Productivity for Proof Engineering

Mark Staples, Ross Jeffery, June Andronick, Toby Murray, Gerwin Klein, Rafal Kolanski
NICTA, Australia and
University of New South Wales, Australia
Level 4, 223 Anzac Parade, Kensington, NSW 2052 Australia
+61 2 9376 2000
<firstname.lastname>@nicta.com.au

ABSTRACT

Context: Recent projects such as L4.verified (the verification of the seL4 microkernel) have demonstrated that large-scale formal program verification is now becoming practical.

Objective: We address an important but unstudied aspect of proof engineering: proof productivity.

Method: We extracted size and effort data from the history of the development of nine projects associated with L4.verified.

Results: We find strong linear relationships between effort and proof size for projects and for individuals. We discuss opportunities and limitations with the use of lines of proof as a size measure, and discuss the importance of understanding proof productivity for future research.

Conclusions: An understanding of proof productivity will assist in its further industrial application and provide a basis for cost estimation and understanding of rework and tool usage.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification–*correctness proofs;* D.2.9 [**Software Engineering**]: Management – *productivity.*

General Terms

Management, Measurement, Verification.

Keywords

Proof engineering, Productivity, Proof sizing, Formal verification.

1. INTRODUCTION

The L4.verified project completed the machine-checked formal verification of the full functional correctness of the source code (and later also the binary code) of the embedded systems microkernel seL4 [5]. It demonstrated that the long-held dream for the use of formal verification in software engineering from specification through to code is becoming realizable. Formal verification is cost-effective for some highly critical systems [5], but is too expensive for most projects. Increasingly, software systems are safety- or security-critical and so could benefit from formal verification to provide direct evidence about system dependability. Nonetheless, to broaden the reach of formal verification requires that its cost be reduced.

ESEM'14, September 18-19, 2014, Torino, Italy.

Formal proofs are not present in traditional software engineering, but are an intrinsic part of projects using formal verification [2]. The major cost in the L4.verified project was the effort required to create and maintain proofs. While 2.2 person-years were required to design and implement seL4, the formal verification took more than 20 person-years [5]. The importance of p*roof engineering* has been previously recognized for hardware verification [4], and also for the L4.verified project [2]. However, most proof engineering research has focused on proposing new technologies. For cost-effective proof engineering, a key consideration is *proof productivity*. Understanding proof productivity is also key for effort estimation models for projects using formal verification [2].

We report on a study of proof productivity based on a retrospective analysis of nine formal verification projects. We first provide background on software development productivity and proof engineering. Then we describe the method, analysis and discussion, of our study of overall productivity for these projects, and of productivity variation across individual engineers. We find that effort is highly correlated with proof size. This result was surprising to the verification experts involved in the project: clearly, there are proofs that are much simpler and less complex than other proofs, and everyone will have seen small, elegant mathematical proofs that were very hard to find initially, and so would need uncharacteristically high effort to produce. Our results show that this effect did not have a great influence over the lifetime of larger software verification projects.

2. BACKGROUND

2.1 Software Development Productivity

There is now a good understanding of factors that drive software development productivity and the differences in productivity experienced in different organizations and software domains [10]. A distinction is made between context factors, scale factors and effort drivers. An extensive study [11] identified the most common context factors as programming language, application domain, and development type. Factors that influence the effect of scale are team size, process maturity, project novelty, complexity of interfaces, and project management complexity [10]. Effort drivers [11] can include team capability and experience, software complexity, project constraints, and tool quality and usage.

In software, productivity has usually been measured as output in terms of lines of code (LOC) or function points produced per unit of effort expended. So, for example, the European Space Agency reported 0.35 kLOC per person month for on-board systems and 0.58 kLOC per person-month for "other" systems [10].

2.2 Formal Verification & Proof Engineering

Formal verification can show that all possible behaviors of a program are allowed by a specification. Unlike testing, formal verification checks all behaviors for all allowed inputs. The semantics for programs and specifications are defined by

mathematical models of the behavior and requirements of real-world computer systems. The checks performed by formal verification use mathematical proof. Programs can be large and complex, as is the semantics of real-world computer systems, so verification proofs are too. Proof engineering is supported by tools and techniques to manage proof size and complexity. Proofs have many qualities. The most important is logical soundness, but despite the size and complexity of verification proofs, their soundness can be achieved with extremely high confidence using modern theorem provers such as Isabelle [6], used in the L4.verified project [5]. Other qualities targeted include readability [12], and maintainability [3]. Previous research has recognized the importance of proof productivity [9], but apart from laboratory experiments on user interaction [1] we find no prior reported empirical studies of proof productivity.

