example follows in Section 2.3 immediately below.

2.3 Example

An example of an interesting defect condition is

$$D_p^{\text{ex}} = \begin{cases} x > 1, & p \text{ a lock call} \\ x \leq 0, & p \text{ an unlock call} \\ \text{false}, & \text{otherwise} \end{cases} \tag{3}$$

where x is a logical (i.e., non-program) variable which counts the number of stacked locks taken in the program. x is incremented by lock calls and decremented by unlock calls. The pre-/post-condition logic describing x for the analysis is:

$$\begin{array}{l} \{\phi[x+1/x]\}\ \textbf{lock()}\ \{\phi\} \\ \{\phi[x-1/x]\}\ \textbf{unlock()}\ \{\phi\} \end{array} \tag{4}$$

Where this particular defect condition checks out as feasible in the sense of (2), it indicates either that (a) a lock might have been taken twice by that point without a release between the two takes; or (b) a lock may have been released twice by that point without a lock attempt between. These defects d_p^{ex} can by definition (3) only be detected at the sites of a lock or unlock call p. Certification in this case means that the code C has been scanned and defects d_p^{ex} have been ruled out. That is, no lock can be taken twice in a row, nor unlocked twice in a row, without an unlock, respectively a lock, operation occurring between the two.

2.4 False positives

Note that there may be codes C which are flagged as having a defect in the sense of (2) but which are nevertheless semantically correct ('false positives'), in the sense of never in practice triggering the condition $D_p(\mathbf{x})$.

That is the rationale for in practice maintaining a list L_d of detected defect sites – they have individually to be signed off by the developer as 'false positive' or 'noted for correction in the next release', or 'noted but no solution yet'. The certificate certifies that it is unequivocally the case that the defined defect cannot arise anywhere other than the sites listed.

False positives generally fall into two classes. In the first class, a guard condition such as $y^2 < 0$ cannot in practice be breached, but the analysing logic does not know that, and explores a factually impossible code trace as though it were possible. That kind of semantic 'inexactness' is a result of the deliberately approximating nature of the symbolic logic used in any real life analysis. The analysing logic has to be less exact ('more alarmist') than reality or the computation would never finish in practice.

A typical instance of the second class of false positive arises naturally in the context of the example above in Section 2.3. It occurs when two *different* locks are taken in sequence in the code, without an unlock between them. A defect will be detected. The fault here is purely a definitional one. The situation is factually harmless in itself, but it is captured by the defect definition. The problem may be said to be rooted either in the poverty of the analysis language – different counts for different locks may be difficult to define – or in the poverty of the analysis logic – one may not be able to reliably distinguish references to different locks in C. The latter is the case here. Different pointers may point to the same underlying lock, and the same pointer may point to different locks at different times.

2.5 Accountability

It is important for a certification procedure that it can be checked that the certificate X produced relates the certified code C and the method M used to certify it. That is, there is a checking procedure K such that

$$K(X,C,M) = \begin{cases} \text{true}, & \text{if } X = CM \\ \text{false}, & \text{otherwise} \end{cases} \tag{5}$$

How is that guaranteed?

The answer is, in our procedure, via digital signatures. A digital signature $\sigma(T^\ddagger)$ is generated for: (a) a printout of intermediate results $T^\ddagger$ from the analysis in M; (b) another signature $\sigma(\mathscr{A})$ is generated from the short ASCII file that configures the analysis; (c) another $\sigma(\mathscr{H})$ is generated from the ASCII file that expresses the defect condition(s) being scanned for; (d) another signature $\sigma(L)$ is generated for the list of allowed defect exceptions L; (e) another signature $\sigma(C)$ is generated for the code C; (f) a signature $\sigma(\mathscr{P})$ is generated from the file that configures the code parse $\mathscr{P}$. Those digital signatures comprise X, as will be detailed in the following paragraphs.

Then, provided the code developer holds on to a copy of the intermediate results, a copy of the analysis method configuration, and a copy of the original code, then any part of the computation via M can be repeated at will for the benefit of anyone that doubts it. That is the procedure K, modulo checking the digital signatures to confirm the veracity of the three components. That is to say

$$K(X,C,M) \Leftrightarrow CM = X$$

as required in (5). That means that

$$K(X,C,M) \Leftrightarrow \forall p \in C : \neg d_p \vee p \in L_d$$

according to (1). The important idea here is that a part of the calculation can be repeated as needed in order to check the result, and that in order to be sure that the repeated calculation starts from the right place (and finishes in the right place) the digital signatures in the certificate are necessary – as is the data signed, but that has to be stored separately. It is not present in the certificate. Where the data is kept is a separate question.

To explain how the calculation can be reconstructed when required, the calculation needs first to be described in more detail. The certification method M consists first of a parse $\mathscr{P}$ to give a syntax tree T:

$$T = C\mathscr{P} \tag{6}$$

Next an analysis $\mathscr{A}$ is applied to the tree T to decorate it with symbolic logic expressions, giving the decorated tree $T^\dagger$:

$$T^\dagger = T\mathscr{A} \tag{7}$$

Then a checker $\mathscr{H}$ is applied which further decorates the tree with evaluations of the logic to see if defects are feasible:

$$T^\ddagger = T^\dagger\mathscr{H} \tag{8}$$

The list of sites p within the code C at which defects d_p are detected here is what is basically of interest to developers and consumers alike, and it is supposed to be covered by the list L of knowns.

$$L_d \supseteq \{p \in C : d \text{ decoration of } T^{\ddagger} \text{ at } p \text{ is not false}\} \tag{9}$$

The certificate X consists exactly of the digital signatures of the code, (printed out) tree decorations, and the configuration used for the parser, for the analysis and for the checker, and the list of known defects:

$$X = (\sigma(C), \sigma(\mathscr{P}), \sigma(T^{\ddagger}), \sigma(\mathscr{A}), \sigma(\mathscr{H}), \sigma(L)) \tag{10}$$

Every step of this procedure can be repeated unambiguously. For example, to get to $T^{\ddagger}$, one needs to repeat at least the step (8). That starts from $T^{\dagger}$. But $T^{\dagger}$ is just $T^{\ddagger}$ with some of the decoration dropped. So it can be unambiguously obtained from $T^{\ddagger}$, which is signed. The configuration for $\mathscr{H}$ is signed and available, and so $\mathscr{H}$ can be applied unabiguously to check the derivation of $T^{\ddagger}$ from $T^{\dagger}$.

2.6 Analysis and evaluation

The analysis procedure $\mathscr{A}$ is organised in detail according to the structure of the code. It generates a pre-/post-condition pair (it is more than a single pair as there are various different postconditions corresponding to the different kinds of exit flow available to the code - see the interior of this article for more detail) for each program fragment p:

$$\{\text{pre}_p\}\ p\ \{\text{post}_p\}$$

The pair is computed from the results for the component fragments $p_i \in p : p = P(p_i)$ where P is the constructor (`if`, `while`, etc.) that produces p from the p_i:

$$\{\text{pre}_{p_i}\}\ p_i\ \{\text{post}_{p_i}\}$$

and

$$(\text{pre}_p, \text{post}_p) = [P](\text{pre}_{p_i}, \text{post}_{p_i})$$

where $[P]$ is an appropriate generator of symbolic logic. It is specified for the source language (usually C) being treated in the configuration file for the analysis. ‘Appropriately’ here means that the logic is sound with respect to the semantics of the language, in that for each pre-/post-condition pair generated:

$$\text{pre}_p \ \Rightarrow\ \mathbf{wp}[p](\text{post}_p) \tag{11}$$

where **wp** is the semantic *weakest precondition* constructor for the language.

That (11) is not an equivalence means that the symbolic logic generated by this scheme is *approximate* (but sound, according to (11)). That gives rise to the name *symbolic approximation* [9] for the general technique.

Note that some complexity reduction *is* performed by our tools via lightweight automatic theorem-proving techniques at the stage of producing the tree $T^\dagger$ with the symbolic logic annotations. For example, a formula of the form

$$p \wedge q$$

will be reduced to q if $p \rightarrow q$ is proved on the fly as the formula is generated. Similarly for $p \vee q$. That has proven very effective in reducing complexity. What our tools are not good at is reducing formulae of the general shape $\bigvee_i \bigwedge_j q_{ij}$ to a simpler expression p when there is one, such as in the case of $p \wedge q \vee p \wedge \neg q$. The inadvertent and unrecognised splintering of simple logical expressions into multiple complex cases in this style is the most significant source of the computational explosions that are occasionally encountered during processing. In principle the situation could be detected and repaired at the checking stage of the process when $T^\ddagger$ is generated (all the atomic propositional forms here are linear inequalities and one could detect when dropping one failed to relax the problem), but that is not done, because the extra computation is usually prohibitively expensive and apparently only rarely productive in practice.

In the final phase that produces $T^\ddagger$, the volunteer clients in the cloud apply a modelling technique to decide whether

$$\mathrm{post}_p \wedge D_p(\mathbf{x})$$

is satisfiable at any node p of the abstract syntax tree. Since all questions of satisfiability for the predicates in our logic can be reduced to questions of the feasibility of systems of linear inequalities in integer variables, the evaluation is performed using mixed linear integer programming, supported by open source libraries such as GNU's Linear Programming Kit (GLPK).

2.7 The cloud computation

A non-negative answer to the question asked by the model-theoretic evaluation procedure indicates a *possible* defect d_p. The analysis $\mathscr{A}$ and evaluation calculations $\mathscr{H}$ are incremental, stateless, and can be broken off and restarted from the break-off point, as well as repeated either partially or wholly. That is the basis for performing the computation via a cloud and the following paragraphs describe the properties that permit that implementation in formal terms.

Let the constructs p (the nodes and leaves of the abstract syntax tree T produced by the parse) that appear in the code C be $p = P(p_i)$ for a syntactic constructor P and components p_i. The constructions define a *dependency* pre-order:

$$p = P(p_i) \;\Leftrightarrow\; p_i < p \tag{12}$$

which extends uniquely to a minimal partial order via transitivity. In the partial order, one code construct 'depends on' (is greater than) another if the second is a component or sub-component of the first. For example, `if(x<0){ x++; y++; }` depends on the component `x++; y++` and on its component `x++`.

The operations $\mathscr{A}$ and $\mathscr{H}$ can then be split up as follows:

$$\mathscr{A} = \mathop{\circ}_p \mathscr{A}_p \tag{13}$$

$$\mathscr{H} = \mathop{\circ}_p \mathscr{H}_p \tag{14}$$

where the order of the compositions is constrained only by the dependencies $p_i < p$. Formally, operations on different parts of the tree can be performed in any order:

$$\mathscr{A}_{p_1} \circ \mathscr{A}_{p_2} = \mathscr{A}_{p_2} \circ \mathscr{A}_{p_1} \tag{15}$$
$$\mathscr{H}_{p_1} \circ \mathscr{H}_{p_2} = \mathscr{H}_{p_2} \circ \mathscr{H}_{p_1} \tag{16}$$

where $p_1 \not\leq p_2$ and $p_2 \not\leq p_1$ in the dependency relationship. Also, since $\mathscr{A}$ and $\mathscr{H}$ work on different decorative features on the tree

$$\mathscr{A}_{p_1} \circ \mathscr{H}_{p_2} = \mathscr{H}_{p_2} \circ \mathscr{A}_{p_1} \tag{17}$$

whenever $p_1 \neq p_2$. When $p_1 = p_2$ then $\mathscr{H}$ requires the decoration produced by $\mathscr{A}$ first.

In practice, the computation of the symbolic logic forms and their evaluation is performed at the same time, because the former is usually a computationally cheap task relative to the latter. That is, the computation

$$\mathscr{A} \circ \mathscr{H} = \underset{p}{\circ}(\mathscr{A}_p \circ \mathscr{H}_p) \tag{18}$$

is performed. (15, 16, 17) justify the reordering of the components in (18).

That the computation can be broken off and restarted means only that (18) can be further reordered via (15, 16, 17) as

$$\mathscr{A} \circ \mathscr{H} = \underset{p\in P}{\circ}(\mathscr{A}_p \circ \mathscr{H}_p) = \underset{p\in P_1}{\circ}(\mathscr{A}_p \circ \mathscr{H}_p) \underset{p\in P_2}{\circ}(\mathscr{A}_p \circ \mathscr{H}_p) \tag{19}$$

where P_1, P_2 is a partition of the full set of code fragments $P = P_1 \uplus P_2$ such that

$$p_1 \in P_1 \wedge p_2 \in P_2 \;\Rightarrow\; p_2 \not\leq p_1 \tag{20}$$

I.e., P_1 already contains all the predependencies $p_2 < p_1$ for any $p_1 \in P_1$. P_1 is the set of code fragments that have been completely analyzed at the time of the break, and P_2 is the remainder at that time.

Moreover, the computation can be broken off and re-started any number of times. That is, the equation (19) may be extended to match with any partitioning $P = P_1 \uplus \cdots \uplus P_n$ that respects the dependency order, as in (20). P_1 is the set of code fragments completely analysed at the time of the first break, $P_1 \uplus P_2$ the set completely analysed at the time of the second break, and so on. The enabling conditions are (15, 16, 17).

3 Logic

Readers uninterested in formal logic may wish to skip this section, which is included to provide a self-contained account here. It is not needed by what follows after.

The program logic used to generate the assertions which decorate the syntax tree $T^\dagger$ from (7) is called NRBG, for ‘normal, return, break, goto’, the principle kinds of program flow treated. In a program there is the normal program flow, which passes from the beginning of a statement through to its (normal) end, and there are exceptional flows, the break, return and goto flows (and also others in other languages), which exit a statement before it ends normally. The logic considers the interaction of these flows though each code construct.

For example, the rule for sequential statements states that either $a;b$ may terminate normally with condition r or it may terminate exceptionally with condition x. On the way to doing so, a may either terminate normally with condition q and b continue from q to the required termination conditions, or else a may terminate exceptionally with condition x right away. That is:

$$\frac{\{p\}\ a\ \{Nq \vee \mathscr{E}x\} \quad \{q\}\ b\ \{Nr \vee \mathscr{E}x\}}{\{p\}\ a;b\ \{Nr \vee \mathscr{E}x\}}$$

where $\mathscr{E}$ stands for any of R, B, G_l, provided l is not a label defined in a or b.

The logic has been presented and explained many times over the years. See for example [8]. The presentation given here is innovative in that it introduces the N, R, B, G_l as modal operators. The earlier presentations used a set of interacting logics N, R, B, G. The advantage of the new presentation is that the number of logical rules falls to about seven from about twenty.

The rule for a do-forever loop says that breaking from the body of the loop with condition q is the same as terminating the loop normally with q. That is:

$$\frac{\{p\}\ a\ \{Bq \vee Np \vee \mathscr{E}x\}}{\{p\}\ \texttt{while(true)}\ a\ \{Nq \vee \mathscr{E}x\}}$$

where $\mathscr{E}$ stands for any of R, G_l, where l is not a label defined in a. The rule also captures the idea that exiting exceptionally in another way than through break (that is, with either return or goto) from the body of the loop with condition x means exiting the whole loop with condition x too. Experts in logic should note that the normal termination condition p for the body of the loop that appears in the rule is a *fixpoint*, and finding a useful fixpoint (lower than 'true') in practice is a non-trivial feat of leger-de-main.

Exceptional modal conditions R, B, G_l are generated uniquely by the corresponding statements, `return`, `break` and `goto` respectively:

$$\frac{}{\{p\}\ \texttt{return}\ \{Rp\}} \qquad \frac{}{\{p\}\ \texttt{break}\ \{Bp\}} \qquad \frac{}{\{p\}\ \texttt{goto}\ l\ \{G_l\,p\}}$$

The rule for conditionals is, unsurprisingly:

$$\frac{\{p \wedge c\}\ a\ \{Nq \vee \mathscr{E}x\} \quad \{p \wedge \neg c\}\ b\ \{Nq \vee \mathscr{E}x\}}{\{p\}\ \texttt{if(c)}\ a\ \texttt{else}\ b\ \{Nq \vee \mathscr{E}x\}}$$

where $\mathscr{E}$ stands for any of R, B, G_l, where l is not a label defined in a or b.

A suitable assignment rule is always:

$$\frac{}{\{q[e/x]\}\ x = e\ \{Nq\}}$$

but in practice some weaker rule is usually implemented, with special cases that depend on the form of the expression e. From the point of view of the correctness of the logic, it does not matter what weaker rule is implemented because it will be sound. The practical effect of a weaker implementation is eventually to generate more 'false positive' alerts for defects than would otherwise have been the case. For example, non-linear update expressions are typically described in practice by very approximate logic such as:

$$\frac{}{\{x > 0 \wedge |x||y| < 2^{31}\}\; x = x * y\; \{\mathrm{sign}(y) = \mathrm{sign}(x)\}}$$

(the preconditions avoid overflow).

The rule is for a labelled statement $l : b$ says that an initial condition p is required that is the same as the exit condition p from all the `goto` l statements within b. p is a fixpoint:

$$\frac{\{p\}\; b\; \{Nr \vee \mathscr{E}x \vee G_l\, p\}}{\{p\}\; l:\; b\; \{Nr \vee \mathscr{E}x\}}$$

Compare the rule for while forever loops.

A further rule for labelled statements deals with the more general situation where a label occurs in the middle of a sequence of statements $a; l : b$, rather than at the beginning. There it is the case that the entry condition q for $l : b$ must not only be the normal exit condition from a, but also the condition that arises from the 'forwards pointing' `goto`s within a, as well as the 'backwards pointing' `goto`s in b:

$$\frac{\{p\}\; a\; \{G_l\, q \vee Nq \vee \mathscr{E}x\} \qquad \{q\}\; b\; \{Nr \vee \mathscr{E}x \vee G_l\, q\}}{\{p\}\; a; l : b\; \{Nr \vee \mathscr{E}x\}}$$

where $\mathscr{E}$ stands for any of R, B or $G_{l'}$ where l' is not a label defined in a or b.

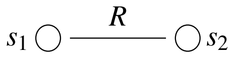
There is a more convenient way to deal with the `goto` computations. It consists of loading the rules with prior 'assumptions' $G_l\, p_l$ (written to the left of a $\triangleright$ in Gentzen style) about the exit condition p_l that will be imposed by the `goto` l statements encountered within the program. The initial estimates are modified upwards by the conditions p found at the sites where the `goto`s are located in the program. The initial guess p_l needs to be loosened to $p \vee p_l$, and so on round and round until a fixpoint is found. A `goto` fixpoint achieved in practice in an implementation is not usually the least fixpoint, but it is generally a useful and nontrivial one.

$$\frac{p \;\Rightarrow\; p_l}{G_l\, p_l \;\triangleright\; \{p\}\; \texttt{goto}\; l\; \{G_l\, p\}}$$

The fixpoint p_l is available as an entry condition at the point in the code where the label l is sited:

$$\frac{G_l\, p_l \;\triangleright\; \{p_l\}\; a\; \{Nr \vee \mathscr{E}x \vee G_l\, p_l\}}{\triangleright\; \{p_l\}\; l:\; a\; \{Nr \vee \mathscr{E}x\}}$$

The model underlying the logic is of individual states s which assign values to the program variables, and links between them that are 'coloured' N, R, B or G_l according to whether the transition is as a result of respectively a normal program termination, hitting a return statement, hitting a break statement, or hitting a goto. The following diagram is of a R-coloured ('return coloured') transition from state s_1 to state s_2:



Each state has only one exit in the present (deterministic) setting, but there may be many entries to any state.

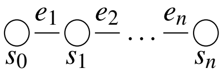
The semantics of the N, R, B, G_l operators is that, for example, a modal statement like Rp holds at the *pair* of a state s_2 *and* an arc e *entering* the state. For example, Rp holds at (e,s_2) if p holds at s_2 and e is coloured with R. In general:

$$(e,s) \models \mathscr{E}p \quad \Leftrightarrow \quad s \models p \wedge e \text{ is coloured by } \mathscr{E}$$

for $\mathscr{E}$ any of N, R, B, G_l.

An atomic programming language statement a causes a change from a state s_1 to a state s_2 via a link e that is of a 'colour' that is normally N, but is exceptionally R, B or G_l, depending as the statement executed in a is a return, break or goto respectively.

Suppose that for the (non-atomic) statement a, the sequence



is a sequence of states run through by the execution of a. Then $\{p\}\ a\ \{q\}$ means

$$\begin{aligned}&\{p\}\ a\ \{q\} \ \Leftrightarrow \\ &\quad \forall s_0,e_1,s_1,\ldots,e_n,s_n.\ p(s_0) \ \Rightarrow\ (e_n,s_n) \models q\end{aligned} \tag{21}$$

By convention, not specifying a 'colour' means that colours are ignored, i.e.:

$$p \ \Leftrightarrow\ Np \vee Rp \vee Bp \vee G_l p \vee \ldots \tag{22}$$

for all possible labels l in the program. Then (21) can more symmetrically be written as

$$\forall e_0,s_0,e_1,s_1,\ldots,e_n,s_n.\ (e_0,s_0) \models p \ \Rightarrow\ (e_n,s_n) \models q$$

Making (22) work requires a few axioms for the modal operators. Firstly, repetition of modal 'colouring' operators has no effect:

$$\mathscr{E}\,p \ \Leftrightarrow\ \mathscr{E}\,\mathscr{E}\,p \tag{23}$$

for $\mathscr{E}$ any of N, R, B, G_l. Also, an arc cannot be two colours at the same time:

$$\mathscr{E}_1\,p \wedge \mathscr{E}_2\,q \ \Rightarrow\ \text{false} \tag{24}$$

for $\mathscr{E}_1$, $\mathscr{E}_2$ from N, R, B, G_l and $\mathscr{E}_1 \neq \mathscr{E}_2$. Similarly

$$\mathscr{E}_1\,\mathscr{E}_2\,p \ \Rightarrow\ \text{false} \tag{25}$$

for $\mathscr{E}_1 \neq \mathscr{E}_2$. And colouring a (positive) formula is the same as colouring its parts:

$$\mathscr{E}\,(p \vee q) \ \Leftrightarrow\ \mathscr{E}\,p \vee \mathscr{E}\,q \tag{26}$$

$$\mathscr{E}\,(p \wedge q) \ \Leftrightarrow\ \mathscr{E}\,p \wedge \mathscr{E}\,q \tag{27}$$

for $\mathscr{E}$ from N, R, B, G_l. Together, (22), (23), (24), (25), (26), (27) mean that all modal formulae have the form

$$Np_N \vee Rp_R \vee Bp_B \vee G_l p_{G_l} \vee \ldots$$

for non-modal formulae p_N, p_R, etc.

4 Implementation and experiment

We report briefly here on our experience [10] in converting what were originally a set of monolithic semantic analysis tools [7, 8] for C code to the service of the volunteer cloud computing approach, and the re-running on the cloud of an experiment that had previously been run locally.

4.1 Populating the database

The first practical task for any submitter in presenting a problem to the cloud for solution consists of parsing the source code and storing the resulting syntax tree into the cloud's remote database. It is quite a considerable logistical problem, and it turned out to be too difficult to treat naively.

In our experiment [10], a million or so lines of Linux kernel source code was offered up for analysis, and that gave rise to over ten million syntax tree nodes. Each insertion involved several relational database updates on the (postgresql [15]) back-end and the acknowledgement and locking requirements slowed the transactions down to as much as a second or more across the network (the average time was a tenth of a second or so). There are only 86400 seconds in a day, and no developer is going to wait on the order of weeks to upload their problem. The efficiency might have been improved tenfold with effort, but it would have still been too slow for multi-million line source code bases.

This 'population problem' was eventually overcome by writing the parse data to a fast local non-relational data store (a GNU DBM 1.8.3 based store was used), then copying it to the remote database site in one lump, and converting it to relational database format in situ at the remote database. That got the job done in a day. It may be expected that incremental updates will comfortably handle new point releases of the source code base from there on. Attempts to use local and remote database replication pool services to upload the data in trickle mode failed. Each stream tended to stop completely while the database was otherwise in use, and the end result was slower overall. Clearly in the future source code will have to be uploaded whole to an extra cloud service from where it can be transferred into the database from close by.

4.2 Pruning the analysis

A single analysis task downloaded for solution by a volunteer client in the cloud usually consists in practice of the analysis of a single top-level functional unit. However, it turned out in our experiment that many of the function definitions from common header files had effectively been duplicated up to thousands of times through being declared *static* and *inline*, a combination which, in C, signals local scope and context at every implantation site. The number of analysis tasks was reduced tenfold overall by choosing to analyse only one representative from each class of syntactically identical functional definitions.

There is a potential problem in that the semantics of some of these apparent duplicates might have been modified unexpectedly by the differing contexts into which they were copied. It was supposed that that did not happen. The assumption was made that no two syntactically identical definitions captured identically named but different external references. This is a good assumption for well-written code, but it was not checked systematically. The numbers were certainly prohibitive - there were three quarters of a million top level function definitions in the database. Only seventy-two thousand of them corresponded to non-duplicates, and those were the ones eventually allowed to proceed to analysis.