Isabelle is an interactive theorem prover. Users create proofs by writing proof scripts, which Isabelle checks for validity by execution. Theories and formal specifications are also represented in these scripts. Proofs contain many steps achieved by automated search or decision procedures, but because of the extreme complexity of formally verifying deep semantic properties of programs, fully automatic proof is not feasible. So, these proofs also contain some hand-guided steps. The overall interaction style is much like that of programming [1]. In proof engineering, the most costly input is the time required by proof engineers to create and maintain proof scripts.

3. METHOD AND DATA

We conducted a retrospective study of projects creating formal specifications and proofs built on the main L4.verified result. For the first five projects in Table 1 we derived new data sourced from management records, weekly reports, and the Mercurial version control repository used to record changes to the proof scripts. Data on the final four projects in Table 1 was sourced from projects reported in [5] with additional input from project managers. Below we describe the projects, how we measured effort and size, and the productivity factors we studied.

3.1 Projects Studied

We selected nine non-trivial completed projects with cleanly identifiable outcomes (i.e. not intermingled with other work), which all used Isabelle for proof or specification. The L4.verified project and follow-up work produced three formal specifications of seL4, at increasing level of abstraction: the *Exec* specification models an (executable) representation of seL4's design[1]; the *Abstract* specification is a complete functional specification; and the *CapDL* specification is used to initialize sel4-based systems by only describing the *capabilities* (access rights) between components, abstracting away everything else. The six other artifacts we study here are proofs. Three of them are proofs of refinement: *Code-to-Exec*, *Exec-to-Abstract* and *Abstract-to-CapDL*. They show that all the behaviors of one specification (e.g., *Exec*) are included in the behavior of a more abstract specification (e.g., *Abstract*). We also study two security proofs: *Info.flow* and *Integrity*, showing that (the abstract specification of) sel4 enforces information flow and integrity of components running on top, according to a given security policy describing allowed access rights. The last proof we study links a capDL description to the corresponding security policy. The original L4.verified project produced *Code-to-Exec Refinement*, *Exec-to-Abstract Refinement*, *Exec Spec*, and A*bstract Spec*. Together, these produce the proof of functional correctness of seL4, showing that seL4's C code is correct with respect to its functional, abstract specification. The proofs are described elsewhere [5]. For each project, we explicitly defined the subset of the overall repository containing just the relevant proof script files and changes occurring during the term of the project. All projects worked with the same background theory from L4.verified, and used a broadly consistent proof scripting style.

[1] This executable specification is generated from a Haskell implementation of seL4. The effort numbers reported here for producing Exec include the effort of producing the Haskell implementation, plus the effort of producing the Haskell-to-Isabelle translator.

3.2 Measuring Input (Effort)

Effort was determined by managers for each individual for each project, where possible using weekly reports from individuals. For each person, for each week, fractions of a person-week were recorded for each of 'Initial Discussion', 'Actual Proof', 'Tool Improvement', 'Other Project', 'No PR', and 'Leave'. 'No PR' was recorded when no progress report or other historical record was available for an individual in a week. These fractions were checked to sum to the EFT-person-week (1.0 for full-time staff) for each week for each person. In productivity calculations below, we measure total effort as the sum of 'Initial Discussion' and 'Actual Proof', and add 'No PR' values in sensitivity analyses.

As a data quality check, we cross-referenced effort records with Mercurial. The only "missing" changes were for people with zero proof effort (their effort was on other work). Some changes were made by people with no effort records, but after review, these were judged by the project managers to have been a small number of inconsequential changes, and were excluded from analyses. As noted above, data for the first five projects in Table 1 was derived anew for this paper while data for the final four projects was derived from the earlier paper [5] with assistance from the authors.