4.3 Improving performance

Fetching data from the database in the cloud to a client as needed turned out to be far too inefficient as a general strategy. The latency of each database transaction was sufficient that the computation as a whole proceeded about one thousand times as slowly as it would have done on locally stored data (that experiment had already been tried [8]).

The situation was improved firstly by avoiding downloading syntax trees (node by node) in favour of downloading the relevant source code text in one lump and re-parsing it locally on the volunteer client. The issue of generating the same database keys locally as remotely was handled by storing an elaborated version of the source decorated with extra annotations, among them the in-database key for each identifier reference (each reference appeared in the elaborated text as ‘x@123456’, where ‘x’ was the label in the original source, and ‘123456’ the primary database key indexing the reference to ‘x’ at that line and column in that source code file). That provided enough information to generate all the other database keys too.

Secondly, a persistent cache was added on the client side just atop the database interface. The cache scored hits around the 95% level, with the corresponding order-of-magnitude-and-more speed-up.

Thirdly, the few database interactions that turned out to take minutes each – queries involving complex searches and aggregates across millions of database entries, such as calculating new priorities for the remaining work tasks after each task completion by a volunteer client – were amortised by calculating up to five hundred results ahead of time and then doling them out as needed. That implied that work task priorities in particular were never quite what they should have been according to theory, but the effect was not significant in practice.

Finally, significant reductions in the complexity of the logical formulae generated during the processing were achieved by building in automatic theorem-proving techniques to the mechanisms that generated the formulae in the first place. There is a trade-off between expending time to reduce complexity and gaining time through the reduced complexity, but there were huge gains made by the simplest reduction techniques, based essentially on automatic deduction in the symbolic logic in order to remove extraneous terms from the formulae. If the automatic deduction failed to obtain an improvement, abstract interpretation and finally mixed integer linear programming were used first to see if a reduction in complexity could possibly be achieved and then to check definitively [10].

4.4 Allocation and management strategy

Allocating work to volunteer clients required an allocation strategy. Naively sending out the next work task in alphabetical order would have eventually gotten all working clients stuck executing very hard tasks with no appreciable progress being made. The group of volunteer clients makes more progress overall if they complete the easy work tasks first. But which are the easy tasks? There is no definite way to tell other than by trying and seeing.

The size of analysis task taken on by volunteer clients was initially set to ‘one complete functional unit', i.e. a top-level function definition. Each functional unit was initially assumed to be equally as hard to analyse as every other. Each volunteer was initially given $T_0 = 10$ minutes of CPU time (normalised to a 1GHz CPU) in which to complete the work task. If the limit was exceeded, the client abandoned the task, reported back the incompletion statistic to the cloud's database, and moved on to a different work task. The task's estimate of intrinsic difficulty was raised, as reflected by an increased timeout value $T_1 > T_0$ now associated with it.

It was intended by this means to tamp down as much as possible on the total concurrent interactions with the cloud's database. Network bandwidth is a finite bound that cannot be exceeded, and the database has a limit on the number of transactions per second it can absorb. The cache at each client served to prevent 90% of that client's database transactions from escaping onto the net but work task startup and shutdown are points where large amounts of novel information are exchanged with the cloud. Giving volunteer clients by default a relatively large work unit to chew on reduces the number of data requests flying about the network and thus in principle helps the computation overall. The downside is that clients may be given more than they can deal with, plugging progress overall. Unplugging by imposing a timeout was the simplest cure for that ill. Its downside was the loss of the data that may have been accumulated at that point of abandonment.

Every time a work task was abandoned uncompleted, the estimated time required to complete it was increased by 50% ($T_{n+1} = 1.5\,T_n$), so that the next client to take it on would spend longer on it before abandoning. Moreover, tasks with a higher timeout were handed out with lower frequency (i.e., with lower priority) so that clients would tend to take the easier tasks first.

In the end, the 'hard' work tasks that took longer than the initial 10 minutes turned out to comprise only 0.5% (three hundred-odd) of the total. One might argue that the tasks handed out initially were not difficult enough since 99.5% failed to prevent their host from emitting significant noise on the network for less than ten minutes at a time.

However, of the hard tasks, two thirds eventually did complete in an hour or less, availing of lengthened timeouts. Abandonment wastes the earlier effort put in, but the time taken overall is still dominated by the successful final stint, so at most three hours of computation time were spent for each hour-long completion here.

Those 'very hard' work tasks that still were taking longer than an hour without completion (a hundred or so, or 0.15% of the original total) were dealt with in accord with the theory developed in Section 2. That is to say, the incremental progress in the client dealing with them was checkpointed to the cloud's database every minute. That pushed up the number of remote database transactions, but in return for guaranteed progress. Any volunteer client could take up the work where another had left off.

That eventually successfully dealt with all but 0.03% of the original set of seventy-two thousand functional units submitted for analysis. The remaining twenty or so 'ultra hard' exemplars remained intractable. A few of these contained constructs particular to GNU C that could not be handled by the parser, 'interior' (local) function definitions within other function definitions being the most significant such. The rest were notably characterised by the presence of generated symbolic logical assertions of great complexity, containing more than 40,000 terms each. Clearly, making progress on those requires better techniques with which to reduce the complexity of the symbolic logic expressions encountered or else better techniques for reducing the granularity of the calculations still further.

4.5 Statistics

It would take around 500 1GHz volunteer clients in the cloud in order to complete the work undertaken in the experiment in under six hours.

A rough average time needed overall for processing per top-level functional unit in the source code was 116 seconds on a notional 1GHz CPU. The 'very hard tasks' (taking more than an hour), though they accounted for not much more than 0.15% of the numbers, required around 8% of the processing time.

Regarding the CPU load expressed on a volunteer, one may conclude that it is only significant in the case of the hard work tasks, which comprise approximately 0.5% of the total number. For these cases, CPU load could in the future be limited by throttling the software automatically. It was not limited during the experiment undertaken, and CPU load was rarely more than a few percent for 99.5% of the tasks undertaken, rising towards maximum levels only on the hard tasks. The implication is that the clients were generally I/O bound, or CPU load would have routinely been much higher. The relatively infrequent queries to the cloud database that penetrated the local caches apparently stalled the client software significantly. The clients could be observed averaging between 150-500 accesses per second to the local cache layer on a 1GHz system (about 90% reads, 10% writes), leaving about 10 transactions per second per client to wend their way out to the rest of the cloud and back.

The back-end database fanout is presently limited to about 10 clients per server in the cloud, though that figure could be improved with better client-side caches. A fanout of around 40 would appear to be feasible by doubling the number of cores per server and increasing server RAM to 64GB (our experiment used a single core 1.8GHz Athlon with 3GB RAM), since a large proportion of real server load appears to come about through paging data to disk and back in order to accommodate database images that exceeded the available RAM. So between 12-50 servers in the cloud are needed to support the 500 volunteer clients projected as necessary to analyse a million lines of source code in 6 hours.

How does the cloud computation compare to the original monolithic computation from which it is descended? The short answer is 'about 50 times as slow', at present. But the original computation threw away all its intermediate calculations as it produced answers, meaning that accountability meant repeating the whole computation from scratch. It was not a scalable solution.

5 Summary

The computation of a certificate guaranteeing the absence of formally defined defects in an open source code base has been formally described.

It has been shown that the computation may be handled incrementally by a distributed 'volunteer cloud' of client CPUs each taking a fragment of the work upon themselves at a time. An experiment in which the cloud was organised to analyse about a million lines of C code (requiring about nine million seconds of standardised 1GHz CPU time) has validated the idea.

Bibliography

[1] F. Abujarad, B. Bonakdarpour and S. Kulkarni, "Parallelizing Deadlock Resolution in Symbolic Synthesis of Distributed Programs". In Proc. PDMC 2009: 8th International Workshop on Parallel and Distributed Methods in Verification, November 4, 2009, Eindhoven, The Netherlands (with Formal Methods 2009, November 2 - November 6, 2009).

[2] D. P. Anderson, "BOINC: A System for Public-Resource Computing and Storage", in *Proc. 5th IEEE/ACM International Workshop on Grid Computing*, November 8, 2004, Pittsburgh, USA

[3] D. P. Anderson, Eric Korpela and Rom Walton "High-Performance Task Distribution for Volunteer Computing", in *Proc. First IEEE International Conference on e-Science and Grid Technologies*, 5-8 December 2005, Melbourne, Australia

[4] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL '02: Proceedings of the ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, 2002.

[5] P. K. Bobko, "Open-Source Software and The Demise Of Copyright", 1 Rutgers Computer & Technology Law Journal 51 (2001)

[6] M. Barnett, K. Rustan, M. Leino, W. Schulte, "The Spec# programming system: An overview", CASSIS 2004, LNCS no. 3362, Springer, 2004.

[7] P. T. Breuer and M. García Valls, Static Deadlock Detection in the Linux Kernel, pages 52-64 in *Reliable Software Technologies - Ada-Europe 2004, 9th Ada-Europe Intl. Conf. on Reliable Software Technologies, Palma de Mallorca, Spain, June 14-18, 2004*, Eds. Albert Llamosí and Alfred Strohmeier, ISBN 3-540-22011-9, Springer LNCS 3063, 2004.

[8] P T. Breuer and S. Pickin, One Million (LOC) and Counting: Static Analysis for Errors and Vulnerabilities in the Linux Kernel Source Code, pages 56–70. in *Proc. Reliable Software Technologies – Ada-Europe 2006, 11th Ada-Europe Intl. Conf. on Reliable Software Technologies*, June 2006. Oporto, Portugal, Eds. Luis Miguel Pinho and Michael González Harbour, Springer LNCS 4006.

[9] P. T. Breuer and S. Pickin, "Symbolic Approximation: An Approach to Verification in the Large" *Innovations in Systems and Software Engineering*, Oct. 2006, Springer, London.

[10] P. T. Breuer and S. Pickin, "A formal nethod (a networked formal method)" *Innovations in Systems and Software Engineering* Dec. 2009, Springer, London.

[11] "American National Standard for Information Systems – Programming Language C, ANSI X3.159-1989", American National Standards Institute, 1989.

[12] "ISO/IEC 9899-1999, Programming Languages – C", International Standards Organisation, 1999.

[13] E. Clarke, E. Emerson and A. Sistla, “Automatic Verification of FInite-State Concurrent Systems using Temporal Logic Specifications,” ACM Trans. on Programming Languages and Systems, 8(2):244-253, 1986.

[14] P. Cousot and R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symp. on the Principles of Programming Languages*, pages 238–252, 1977.

[15] K. Douglas, “PostgreSQL”, Sams Publishing (2nd ed.), 2005.

[16] D. Engler, B. Chelf, A. Chou and S. Hallem, “Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions”, in *Proc. 4th Symp. on Operating System Design and Implementation (OSDI 2000)*, Oct. 2000, pp. 1–16.

[17] D. Evans and D. Larochelle, “Improving Security using Extensible Lightweight Static Analysis”, IEEE Software, Jan/Feb 2002.

[18] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’02)*, pages 1-12. Berlin, Germany. June 2002.

[19] R. W. Gomulkiewicz, “How Copyleft Uses License Rights To Succeed In the Open Source Software Revolution and The Implications For Article 2B”, 36 Houston Law Review 179 (1999)

[20] J.V. Guttag and J.J. Horning, “Introduction to LCL, A Larch/C Interface Language” http://ftp.digital.com/pub/Compaq/SRC/research-reports/abstracts/src-rr-074.html.

[21] A. Griffith, “GCC: The Complete Reference”, McGrawHill/Osborne, 2002

[22] G. J. Holzmann, “The SPIN MODEL CHECKER: Primer and Reference Manual”, Addison-Wesley, September 2003

[23] G. J. Holzmann, Rajeev Joshi1 and Alex Groce, “Swarm Verification”. in *Proc. ASE 2008, 23rd IEEE/ACM International Conference on Automated Software Engineering*, l’Aquila, Italy, September 2008.

[24] G. J. Holzmann, Rajeev Joshi1 and Alex Groce, “Model driven code checking”, Automated Software Engineering, Vol. 15, No. 3-4, December 2008.

[25] G. C. Necula and P. Lee, “Safe kernel extensions without run-time checking”, SIGOPS Operating Systems Review 30, pp. 229-243, SI, October 1996.

[26] Rob Johnson and David Wagner. Finding User/Kernel Pointer Bugs With Type Inference. In *Proc. 13th USENIX Security Symp., 2004* Aug 9-13, 2004, San Diego, CA, USA.

[27] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken, A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proc. Network and Distributed System Security (NDSS) Symp. 2000*, Feb 2-4 2000, San Diego, CA, USA.