3.3 Measuring Output (Proofs)

We measured Isabelle proof scripts using two variations on Lines of Proof, analogous to Lines of Code from traditional software development. (We discuss this in section 5.1.) For each change, Mercurial reports lines added (including lines modified), lines deleted (including lines modified), and the parent commit(s). We exclude changes that only merge parallel work, and so all changes that we consider (except for the first) have exactly one parent. We excluded large outliers (less than 1% of changes) that were only textual or syntactic changes (i.e. had no substantive proof work) – these were mostly file or constant renames. We define *lines-work* as the sum of raw lines added and deleted. For each change in each proof, we examine the repository at that time and calculate the normalized line count of the proof by excluding comments and white-space. We define *repo-delta* as the absolute difference in this size for each change compared to its parent. This is somewhat like lines added minus lines deleted. Neither is an ideal measure of output or work: *lines-work* includes all comments and white space and double-counts modified lines, whereas *repo-delta* excludes modified lines. Nonetheless, they provide bounds on the "true" lines of work. *Final size* is final lines of proof.

3.4 Productivity Factors

We identified potentially important factors in section 2.1. The context factors (programming language, application domain, and development type) are all constant for our projects. The scale factor of process maturity is constant, but team size and

novelty/complexity vary. The effort driver of tool quality and usage is constant, but others vary in our projects.

We investigated the following project-level factors: final size of the project (excluding comments and white-space); maximum team size; schedule pressure (as a project constraint); and overall difficulty (reflecting novelty and complexity). Schedule pressure and overall difficulty were recorded on a 5-point Likert scale from very high to very low. Assessments of factors for each project were jointly agreed by two managers. We also investigated four individuals factors, i.e. years experience with: Isabelle, formal methods or theorem proving (including Isabelle), the domain (of operating system development or verification), and work on L4.verified projects specifically. These values were collated by a project manager in consultation with the individual engineers.

4. ANALYSIS AND RESULTS

We present results first for the nine projects overall, and then for the 24 individual contributions to five of those projects for which we had the data (first 5 in Table 1). Both R and SPSS V22 was used for analysis.

4.1 Overall Project Productivity

Table 1 shows characteristics for the nine projects.

Table 1 Characteristics of the nine projects. Final Size is in kilo-Lines of Proof, Total Effort in person-weeks. Schedule Pressure and Overall Difficulty range from very low to very high. Maximum Team Size is headcount.

	Final Size	Total Effort	Sched. Press.	Overall Diffic.	Max Team
CapDL Spec	2.14	27.5	AV	LO	5
CapDL-policy proof	0.85	11.3	LO	AV	1
Abstract-to-CapDL Refinement	20.4	66	AV	AV	5
Integrity	7.05	28.5	V.HI	HI	4
Info. Flow	27.1	75.9	V.HI	V.HI	8
Exec- to-Abstract Refinement	96.6	368	HI	V.HI	6
Code-to-Exec Refinement	53.34	138	V.HI	HI	6
Exec Spec Haskell	6.01	92	AV	HI	1
Abstract Spec	4.9	15.3	AV	AV	3

Productivity varies, but this is explained by a constant overhead: *Total Effort = 9.98 + 3.35*Final Size*. Figure 1 shows this linear relationship. For our nine projects, the correlation is strong (R^2=0.914, p<.001), and not sensitive to inclusion of the 'No PR' effort. (We recognize the limitations arising from our small sample size.) 'Final' size was taken from the end of the initial development periods, to match the periods for the reported effort. Person-years from [5] are scaled to person-weeks assuming 230 working days per year and 5 days per week. Visually, there may be two outliers, both from [5]: the very large abstract refinement proof, and the very detailed executable specification. Effort for the latter includes the two person-years spent programming the Haskell prototype of seL4. Although the correlation is high, we investigated possible project-level productivity factors in multiple regressions. The effect of these is small and not significant at 0.05, but there was weak evidence that schedule pressure is associated with decreased effort, and overall difficulty and max. team size with increased effort. These results are consistent with prior research [10]. We also investigated each possible individual productivity factor (the various types of years of experience) in multiple regressions against individuals' total effort. There was no evidence that any were significant in this dataset.