Damages and Benefits of Certification: A perspective from an Independent Assessment Body

Mario Fusani

Systems and Software Evaluation Centre, ISTI-CNR, Pisa

Abstract: The need of confidence that a product or service actually possesses declared behavioural / structural characteristics is the main reason for certification. It seems that this very concept and its implications have not been discussed deeply enough in literature: mostly it was insisted in describing tests and measures [Voa00] [Tri02] [MTT09], and little attention was paid to the 'confidence' aspect itself. This is probably one reason why certification was often mistaken by users as a guarantee, and by producers as particularly severe verifications and validations. If we consider the software, then the confidence is generally weakened, and it can be observed that responsibility disclaimers associated to products are much more frequent than certificates. Even the term 'certification' was somewhat banished in the US for years, for the unspoken threat that customers unions could claim refunds for unsatisfying 'certified' services.

Yet really achievable benefits can be expected from the appropriate introduction of certification also for software-intensive products and services. The purpose of this talk is to point out some of the relevant factors that impact in the certification outcomes. These emerged from a long activity of third-party assessments, carried out at at the Systems and Software Evaluation Centre (SSEC) of Pisa.

The SSEC, during its 25-year activity, has been trying to make consistent, if not always with success, the diverging interests and views of its service counterparts (Public Administration, Manufacturing Industry, Systems and Software Suppliers) about certification and certificates. This effort has led to investigate not only the features of the certification process itself [FFL06], but also the nature of the references (Standards, System Requirements [Fus09]) and the targets of certification (properties of objects that typically are products and product life-cycles).

The talk first re-visits, with the purpose of clarification, the concept of certification, depicting a framework in which the role of the actors involved (producers, customers, end-users, certification bodies, accreditation bodies, standardisation organisms) are described with their mutual relationships.

Then some target types of entities (whose properties are the actual object of certification) are compared to the main factors that impact in the certification goals, and the risks of non achieving such goals are assessed and discussed.

The following types of target entities have been chosen for this talk, as they provide an interesting comparison among the certification factors:

- Closed Source Software, either stand-alone or embedded in systems.

- Open Source Software, also stand-alone or embedded in systems.

The considered factors are Reference Requirement domains for the certification objects (usually expressed in defined Standards) and the Certification Process.

The Reference Requirement domains can be summarised in three rather wide domains:

- Requirements concerning functional and performance behaviour (examples are communication protocols), whose conformance verification is basically testing-oriented.
- Requirements concerning lifec-ycle processes and practices (examples are Safety-related Standards), whose conformance verification is process assessment.
- Requirements concerning intrinsic product properties, such as qualities, structural characteristics and constraints in execution paths, whose conformance verification uses measuring techniques, tools and possibly formal methods. These references may appear, at a lower level of abstraction, in life-cycle based Standards.

The other factor, the Certification Process, is described in some detail, together with its actors, technical resources and various execution options. We may notice for now that this process should refer itself to Standards as well (for example, to some ISO/IEC 17000), as repeatability and reproducibility increase confidence in the results of certification.

Finally, each type of the target entities is considered as if it were submitted to certification against the various Requirement domains and with different possible process options. For the most interesting combinations of these factors, risk identification and analysis are performed. We just recall that risk means the likelihood of not reaching the certification goals (that is, transferring the right degree of confidence to the certification users). The analysis also includes the case where the formal act of certification can cause problems or damages to some stakeholders.

One relevant, yet unsurprising, conclusion is that, given the unchangeable certification goals and given the unchangeable nature of OSS, traditional References and Certification Process must change for this target type. In particular, the familiar concepts of 'product identification' and 'designer-programmer' need to evolve. It is possible that the latter concept, being the 'programmer' now represented by a motivated community of users, can conceal the means to recover that confidence we feel we are loosing for the non-applicability of many of the traditional References. The solution proposed in [Voa00], for instance, may be suitable for this scenario. Also, we should not exclude the possibility that, once it is recognised that certification brings some added-value, the OSS development process adapts itself to new circumstances (defined, for example, by some OSS life-cycle oriented Standards), a conjecture that can take from the analysis results. Various authors seem to welcome such an evolution, e. g., [OMK08]. It seems however that the OSS communities are gaining too much momentum for accepting any change request from the ’outside’.

In conclusion, the risk analysis outcomes presented in this talk provide no final solution, but highlight a set of pros and cons in the adoption of existing certification paradigms (either in Reference domains and in Certification Process) and bring forward the claim for new ones.

Keywords: Certification, Standardisation, Open Source Software

Bibliography

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

[Fus09] M. Fusani. Examining Software Engineering Requirements in Safety-Related Standards. In *Proc. of DeSSerT (Dependable Systems, Services and Technologies)*. April 22-25, 2009.

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

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

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

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

Using Free/Libre Open Source Software Projects as E-learning Tools

Antonio Cerone[1*] **and Sulayman K. Sowe**[2]

[1] antonio@iist.unu.edu
UNU-IIST, Macau SAR China.

[2] sowe@merit.unu.edu,
UNU-MERIT, Maastricht, The Netherlands.

Abstract: Free/Libre Open Source Software (FLOSS) projects can be considered as learning environments in which heterogeneous communities get together to exchange knowledge through discussion and put it into practice through actual contributions to software development, revision and testing. This has encouraged tertiary educators to attempt the inclusion of participation to FLOSS projects as part of the requirements of Software Engineering courses, and pilot studies have been conducted to test the effectiveness of such an attempt. This paper discusses two pilot studies with reference to several studies concerning the role of learning in FLOSS projects and shows how using FLOSS projects as E-learning tools has a potential to increase the quality of the software product.

Keywords: OSS development; Education; Pilot Studies; Knowledge Exchange; E-learning; Software Quality

1 Introduction

Over the last years Free/Libre Open Source Software (FLOSS) communities have proven themselves to be able to deliver high-quality system and application software. Although FLOSS communities consist of heterogeneous groups of independent volunteers, who interact but are driven by different interests and motivations, and may appear to an external observer chaotic or even anarchic, they actually have specific organisational characteristic [Muf06]. These characteristics have been identified and analysed through empirical studies, which highlighted the implications of the FLOSS phenomenon throughout the information, knowledge, and culture economy, in a multidisciplinary context that goes well beyond software development [Muf06, Ben02]. Benkler [Ben02] goes even further and suggests reasons to think that peer-production may outperform market-based production in some information production activities in which a pervasively networked environment plays a major facilitating role. The generality of Benkler hypothesis makes it suitable to be applied to an educational context [fut06]

Education has been showing during the last years multifaceted signs of crisis which affect all levels from primary to tertiary: diminishing academic achievements, increasing number of dropouts, teacher shortages and collapse of education reforms. A workshop held at Bagnols,

[*] *Correspondence Author*: Antonio Cerone. Email:{antonio@iist.unu.edu}. *Address*: UNU-IIST, Macau SAR China. Tel: +853 2871-2930, Fax: +853 2871-2940

France, attended by educational practitioners, technologists, brain scientists and cognitive psychologists has identified factors in the current crisis in education and examined the potential uses of innovative technologies to support education [TS03]. Two important conclusions of the Bagnols workshop are that *education must be learner-centred* and that *learning must be social and fun* [TS03]. Learners are no longer comfortable with traditional modes of education, in which information is presented linearly, mostly in a text-based way, with almost no activities aiming to put acquired knowledge into real-life practise. This has created a mismatch between modes of education adopted by schools and universities and modern living style. In fact, nowadays information is presented to the public in daily life throughout multiple streams and multiple modalities simultaneously. The Internet provides a richer, much more frequently updated and more appealing source of information than printed newspapers, magazines and book. Moreover, information on the Internet is multi-modal and is organised in a tree-like or even graph-like structure rather than linearly. This allows learners to quickly *navigate* towards the targeted information in a way that appears to them more similar to pure entertainment than to academic work. Social relationships have been also heavily affected by the Internet: social networks, such as Facebook, allow individuals geographically distributed and with different cultural backgrounds to become friends, participate in online activities and games, join discussion fora and even establish romantic relationships.

FLOSS communities seem to have many characteristics that match the way information is best received by nowadays learners. They provide that sort of virtual world in which we often carry out our social and free-time activities. Commons-based peer-production [Ben02, Ben07] is the model of economic production in which the creative energy of large numbers of individuals is remotely coordinated, usually through the Internet, into large, meaningful projects mostly without traditional hierarchical organisation. Individuals participate in peer-production communities not just because of extrinsic motivations, such as solve problems, improve technical knowledge base, increase reputation and peer recognition and pass examinations, but also, and probably mainly, for a wide range of intrinsic reasons: they feel passionate about their particular area of expertise and enjoy self-satisfaction from sharing their knowledge and skills; they revel in creating something new or better; they have a personal sense of accomplishment and contribution and a sense of belonging to a community [Muf06, TW06, CS08].

FLOSS communities are therefore an ideal platform to implement learner-centred education in which learning is social and fun, as envisaged by the Bagnols workshop. FLOSS communities are a natural instantiation of this model, which has recently been taken as the basis on which to build new approaches to education [fut06]. Although this can be potentially applied to any level [fut06] and field of education [MGS09], this paper focuses on Software Engineering (SE) undergraduate and postgraduate courses [Kho09, SSD06, JØ07]. Application of FLOSS learning approaches to Software Engineering education is also a way of implementing the suggestion of the joint IEEE/ACM CS undergraduate curriculum guidelines [IEE04] that CS curricula should have significant real-world basis necessary to enable effective learning of software engineering skills and concepts.

All previous work in analysing learning aspects of FLOSS communities emphasises the benefits that the exploitation of such aspects may have on the educational process. In our work we also aim to identify the benefits that the explicit linkage of a FLOSS project to a formal education programme, such as a Software Engineering course or postgraduate research activity, brings to the FLOSS community itself and, in the end, to the quality of the FLOSS product.

In Section 2 we consider recent work that explores the link between FLOSS approaches and education [fut06]. FLOSS communities are analysed as *collaborative networks* and *communities of practice* to extrapolate the learning process that facilitates the emergence and evolution of community member's knowledge [MGS09]. Challenges in adapting and transferring such a learning process to an educational setting are discussed.

Section 3 considers two research frameworks and corresponding pilot studies conducted to empirically analyse the use of FLOSS communities for formal education in Software Engineering at undergraduate [Sow08, Sta09, SGG, SSL06] and postgraduate [JØ07] levels. The two approaches are discussed with respect to the student's degree of freedom (Section 3.3) and topical focus (Section 3.4). The proposal of a third pilot study [CS08] more ambitiously aims to operate changes into the structure and organisation of the FLOSS community to facilitate the use of innovative methodologies, such as formal methods, in which to involve students.

In Section 4 we show, with respect to Shaikh and Cerone's framework for evaluating quality of Open Source Software (OSS) [SC09], that the usage of FLOSS projects as e-Learning tools has the potential to increase the quality of the FLOSS product.

2 The Role of Learning in FLOSS Communities

One important attempt to identify a general link between FLOSS approaches and educational agendas is a 2006 Futurlab report [fut06] that looks at FLOSS as a cultural phenomenon and aims to extrapolate new approaches to teaching and learning and to define new models of innovation and software development in education. Drawing on Benkler's work on commons-based peer-production [Ben02] the report discusses strengths and weaknesses of FLOSS approaches which might apply to educational settings. Then it focuses on two ways in which peer-production FLOSS-like approaches may be used in teaching and learning:

collaborative network that is network that consists of a variety of entities that are largely autonomous, geographically distributed and heterogeneous in terms of their operating environment, culture, social capital and goals, but nevertheless collaborate to better achieve common or compatible goals and whose interactions are supported by computer network [CA06];

community of practice that is a group of people who share an interest, a craft, and/or a profession and can evolve naturally because of the members' common interest in a particular domain or area, or it can be created specifically with the goal of gaining knowledge related to their field [LW91].

Distributed *collaborative networks* provide a powerful platform in which, due to the mediation of digital technology in a virtual environment, the duality teacher-learner fades out, and the two roles of teacher and learner merge together into the generic role of actor within the participatory culture of the network and its informal learning spaces [fut06, MGS09]. From the learner's perspective, this enables the full range of potential intrinsic reasons mentioned in Section 1 to become actual motivations and to urge learners to play, alongside with teachers, their common role as actors in the community's activities. In addition, FLOSS communities are characterised by the freedom with which actors choose projects as well as the total control that actors have on the degree of their own contribution to the project.

Freedom and equality of participants constitute a "democratic" basis for analysing FLOSS communities as *communities of practice*. Novices are always welcome by FLOSS communities, in which they undergo through a gradual process of social integration and skill development that allows them to earn a reputation as reliable developers and then move towards the leading positions in the community [Tuo05]. FLOSS communities are in this sense open participatory ecosystems [MGS08, MGS09], in which actors create not only source code but a large variety of resources that include the implicit and explicit definitions of learning processes and the establishment and maintenance of communication and support systems. Furthermore these resources are made visible and available to other actors. Therefore development (source code), support (tools) and learning (knowledge) emerge as the product of a continuous socialisation process in a virtual environment. Development of source code is enabled by building up knowledge about already produced code, through direct observation, review, modification as well as discussion with other actors, and about support tools, through direct interaction as well as access to documentation and discussion with other actors. As suggested by Sowe and Stamelos [SS08a] the learning process of individual actors can be divided in four phases through which knowledge evolves. We give our slightly different characterisation of such phases as follows:

socialise by *implicitly sharing knowledge*;

externalise *tacit knowledge* by making it explicit to the community;

combine community *explicit knowledge* and organise it as abstract knowledge;

internalise *abstract knowledge* by absorbing it and combining it with own knowledge and experiences to produce new tacit knowledge.

The four phases are not fully sequential but overlap in a certain measure. In particular, socialisation, after playing the role to initiate the learning process, is still active during the other phases for which is actually the enabling factor.

If we want to transfer the learning process occurring within FLOSS communities to an educational setting, we need to better understand the cognitive aspects of the four phases above and interpret and implement them in a context driven by educational goals rather than just by software development.

Socialisation does not require an education-oriented interpretation and is probably the easiest phase to implement in an educational setting. In fact, socialising in a virtual environment, specifically through the Internet, already permeates our daily life and specific mechanisms and tools used by FLOSS communities, such as discussion fora, are general enough to be used for educational purposes; moreover, there are already several specific, and even more sophisticated (i.e. supporting multi-modal interation) e-learning tools and environments [Imm] that implement socialisation, such as Moodle [Moo] and Second Life [Sec].

Externalisation naturally occurs in an implicit way through socialisation tools such as discussion fora, but needs to be addressed by knowledge-management tools, such as repositories, to be effectively implemented in an explicit way. Tools used to manage and organise knowledge within FLOSS communities are often a challenge for the novice and actually require the user to go through a learning process before using them. Although this may be acceptable in a context purely driven by software development, in which skill in quickly acquiring familiarity with new tools may be considered a reasonable pre-requisite to enter the community, and may be even seen as a parameter to naturally select skilled contributors, the situation is totally different in an education-driven context. In such a context going through an heavy learning process to be able to use learning tools is definitely unacceptable. Therefore, existing tools have to be made more usable while more appropriate tools have to be developed to effectively implement externalisation in an educational setting. Externalisation is also intimately related to the intrinsic motivations of the user in joining the community and contributing to it. Intrinsic motivations, such as

- feel passionate about particular area of expertise,
- enjoy self-satisfaction from sharing knowledge and skills,
- have a sense of belonging to a community,

are all strong drivers for externalisation. There are also a number of extrinsic motivations that contribute to externalisation, which include

- solve particular technical problems/needs by exploiting Linus' Law: "given enough eyeballs, all the bugs are shallow" (from Linus Torvalds);
- public visibility to increase reputation and and peer recognition.

Combination of knowledge is incremental and consists of two main activities:

- multiple interactions with knowledge-management tools as well as with other members of the community to identify and extract relevant bits of explicit knowledge;
- combination and organisation of such bits of explicit knowledge to produce meaningful abstract knowledge.

The interaction with knowledge-management tools presents the same challenges as discussed for the externalisation phase. Organisation of explicit knowledge and production of meaningful abstract knowledge are cognitive activities within the ambit of *knowledge representation*. Several alternative theories have been proposed in cognitive psychology to explain knowledge representation within the human mind, but it is beyond the scope of this paper to deals with such theoretical aspects. From a pragmatic point of view we can say that the way individuals combine explicit knowledge is affected by the accessibility, structure and presentation of the contents of such knowledge and by own personal learning attitudes. Knowledge-management tools have therefore to address this issues as well as to enable individuals to have more control and responsibility for their learning [GFR+05].

Internalisation of knowledge is a cognitive activity which is driven by both intrinsic and extrinsic motivations. *Intrinsic motivations* for internalisation are:

- revel in creating something new or better;
- have a personal sense of accomplishment and contribution.

Extrinsic motivations for internalisation are:

- improve technical knowledge base;
- pass examinations;
- develop the solution to a technical problem.

We will discuss in Section 3.4 how the grading approach utilised by the lecturer affects these extrinsic motivations and may create conflicts with intrinsic motivations, thus leading to a partial inhibition of internalisation. Internalisation is also facilitated by the efficacy and usability of code analysis tools such as bug trackers.

We will also discuss in Section 3.3 that limiting the degrees of freedom of students in participating in FLOSS projects may produce community members with little extrinsic motivations, with negative consequences for both the externalisation and internalisation phases of their learning process.

3 Frameworks and Pilot Studies in SE Education

The joint IEEE/ACM CS undergraduate curriculum guidelines [IEE04] suggest that CS curricula should have significant real-world basis necessary to enable effective learning of software engineering skills and concepts and should incorporate Capstone projects. Although many efforts have been made to involve students in software projects in local companies, most companies are not willing to sacrify their products to students who are constrained to complete the assigned work in one semester [Alz05]. Therfore the *bazaar of learning* offered by FLOSS projects represents a meaningful alternative learning context to expose students to real-world software development activities [SSD06].

Characteristics and evolution modalities of FLOSS communities have been largely studied empirically by extracting data from repositories and performing statistical analysis on such data [SII07]. However, learning aspects cannot be easily captured using this research methodology due to the absence of related information inside repositories. In this section, we focus on Software Engineering education and survey studies aimed to explore the use of FLOSS projects as e-learning tools.

During the last decade the FLOSS development model has deeply changed the way we develop and commercialise software, affecting traditional software development methodologies and posing serious challenges to commercial software industry. Students are strongly attracted by this new software development paradigm and enthusiastically join FLOSS projects. At the same time software industry is more and more including OSS skills and knowledge among their hiring selection criteria [Lon08]. This new scenario, in addition to the fact that FLOSS projects are actually Software Engineering practice, has made Software Engineering the most appropriate teaching subject to test the educational capabilities of FLOSS projects and has encouraged tertiary educators to attempt the inclusion of participation to FLOSS projects as part of the requirements of Software Engineering courses. Several pilot studies have been conducted to test the effectiveness of such an attempt and to assess the feasibility of full-scale studies.

3.1 Undergraduate Education Pilot Study

A pilot study conducted by Sowe and Stamelos [SS08b] addressed the open question as to whether the FLOSS methodology can be used to teach Software Engineering courses within a formally structured curriculum. The study was based on a pilot programme to teach software testing [SSD06] and aimed to develop and test a research method [SS08b] and to develop an approach to evaluate student participation [SSL06]. Within a pool of 150 undergraduate students enrolled in a course of "Introduction to Software Engineering" at Aristotle University, Greece, 15 joined the programme and 13 of them completed it. The study consisted of three phases in which students:

1. received lectures on FLOSS-related topics, browsed projects and selected one of them;
2. participated in the selected project with the aim to find and report bugs, and possibly fix them;
3. were evaluated and graded by the lecturers.

The study made use of two surveys in which students showed their interest in continuing their participation in the project after graduating. Student were actually forwarding responses from their projects to the lecturers after the pilot programme was ended and student grades published. This is a clear evidence that FLOSS projects can involve students in a long-term participation, which is in line with the need for long-life learning experiences, typical of a discipline in exponentially rapid evolution as is Software Engineering.

3.2 Postgraduate Education Pilot Study

Jaccheri and Østerlie [JØ07] use an approach for teaching master level students in which students are given assignments for which they have to

- survey literature on OSS development and formulate one or more research question(s) that could be addressed by participating in a project;
- select an OSS project which is appropriate for the assignment and the formulated research questions;
- act as developers in the selected project;
- act as researchers in the selected project by addressing the formulated research questions.

This approach has been used since 2002 by the Software Engineering Group (SU) [Con] of the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). Jaccheri and Østerlie report on a concrete study based on this approach, in which one master student was requested to participate in a commercially controlled OSS project, the Netbean open source project, to understand how firms can benefit from using OSS [JØ07]. More specifically, the student was asked to determine how the use of Software Engineering techniques, such as explicit planning, ownership, inspection and testing, affects the OSS project. Within the scope of the assignment, the only constraint was to use action research

(AR) [DMK04] as the methodology for the study. This study rises important considerations about the degree of freedom given to the student. While students appreciate freedom in assignments as a positive learning experience, the authors recognise, as a result of the evaluation of their work by industrial professionals as well as discussions with other researchers, that it would have been more effective from a research perspective to provide students with predefined research questions. In the particular case study that they reported research questions were about the interaction between professionals and volunteers; this required the selection of a project in which commercial actors actively play significant roles. In an alternative research framework the project could have been selected before formulating the research questions. In an even more constraining framework the selection of the project could even been made by the lecturer.

3.3 Student's Degree of Freedom

The degree of freedom given to students is an important issue in both studies. In the first study [SS08b] undergraduate students joined the programme on a volunteer base and had full freedom in selecting the project; given that the course was specifically about software testing, the assignment generically asked to find and report bugs, and possibly fix them. In the second study [JØ07] postgraduate students had full freedom in formulating research questions, but were constrained by their own choice of research questions in selecting the project.

One of the main reasons for the success of FLOSS projects is to be based on communities of volunteers who are totally free in choosing the way of contributing both in terms of tasks and time commitment. Intrinsic reasons are fundamental in motivating active and effective participation in a FLOSS project. Forcing the injection of actors who partly or entirely lack intrinsic motivations but are requested to play an active role in the community would not produce effective learning in those actors and may even be detrimental to the whole FLOSS project community. We have seen in Section 2 that the phases of learning that are heavily dependent on intrinsic motivations are externalisation and internalisation. These two phases include important cognitive activities and their incomplete actuation, as in case of lack of intrinsic motivations, severely inhibits the whole learning process.

It is therefore essential to preserve the volunteer-based approach while using participation in FLOSS projects for educational purposes, as it was done in the pilot study conducted at Aristotle University. In general, for undergraduate courses, we would suggest not to include participation in a FLOSS project as a course requirement unless the course is a very focussed elective. For postgraduate students, participation in a FLOSS project may be either related to a course or to a final thesis or project work. In general, it is expected that postgraduate students have more focussed interests and a higher degree of maturity than undergraduate students. In this perspective, a postgraduate student who has chosen an elective course or a thesis topic which requires participation in a FLOSS project is supposed to have sufficient intrinsic motivations to succeed in the task.

The issue of the project selection is a very subtle one. In both pilot studies described above the project selection is left to the student, although some general selection criteria are provided. However, in the pilot study conducted at the Norwegian University of Science and Technology the selection of the project strongly depends on the research questions previously formulated by the student. The fact that the student has chosen a specific research question does not exclude that such research question may rule out all projects in which the student is likely to be enthusiastically interested. In this sense a research framework in which the project is selected before formulating the research questions is more sensible. In general, in designing the study research framework it is essential to ensure that there are no requirements for the student that may explicitly or implicitly reduce the student's degree of freedom in choosing the project.

3.4 Student's Topical Focus

In the two pilot studies described above the student's topical focus in participating in the project was dictated by the assignment. In the pilot study conducted at Aristotle University the student had to find and report bugs, and possibly fix them. The grading system included marks for email exchange with the lecturer concerning the project, proper use of bug tracking system or bug database and testing activity measured by the number of bugs found, reported and fixed, and by the number of replies to the reports. This restricted focus has probably worked as an extrinsic motivation that prevented students from contributing to the project in terms of software development, for which there was no mark. As a result students probably felt that the effort needed to fix bug was not sufficiently rewarded in terms of marks. This hypothesis is confirmed by one outcome of the study: although students performed well in finding and reporting bugs, they did not well in fixing bugs [SSL06].

It is inevitable that the grading and evaluation approach strongly affects student's extrinsic motivations: the more transparent and explicit the grading approach the stronger the effect on extrinsic motivations. A grading approach that has a strong effect on extrinsic motivations does not allow students to achieve a complete involvement in the project and often causes a conflict with intrinsic motivations, which are an essential driver in FLOSS project. Such a conflict may result in an incomplete actuation of the internalisation phase of the learning process, which depends on both intrinsic and extrinsic motivations, and inhibit potential learning capabilities of the student.

However, avoiding such a strong effect is not easy in a formal education context. Quantifying the evaluation of the participation to the project as a whole with no further details would not be a feasible solution. Such a solution would be clearly against usual university policies that require lecturers to make the grading approach public by quantifying each contribution in terms of percentage of the final grade. Moreover, limiting the information about the assessment procedure that is provided to students would be inherently unfair and might promote suspicion among students. And in the end, this would actually reduce extrinsic motivations of students. Possible solutions to the problem could be that lecturers

- evaluate the participation in the project indirectly by assessing a written report and publish details of the grading of such report;
- discuss beforehand the grading approach with the students and agree on the details with them;
- provide alternative assessment and/or grading approaches among which the students may choose;
- develop an appropriate peer assessment approach.

These proposed solutions are neither exhaustive nor mutually exclusive.

In the pilot study conducted at the Norwegian University of Science and Technology, which involved postgraduate students, the student's task was not only to actively participate in the project, but also to use such participation to address research questions previously formulated. This is an interesting attempt to involve learners in studying and possibly improving the FLOSS development process, that is, studying and possibly improving the learning tool they are using. In the Norwegian study the student's focus was on management and organisational aspects of Software Engineering.

There are other aspects of Software Engineering in which students, especially postgraduate students, may contribute, through their participation in a FLOSS project, to provide new insight in relation to the FLOSS approach. One of these aspects is *quality assurance*. The lack of central management in FLOSS projects 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 in the project [Mic05, MHP05]. Without precise indicators of this sort we cannot produce an effective quality assurance methodology for the released software. Zhao and Elbaum [ZE00] conducted a survey to examine the factors underlying quality assurance methods used within FLOSS communities and found out that their general attitude and practices towards quality and realising quality assurance practices are somewhat different to those prevalent in traditional software development. This situation opens a lot of research questions which could be addressed in studies conducted by postgraduate students through their involvement in FLOSS projects. In Section 4 we will further discuss the impact that such involvement could have on the quality of FLOSS products.

Postgraduate students are often exposed during their study to innovative software design and analysis technologies that enjoy little appreciation outside the academic world, either because such technologies are not mature enough to be applied to practical projects or because, in an industrial perspective, their cost prevail on the actual benefit they bring. Formal methods are one of such innovative technologies. Postgraduate students could bring new insights in FLOSS communities through the application of new specification and verification technologies such as formal modelling, model-checking and theorem-proving. This would require students to reverse engineer FLOSS code into a formal model and apply formal techniques to analyse the model. Unfortunately most FLOSS developers are unlikely to be familiar with formal methods and probably view them with a similar reluctance as does the industrial world. Feeding results of formal analysis back to the FLOSS project would be therefore a big challenge for the students. Here a soft approach would be needed: outcomes of formal analysis should be mapped back to code and test cases before been presented to the community. To this purpose, formal modelling techniques that provide counterexamples when a required system property is proven not to hold, such as model-checking, are the most appropriate. Besides the soft approach, it would be important to include in bug reports some information about the formal results that led to the bug identification. In this way, students would play the role of educators in their interaction with FLOSS developers, so fostering a gradual acceptance of new technologies by the FLOSS communities.

An alternative approach to promote the use of formal methods in the FLOSS community is a pilot project proposed by Cerone and Shaikh [CS08] as an attempt to explicitly introduce formal methods in the FLOSS development process. The most difficult task in this attempt is to preserve the intrinsic freedom that characterises contributions by the volunteers who join FLOSS projects.

In fact, it would not be acceptable, and neither would it be accepted by the FLOSS community, 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 should therefore introduce and present formal methods only as a possible but not mandatory option available to the contributors. This approach would require an additional effort by the project leader team in facilitating the integration of those contributions that do not make any use of formal methods into the new development model. An important role would be played here, once again, by postgraduate students called to reverse engineer code, produced by other actors in a traditional FLOSS way, into changes and extensions to the formal model.

4 Impact on the Quality of FLOSS Products

We have seen in Section 3 that students can successfully use FLOSS projects as e-Learning tools and gain effective learning of software engineering skills and concepts from participating in FLOSS project. We have also seen that students, and in particular postgraduate students, can produce important contributions to the evolution of the FLOSS development model. In this section we investigate how such contribution can actually have impact on the quality of FLOSS products.

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

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

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

quality by design which corresponds to the traditional notion of software quality [IEE99, Pre00]: the end quality is judged by the design and implementation of the actual software and the code that underlies it.

Quality by access would greatly benefit from the use of formal methodologies by postgraduate students participating in the project. The reverse engineering of FLOSS code into formal models improves understanding the system architecture and the structure of code and leads to the production of better documentation. A by-product of the reverse engineering process is also the identification of inconsistencies and redundancies in the code and, as a consequence, its improvement with an increase in readability. Formal verification techniques produce results that are more general and understandable than the ones obtained using traditional testing techniques. Moreover, these results can be tracked back to the model, facilitating the fixing of bugs.

We have seen in Section 2 that availability and usability of knowledge-management tools is essential to enable the externalisation phase of the learning process. The development of new tools and the improvement of usability in existing tools with the aim to address the learning process in FLOSS communities is therefore likely to increase quality by access.

Quality by development is an attempt to measure the efficiency of all processes aiming to produce and review code and the interaction between them. Shaikh and Cerone [SC09] identifies five factors on which this notion of quality depends:

- precise and explicit understanding of software goals and requirements;
- choice of methodologies for testing, debugging and error and bug reporting;
- choice of programming languages and development environments;
- tools to provide effective communication, coordination and overall management of the project;
- facilitation of rapid frequency of beta releases.

We observe that the usage of FLOSS projects as e-Learning tools has the potential to affect these factors in a way that increases quality by development. First we observe that an additional effect of reverse engineering FLOSS code into formal models is the explicit definition of software requirements. Second, we believe that if methodologies, programming languages and tools are chosen having in mind not only their usage in software development but also their educational values, then there is a positive impact on the entire project community and, as a result, an additional benefit for the development process. Third, the frequent injection of students with short deadlines to complete their assignments may facilitate rapid frequency of beta releases. Finally, empirical studies such the one presented in Section 3.2 can produce a better insight in how these factors interact with each other and affect each other in the global context of the project management and organisation.

Quality by design, the traditional notation of quality, can be seen in the FLOSS context as a specific measure of

- the use of recognised software design notations, formal notations and analysis techniques to provide correctness with respect to explicitly desired safety, security and non-functional properties, and
- the production and frequent update of appropriate and explicit documentation that helps both the users and future developers.

We have seen in Section 3.3 that student participation can bring innovative software design and analysis technologies, such as formal methods, into FLOSS projects, thus increasing the community knowledge and, on the long term, increasing the acceptance of these technologies within FLOSS communities. Moreover, pilot projects aiming to explicitly incorporate these technologies in the FLOSS development process [CS08] could show whether or not there is an effective increase in quality by design. As for documentation, it is likely that student participation would increase its production, since written reports to document code production and performed analysis are a common form of assignment.

Finally, as we have anticipated in Section 3.3, postgraduate students may contribute, through research-driven participation in a FLOSS project, to identify quality indicators and define quality metrics appropriate for the FLOSS development model

5 Conclusion and Future Work

In this paper we have considered recent work that explores the link between FLOSS approaches and education and described the dynamics of the learning process that facilitates the emergence and evolution of community member's knowledge. We have then considered two pilot studies conducted to empirically analyse the use of FLOSS communities for formal education in Software Engineering, discussed choices made in designing the research frameworks for the two studies and proposed suggestions to improve the frameworks to better match the student's learning process. Finally, we have shown that the use of FLOSS projects as e-Learning tools has a potential to increase the quality of the software product.

This research has been conducted as a preliminary analysis towards the objective of building a worldwide university network, coordinated by the United Nations University (UNU), to implement the use of FLOSS projects as e-Learning tools in Software Engineering postgraduate education. A first step in our future work is to design a framework, which incorporates the recommendations we presented in Section 3, for geographically distributed pilot studies in which students

- are totally free in the choice of the FLOSS project;
- are evaluated using a grading approach that is not likely to weaken their intrinsic motivations and that possibly strengthen their extrinsic motivation;
- are requested to participate in the project they have chosen but are totally free in choosing the form of participation;
- may have various levels of time commitment, which correspond to distinct numbers of credits;
- may choose, on a volunteer basis, a focus for their participation in the project among different topical areas such as code development, review, testing, reverse engineering, formal analysis;
- may choose, on a volunteer basis, a research topic concerning the investigation of the FLOSS phenomenon, which may include learning, project management, communication, social aspects, software quality, etc.

A second step is the creation of *pilot projects* in line with Cerone and Shaikh proposal [CS08], with academics and former students who have taken part to the pilot studies of the first step, being part of the leader team.

The final objective is to build a *postgraduate e-Learning programme* in OSS approaches to Software Engineering as part of the new UNU postgraduate programmes, and utilise some of the most successful pilot projects as e-learning tools within such a programme.

Bibliography

[Alz05] Z. Alzamil. Towards an Effective Software Engineering Course Project. In *Proceedings of the 27th International Conference on Software Engineering*. Pp. 631–632. ACM Press, 2005.

[Ben02] Y. Benkler. Coase's Penguin, or, Linux and The Nature of the Firm. *The Yale Law Journal* 212:369–446, 2002.
URL: http://www.yalelawjournal.org/images/pdfs/354.pdf.

[Ben07] Y. Benkler. *The Wealth of Networks: How Social Production Transforms Markets and Freedom*. Yale University Press, 2007.

[BTD05] J. M. Barahona, C. Tebb, V. Dimitrova. Transferring Libre Software Develoment Practises to the Production of Educational Resources: the Edukalibre Project. In *Proceedings of the 1st International Conference on Open Source Systems (OSS2005)*. Genova, Italy, 11–15 July 2005.

[CA06] L. M. Camarinha-Matos, H. Afsarmanesh. Collaborative networks: a new scientific discipline. *Journal of Intelligent Manufacturing439452*, 16:439–452, 2006.

[Con] R. Conradi et al. Software Engineering Group homepage. Norwegian University of Science and Technology (NTNU). URL: http://www.idi.ntnu.no/grupper/su/.

[CS08] A. Cerone, S. A. Shaikh. Incorporating Formal Methods in the Open Source Software Development Process. In *Proceedings of the OpenCert and FLOSS-FM 2008 joint Workshop*. UNU-IIST Research Report 398. 2008.

[DMK04] R. M. Davison, M. G. Martinsons, N. Kock. Principles of Canonical Action Research. *Information Systems Journal (ISJ)* 14(2):65–86, 2004.

[fut06] Opening Education. The potential of open source approaches for education. Published online, Futurelab, 2006.
URL: http://www.futurelab.org.uk/resources/publications-reports-articles/opening-education-reports/.

[GFR$^+$05] H. Green, K. Facer, T. Rudd, P. Dillon, P. Humphreys. Personalisation and Digital Technologies. Published online, 2005.
URL: http://www.futurelab.org.uk/resources/publications-reports-articles/opening-education-reports/.

[IEE99] IEEE Std 610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology. February 1999.

[IEE04] IEEE/ACM Joint Task Force on Computing Curricula. Software Engineering 2004 Curriculum guidelines for Undergraduate Degree Programs in software Engineering. 2004. URL: http://sites.computer.org/ccse/SE2004Volume.pdf.

[Imm] Immersive Education. URL: http://immersiveeducation.org/.

[JØ07] L. Jaccheri, T. Østerlie. Open Source Software: A Source of Possibilities for Software Engineering Education and Empirical Software Engineering". In *Proceedings of the Workshop on Emerging Trends in FLOSS Research and Development, co-located at ICSE'07*. Minneapolis, US, 21 May 2007.

[Kho09] A. Khoroshilov. Open Source Certification and Educational Process. In *Proceedings of the 3rd International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2009)*. Electronic Communications of the EASST 20. 2009.

[Lon08] J. Long. Open Source Software Development Experiences on the Students' Resumes: Do They Count? - Insights from the Employers' Perspectives. *The Journal of Information Technology Education (JITE)* 8:229–242, 2008.

[LW91] J. Lave, E. Wenger. *Situated Learning: Legitimate Peripheral Participation*. Cambridge University Press, 1991.

[MGS08] A. Meiszner, R. Glott, S. K. Sowe. Free/Libre Open Source Software (FLOSS) Communities as an Example of successful Open Participatory Learning Ecosystems. *The European Journal for the Informatics Professional, UPGRADE* IX(3):62–68, 2008.

[MGS09] A. Meiszner, R. Glott, S. K. Sowe. Preparing the Ne(x)t Generation: Lessons learnt from Free/Libre Open Source Software — Why free and open are pre-conditions and not options for higher education! In *Proceedings of the 4th International Barcelona Conference on Higher Education*. Volume 2. Knowledge technologies for social transformation. Barcelona, Spain, 15–19 July 2009.

[MHP05] M. Michlmayr, F. Hunt, D. Probert. Quality Practices and Problems in Free Software Projects. In Scotto and Succi (eds.), *Proceedings of the First International Conference on Open Source Systems*. Pp. 24–28. Genova, Italy, 2005.

[Mic05] M. Michlmayr. Quality Improvement in Volunteer Free Software Projects: Exploring the Impact of Release Management. In Scotto and Succi (eds.), *Proceedings of the First International Conference on Open Source Systems*. Pp. 309–310. Genova, Italy, 2005.

[Moo] Moodle. URL: http://moodle.org/.

[Muf06] M. Muffatto. *Open Source — A Multidisciplinary Approach*. Imperial College Press, 2006.

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

[SC09] S. A. Shaikh, A. Cerone. Towards a Metric for Open Source Software Quality. In *Proceedings of the 3rd International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2009)*. Electronic Communications of the EASST 20. 2009.

[Sec] Seconf Life. URL: http//secondlife.com/.

[SGG] S. K. Sowe, R. A. Ghosh, R. Glott. A Model for Teaching and Learning in Open Source Software Projects: Constructivist Approach. Submitted for publication.

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

[Sow08] S. K. Sowe. Pilot Studies Relevant to Learning in FLOSS. In *Proceedings of the 1st International Conference on Free Knowledge, Free Technology (FKFT)*. Education for Free Society, Barcelona, Spain, 15–18 July 2008.
URL: http://www.slideshare.net/andreasmeiszner/pilot-studies-relevant-to-learning-in-floss.

[SS08a] S. K. Sowe, I. Stamelos. Reflection on Knowledge Sharing in F/OSS Projects. In *Open Source Development, Communities and Quality*. IFIP International Federation for Information Processing 275, p. 351358. 2008.

[SS08b] S. K. Sowe, I. G. Stamelos. Involving Software Engineering Students in Open Source Software Projects: Experiences from a Pilot Study. *Journal of Information Systems Education (JISE)* 18(4):425–435, 2008.

[SSD06] S. K. Sowe, I. Stamelos, I. Deligiannis. A Framework for Teaching Software Testing using F/OSS Methodology. In *Proceedings of the 2nd International Conference on Open Source Systems (OSS2006)*. Como, Italy, 8–10 June 2006.

[SSL06] S. K. Sowe, I. G. Stamelos, A. Lefteris. An Empirical Approach to Evaluate Students Participation in Open Source Software Projects. In *Proceedings of the the IADIS CELDA conference*. Barcelona, Spain, 8–10 December 2006.

[Sta09] I. Stamelos. Involving Software Engineering Students in Open Source Software Projects: Experiences from a Pilot Study. *International Journal of Open Source Software & Processes (IJOSSP)* 1(1):72–90, 2009.

[TS03] M. Tokoro, L. Steels. *The Future of Learning*. IOS Press, 2003.

[Tuo05] I. Tuomi. The future of open source: Trends and prospects. In Wynants and Cornelis (eds.), *How open is the future? Economic, social and cultural scenarios inspired by free and open source software*. Pp. 429–459. Vrjie Universiteit Press, 2005.

[TW06] D. Tapscott, A. D. Williams. *Wikinomics: How Mass Collaboration Changes Everything*. Portfolio Books, 2006.

[ZE00] L. Zhao, S. Elbaum. A survey on quality related activities in open source. *ACM SIGSOFT Software Engineering Notes* 25(3):53–57, May 2000. ACM Press New York, NY, USA.

Testing as a Certification Approach

Alberto Simões[1], Nuno Carvalho[2] and José João Almeida[3]

[1] ambs@cpan.org, http://www.eseig.ipp.pt/
Escola Superior de Estudos Industriais e de Gestão, Instituto Politécnico do Porto
[2]smash@cpan.org, [3]jj@di.uminho.pt, http://www.di.uminho.pt/
Departamento de Informática, Universidade do Minho

Abstract: For years, one of the main reasons to buy commercial software instead of adopting open-source applications was the, supposed, guarantee of quality. Unfortunately that was rarely true and, fortunately, open-source projects soon adopted some good practices in their code development that lead to better tested software and therefore higher quality products.

In this article we provide a guided tour of some of the best practices that have been implemented in the Perl community in the recent years, as the pathway to a better community-oriented repository of modules, with automatic distributed testing in different platforms and architectures, and with automatic quality measures calculation.

Keywords: test-driven development, test-coverage, distributed testing, Perl community

1 Introduction

Test-driven development [Max03] is not a new approach on the now widely discussed Extreme Programming Techniques [Bec99]. The idea is simple and effective: before writing code, or even thinking on how it will be implemented, the programmer is invited to analyze how he would like to use the application (or the function or methods being developed), and look at it, as often young scholars do, as a little black box, and decide what gets in and what should get out.

After this first discussion, some tests should be written. These tests will use the application's functions or methods being developed, invoking them with some kind of input, and checking its output against some kind of gold standard. This is also a great opportunity for developers to analyze the code API, if it should have one, because since there is not any code actually written yet, the signature of operations made available by the API can easily change.

Only after a few tests are written the developer should start the implementation. This also gives a chance for the developer to meditate about the expected behavior of the new code without any concerns about implementation details. The behavior should always be chosen outside the scope of implementation, since the expected behavior of a function, or method, should not be tailored by implementation difficulties. This is true for most of the cases, but not always. As soon as a first running code is available, it should be run against the written tests. If any test fails, it means the algorithm is not working properly and, if the test passes, it means the code is supporting the cases described in the tests.

This process iterates. During development it is natural that the developer thinks of some new situation that should be handled. Before coding that portion of code, he should write a new test that tests that specific case.

What test-driven development guarantees is that new code will not break previously working code, as the test suite grows. It does not guarantee that the code handles all situations, as it depends on the written tests and their coverage. Also, it does not guarantee that the developer did not cheat, as he is aware of the input of each test. As any other technique, it depends on the good will of the involved persons.

Another major advantage of using testing frameworks is the ability to easily re-factory. In today's development environments often happens that different teams put together smaller programs to be used by other teams to build more complex applications. If at any given time one of these smaller programs needs to be re-implemented, because of efficiency problems for example, the developer, being the same or a new one, just needs to make sure that the new implementation passes all the tests. This, in most cases, automatically makes complex programs, that use the re-factored code, immediately also work. This scales very well, meaning that if the complex program also passes its own test suite, because the re-factored program respected the old code behavior, then even more complex programs that rely on both of these will probably automatically work with the new code.

The Open-Source community is investing in this approach for code development. Examples are the unit-testing of Java, Ruby or Perl modules, and the number of available frameworks for testing. Even in the corporate world more and more often companies release their applications as open-source projects and many times rely on testing frameworks to make sure their code is not only working, since there is the chance of many more contributions and changes to the original code, but also guarantee that the most recent code still maintains the original behavior.

In this article we would like to give a tour of the initiatives and techniques that are being used by the Perl community to guarantee some minimum quality standard on the modules made available by the well known Comprehensive Perl Archive Network[1] (CPAN).

The article is divided in four main sections. First, section 2, presents briefly the CPAN archive, how it works and the available tools for the common Perl programmer to interact with it. Section 3 covers some of the available frameworks for writing tests for Perl Modules. These frameworks will be divided in three blocks: testing code behavior, testing documentation (both syntax and coverage) and testing module distribution. Trying to overcome the usual problem of discussing who certifies certification agencies, or who controls persons responsible for controlling others, section 4 presents an approach to testing tests using the code that was written to pass those tests. Finally, section 5 will focus on two community initiatives: the support for distributed testing on different architectures and operating systems, and the analysis of modules' code with the computation of a quality measure.

In summary, in this article we will present approaches that will help the developer to tell the user *here is my code* and *here is a way to show you that it works properly*. The certification itself is basically given by a positive outcome of the testing framework, obviously assuming that the tests themselves are trusted and were written in good faith.

2 Quick Introduction to CPAN

The Comprehensive Perl Archive Network (CPAN) has its origins (in concept and name) in the Comprehensive TEX Archive Network[2] (CTAN), the archive of TEX classes and packages. The CTAN idea is simple: create a centralized archive of modules, scripts and other tools related to a community of users, where any user can contribute, and the entire community can make use of the entire archive. This same approach is being used by other communities, like R developers with the Comprehensive

[1] http://www.cpan.org, http://search.cpan.org/

[2] http://www.ctan.org/

R Archive Network[3] (CRAN), Ruby Application Archive[4] (RAA) for Ruby developers, or Python Package Index[5] (PyPI) for Python developers.

Every one of these archives shares the same basic principles, but adds different functionalities according to their user's needs. These archives' baseline of functionalities can be described as:

1. any user can contribute any code/package/module (and, normally, the contribution is not reviewed);

2. there are no restrictions on adding contributions that mimic the behavior of other contributions;

3. there is some kind of taxonomy that allows the contributor to classify his contribution (usually contributions are indexed also by contributor name);

4. any user can search the catalog and download any package he wants. Usually these archives also add some text explaining the package so that the user can choose using something more elaborate than just the package name, author information and contribution class.

It is easy to notice that these operating rules are too liberal. In particular, rules 1 and 2 lead to anarchy very easily, as users can contribute bad, buggy or malicious code, and can even contribute code with similar behavior of other already archived. Therefore, the user searching for a module will have to deal with the questions: how to be sure that a module can be downloaded and used safely; and how to choose from a set of possible modules that can be used to perform the same task.

Unfortunately, unless the rules get replaced by new rigid ones, these two problems do not have a simple solution. Of course this contribution flexibility motivates and promotes more developers to make their code available in the archive. Nevertheless, some extra meta-information can be added to the repository, making the task of choosing what modules to download, and use, easier for the user.

With this objective, CPAN includes a few extra meta-information mechanisms:

1. each module can be rated, as if it were a movie, by any user. The user can add comments on it as well. Unfortunately it is not easy to convince users to rate modules. While some perform that task, most CPAN modules are not rated or commented on;

2. each module has a clearly associated author, with e-mail address and picture (when available). As authors get well known and get reputation, users get confident in using their contributions. This is especially true given the number of conferences organized each year by the community, that work well to introduce developers;

3. together with the module description it is possible to visit, automatically, its documentation. Also, as the community suggests a well structured template for documentation, it is relatively easy to compare modules' documentation and their completeness (therefore, making it easier to choose which module to use);

4. the date of the last update is also shown. Usually modules with old dates are not maintained. But it can also mean the module is stable (although this is rarely the case);

5. a detailed matrix of the tests and their status (pass/fail) on different platforms is also shown. Refer to section 5.1 for more information on how this data is computed.

[3] http://cran.r-project.org/

[4] http://raa.ruby-lang.org/

[5] http://pypi.python.org/pypi

Although not related to tests and software quality, we would like to add a final remark: there are some applications[6] that can be used to install modules from CPAN. When installation fails the user has the option to automatically report the failure. This will issue an e-mail that will be sent to the module(s) maintainer(s).

3 Pure Test-Driven Development

Every Perl module available on CPAN includes a test suite (of course, there are a couple of exceptions that prove the rule), from simple and incomplete to fully featured test suites.

These test suites' appearance was not guided by any rule or requirement imposed by CPAN. That would not work! Instead, an initial framework for testing was born and, at that time, the only tool available to bootstrap empty Perl modules from a skeleton template incorporated one or two simple tests of module usability (for instance, checking the module loads).

It was this initiative that resulted in a greater number of people writing tests, not because they were a requirement. Nobody really cares to complain if a module does not include a test suite, but the author, when creating the module, and noticing there is already a basic framework for writing tests, tries to maintain it, adding new tests.

Three different aspects of Perl modules began to be tested (there are some other aspects that could be included here but that we decided to ignore them, as they can be considered part of one of the categories we present here):

- tests started with simple *code testing* (section 3.1), just like any other programming language unit-testing framework;

- then followed the addition of *tests for documentation* and documentation coverage (section 3.2), checking the syntax of the documentation and its completeness;

- finally, tests for *checking distribution contents* (section 3.3) are arising, to ensure every file required is being shipped in the module tarball.

These same tests can be divided in two categories:

- **developer tests** should be checked only by the programmer, locally, before distribution. They normally check that all files are present in the distribution, that the documentation is complete and with the correct syntax;

- **user tests** should be shipped with the module and should be run by every user that wants to install the module. They usually test the algorithm of the application. These tests will guarantee that the relevant code works independently of the architecture, operating system or Perl version, as developers might have some difficulty in having different machines for testing purposes (check section 5.1 for more initiatives on multi-architecture testing).

Note that while we focus primarily the testing frameworks for Perl module development, the Perl core itself has a complete test suite.

[6] The fact that more than one tool exists for this task is an example of the multiplicity of available modules for performing the same task.

3.1 Testing Code

Testing code is the more usual paradigm of testing. As described in the introduction of this article, the developer is invited to declare the behavior of its method or applications, writing a set of typical inputs (hopefully including some edge cases), and the corresponding correct results (outputs).

Depending on the complexity of the method or application being tested, the complexity of the test can grow. A common good practice is to start writing tests for small auxiliary functions at the beginning of the project, in such a way that every new function has all its dependencies well tested.

The more usual tests for code can be divided in the following categories [Lc05]:

- **Comparing the return value with the correct answer:**
  Most tests receive an input and check the output against a gold standard, the correct answer. This verification can be as simple as checking if the return value is the same as a specific integer or string, or checking if the return value is inside the expected range of possible answers.

  Perl frameworks implement a set of functions to help implement these tests. Each test includes the code to be tested but also a small description of what is being tested. This is useful, as it makes the process of reading test reports easier.

  ```
  is( add(2,3) , 5 , 'Simple test for add' )
  ```

  Modern test frameworks provide more flexible testing mechanisms, so that strings can be matched against regular expressions, or full complex data structures matched against other sample structures.

  ```
  is_deeply( parse('2+3') , ['+', 2, 3] , 'Parse sum op' )
  ```

  Also, there are other modules that allow checking for other kind of output, like text document generation, analyzing XML structures (matching it against a schema or simply analyzing the contents of some XPath expressions), or checking the values present on a database.

  ```
  my $snoopy = Dog->new("Snoopy");
  isa_ok( $snoopy , 'Animal');    # Snoopy is an animal
  can_ok( $snoopy , 'bark');      # Snoopy is able to bark
  ```

  All these tests are of the same kind: with some input, the function, method or application delivers the correct output.

- **Checking that the module is loadable without errors:**
  A fundamental test for any program written in any language is that it compiles or gets interpreted correctly without syntax errors. This test is automatically generated for any new module created by the common Perl module generators, ensuring that each new module that gets in CPAN ships with this basic guarantee.

- **Analyzing an objects' hierarchy and available methods:**
  Object oriented programs can create classes and objects at run time. These classes need to be tested, for instance, checking their parent information (*isa* relationship) and checking that they can handle some specific methods.

More complicated testing mechanisms are also supported in Perl. For instance, there are modules for testing regular expressions (checking that they match the required string and that they will not match false positives), XML (that the document is well formed, or valid against a specific schema), XPath expressions (that the expressions are correct and that they yield the correct value when matched against an XML document), images (checking their size, checking specific pixel colors, etc), web applications (simulating an user, interacting with the web application and analyzing the resulting web pages) and many more.

Last, but not least, testing of coding standards (or best practices [Con05]), such as indentation, or function or variable name capitalization, is also contemplated.

3.2 Testing Documentation

Perl has a great advantage with documentation over some other languages. While Java or C support JavaDoc[7] and Doxygen[8] respectively, they were never seen as a real standard for writing documentation (probably more with JavaDoc than Doxygen), Perl has a *de facto* standard, named POD[9], that is broadly used by all Perl modules.

It can be used in a literate programming approach, where the programmer can interleave code with documentation. Unlike JavaDoc or Doxygen, Perl is very flexible on the POD usage. There is no requirement to write the documentation near the respective functions, for example (JavaDoc or Doxygen work as code annotation).

POD has a simple textual format. It supports a few headings, some lists, basic word highlighting, and verbatim sections.

This documentation should also be tested and, on newly created modules, two kinds of tests are automatically created:

- **Checking documentation syntax correctness:**
  Given that Perl uses a specific syntax for writing documentation, and that that documentation is interpreted to generate the documentation in different formats (Unix man-page, HTML, PDF, LaTeX), it is important that the documentation syntax is correct. For that purpose, a syntax checker exists that is able to search for all documentation present on a Perl module directory and complain about syntax errors.

- **Checking documentation coverage:**
  The second level of quality assurance for documentation is its coverage. It is not enough that the documentation has a valid syntax, it is also required to cover all methods implemented. This framework parses the documentation and ensures that each function or method defined has a corresponding documentation section. As some methods might be irrelevant for documentation (maybe because you just do not want to make users aware of it), their names can be prefixed with an underscore and the testing mechanism will ignore them.

Once more, these testing approaches do not guarantee any documentation quality, but they assist the developer who is interested in writing and maintaining complete documentation.

3.3 Testing Distribution

When creating a tarball with the module files and uploading it to CPAN servers, the developer needs to ensure the tarball is complete, and that all files are edited accordingly. In this area, the Perl community also offers some frameworks for testing purposes:

[7] http://java.sun.com/j2se/javadoc/

[8] http://www.stack.nl/~dimitri/doxygen/

[9] Stands for Plain Old Documentation (whilst old, it is not dead, and has been evolving in the last year).

- **Checking distribution tarball completeness:**
  When developing a program or module, there are a bunch of files that are created with small debug programs, small test cases and other information (such as version control software files). These files are not part of the distribution that should be released. Therefore, Perl adopted the concept of a manifest file with the list of files to be included in the release tarball.

  While this solution is great, it is also annoying. Every time a new file that should be included in the distribution is created the developer needs to edit the manifest file and add the new file. If he forgets to do so, the distribution will be incomplete and unusable. Therefore, a mechanism to ensure that all files that are listed in the manifest file exist is required. For that to work, this test uses another manifest file, with regular expressions that match the files that should be ignored and not included in the distribution. Then, if a file appears that is not listed in any of the two manifest files, the test will fail.

- **Checking that generated files were properly edited:**
  Another kind of test that should be performed prior to the module distribution is ensuring that all files that were generated by the common module generation tools were edited. To explain the relevance of this test I should explain that about 8 years ago, many modules in CPAN had as author "A. U. Thor", the name used by one of the modules generation tools.

  These tests, named *boilerplate*, ensure the programmer edited the generated code, installation and other documentation files.

Other examples of distribution testing include the analysis of modules and sub-modules, change log and read me files, ensuring all refer to the same version.

4 Testing Test Coverage

The main problem when writing tests is the question about how to test the quality of the tests. In fact, testing tests it not really possible. But we can assess how much of our code is covered by currently written tests.

Perl offers a framework for this purpose. For each test, each line of code that gets executed it counted. This results in a table that, for each line of code, shows the number of times it was executed. This kind of coverage testing is great when writing tests. If the written code has some conditional structure, for example, there should be a test to exercise each of the possible branches [Joh05]. All this information is presented to the user in HTML format, with all the code annotated with information about how many times each line was executed. Moreover, it also presents some basic statistics, showing the percentage of subroutines or branches that have been tested.

With all this information it becomes easier for the programmer to find out what areas of the code need extra testing.

5 Automatic Distributed Testing

As already stated, some developers do not have access to all platforms (CPU, architectures or operating systems) where Perl can run. This leads to a problem: how can you know if a specific module works correctly on a specific platform? To overcome this problem the Perl community created a distributed testing service (see section 5.1).

All these initiatives (those already discussed and this distributed testing service) assume the good will of the developer, who wants to make his code better. As an independent initiative, another project named CPANTS[10] (The CPAN Testing Service) was created. The goal of this project is to provide some sort of quality measure (called Kwalittee) by code analysis (see section 5.2).

5.1 Distributed Testing Service

The Perl community has a CPAN Testers framework[11]. Community volunteers can join the initiative, registering one or more machines (together with its architecture, operating system and Perl version) and offering to test module distributions uploaded to the archive. There are some tools to help in finding the latest uploaded modules so that CPAN Testers can know what to test. This process can be completely automatic or semi-manual, depending on the testing tool chosen by the tester.

Currently there are testers running Perl versions from 5.004_05 (the maintenance branch of a Perl version with more than ten years old) to the most recent, development branch 5.13.2 (about two weeks old). Operating systems available for testing include OS/2, SunOS/Solaris, IRIX, Mac OS X, OpenBSD, VMS, Windows, Linux, AIX, etc[12].

If the configuration, compilation and installation succeeds, a success report is inserted in a database that can be queried by any user (therefore, knowing if that module is stable for a specific platform). If some error occurs, an e-mail with a full report is generated (with the full output of the compilation process, and details on the platform and Perl configuration variables) and sent to the module maintainers. This same report is stored in the database, so that any user can query it.

5.2 Automatic Quality Measuring by Code Analysis

Being a CPAN Tester is a risky task. While many CPAN Testers test modules in a virtual machine or some kind of sand box, they are risking their machine or installation to malicious code. As far as the authors are aware, no real malicious code was found yet on CPAN, but it is possible (although, if detected, user would be banned and modules deleted).

CPANTS is another project that aims to evaluate Perl modules. Instead of trying to compile, test and install modules, this approach grabs modules and inspects their code (not executing it).

The module code is checked against a list of Kwalitee[13] metrics. Unfortunately, as the basic idea rejects the interpretation of code, the amount of analysis possible to be performed is reduced.

Nevertheless, CPANTS tests are relevant. To mention some examples, CPANTS checks if all module dependencies are listed correctly in the package meta-data file, if any file mentioned in the manifest file is missing, if every module file has a version number, if there is a clear license in the documentation, if a read me and a change log files are present, etc.

Given the automatic behavior of CPANTS, and its objectiveness, it can be almost considered a game, where modules with better module distributions get points for their kwalitee. Therefore, the web site can show a sorted list of authors, that can *play* or *fight*, trying to climb up the table. This playful approach can motivate developers to improve their distributions.

[1]0 http://cpants.perl.org/

[1]1 http://www.cpantesters.org

[1]2 Unfortunately not all platforms have all Perl versions available for testing. Nevertheless, more common operating systems have most Perl versions available. You can check a detailed matrix of what Perl version are available in what Operating System at http://stats.cpantesters.org/osmatrix-full.html

[1]3 Kwalitee is the name chosen to represent this pseudo-quality information.

6 Conclusions

In this article we provided a brief tour to the mechanisms implemented by the Perl community to help the development of modules, guaranteeing they include tests, delivering methods to test the written tests (namely, checking their coverage), and distributed approaches to test modules in different contexts, architectures and platforms.

While these approaches can not be seen as a formal certification approach, they can easily motivate open-source developers to include quality assurance tests. Also, a test suite can be used to help demonstrate the end user that the code (implemented by the developer or developers) actually does what it advertises, without the need for the user to browse thousands of lines of source code. Assuming the good faith of the tests writers, we can admit that a positive outcome of running the test suite certifies that the module is working properly, at least for the cases tested. And luckily, enough edge cases and gray areas tests were included to certify that the module or application is working as expected.

From these different initiatives we would like to stress that it is important that every kind of software packaging approach includes mechanisms to introduce software tests and that, when they are created by some kind of automatic generator tool, some simple tests are automatically generated. This will motivate the developer to keep the test suite up-to-date. On the other hand, if the task of adding a testing framework is a developer task, it is natural that this framework will often never be used. Moreover, if the testing framework includes any tool to test the tests' coverage, it will help interested programmers in detecting what sections of code are lacking testing.

Finally, knowing that these test suites will be run in different architectures and platforms automatically, without the need to ask for it, can lead the developers to have a greater interest in writing complete tests.

Clearly there are plenty of advantages on creating and maintaining a test suite, this is so obvious that Perl itself has one. Once you build the Perl interpreter you can run this test suite to validate that the binary files were built correctly, and that Perl behaves as expected. The core modules shipped with the Perl distribution test suites are also executed in the process to make sure that everything works as intended.

Acknowledgments

This work was partly sponsored by project grant PTDC/EIA-CCO/108995/2008 (An Infrastructure for Certification and Re-engineering of Open Source Software), from Science and Technology Foundation of Portugal.

An extra acknowledgment to Paul Johnson, author of `Devel::Cover` module (that tests test coverage) for comments and suggestions.

References

[Bec99] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.

[Con05] D. Conway. *Perl Pest Practices* . O'Reilly, Sebastopol, CA, 1st ed. edition, 2005.

[Joh05] P. Johnson. Devel::Cover – an introduction. In Simões and Castro (eds.), *Yet Another Perl Conference, Europe (YAPC::EU)*. Pp. 85–90. Braga, Portugal, August 2005.

[Lc05] I. Langworth, chromatic. *Perl Testing: A Developer's Notebook*. O'Reilly, Beijing, 2005.

[Max03] E. M. Maximilien. Assessing Test-Driven Development at IBM. In *In Proceedings of the 25th International Conference on Software Engineering (ICSE-03*. Pp. 564–569. IEEE Computer Society, 2003.

Author Index