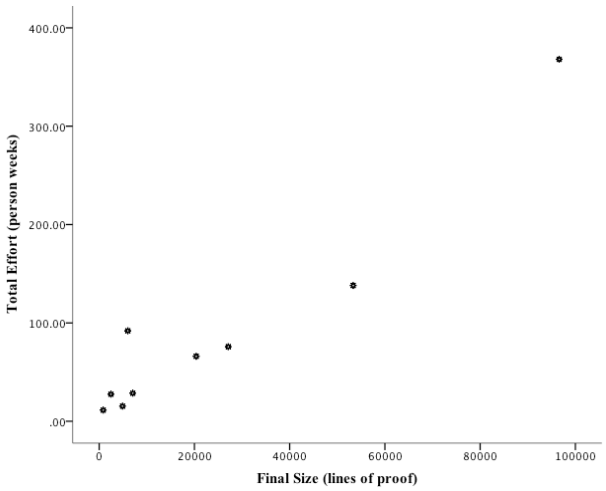

Figure 1 Scatter plot of project total effort vs. final size

4.2 Individual Productivity

Figure 2 shows the linear relationship between size and effort for all of the 24 individual contributions to five of the projects for which we had individual data. The correlation is very strong (R^2=0.93, p<0.001), and is not sensitive to inclusion of the 'No PR' effort (R^2=0.92, p<0.001). With the alternative size measure *lines-work*, there are of course different coefficients for the line of best fit, but again there is a strong linear relationship (R^2=0.91,

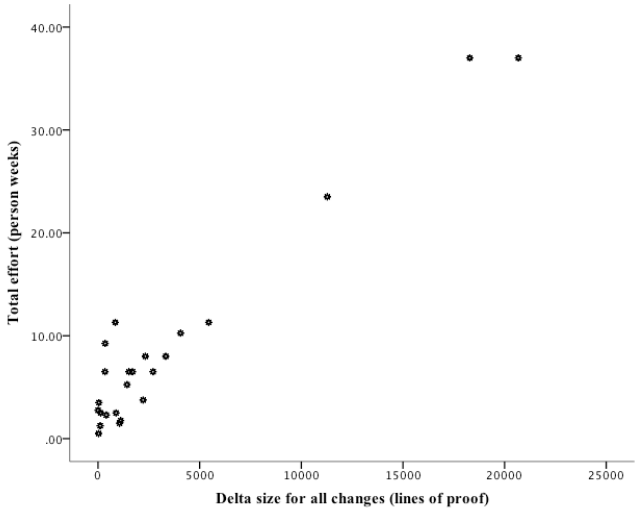

Figure 2 Scatter plot of total effort vs. sum of *repo-delta* size of all changes made in the 24 individual contributions to the four proofs and the specification.

p<0.001 for total effort, and R^2=0.91, p<0.001 with 'No PR' included). Final size is not meaningful for individuals' work, so is not examined.

5. DISCUSSION

5.1 Sizing Proofs

Lines of Code is widely known to be an imperfect measure of software. Lines of Proof is likewise problematic and improved size measures are required. An ideal measure of size would reflect the size of the proof problem: the difficulty or content of the proof goal. For formal verification this should reflect the specification [8] and the program.

Fine-grained measures of effort spent on individual proofs are difficult to collect in practice. However, we have found Lines of Proof are very highly correlated with effort. This may support the use of lines as a proxy for effort in future research. Given a consistent proof style, this result could help to validate new size measures against lines of individual proofs. Alternatively, under a given size measure, this result may help to validate productivity improvements from new proof engineering technologies.

5.2 Threats to Validity

In software engineering it is common to consider experimental validity in terms of construct, internal, and external validity [13]. Runeson and Höst [7] add reliability to this list. We discuss above the construct validity concern of Lines of Proof being a poor measure of size. The size measures *lines-work* and *repo-delta* both have limitations as discussed in section 3.3, but bound the true value of lines changed, and our overall results are supported under either measure. Subjective measures used in this study have been carefully defined for the persons from whom measures were obtained so as to avoid construct validity issues to the extent possible. There is a possibility that factors not measured in this study have an impact on productivity. Because the interaction style of our proofs resembles programming [1], we have carefully investigated factors previously reported to affect programming productivity, in order to ensure internal validity in our study. Our use of projects from a single context (L4.verified) aids internal validity, but limits external validity. It is not yet known if or how our findings may generalize to formal verification projects beyond L4.verified. We tried to ensure reliability of data collection and analysis by having a review process with multiple researchers.

6. CONCLUSIONS AND FUTURE WORK

Proof engineering research can help to bring the benefits of formal verification to more software engineering projects, but the assessment of cost-effectiveness of formal verification (and its improvement) hinges on understanding proof productivity. We have shown that proof effort and size are very strongly linearly related, in our study of nine projects building on L4.verified [5]. This result holds for the projects overall, but also for the individuals working within those projects.

An understanding of proof productivity can inform the creation of cost estimation models to select formal methods in projects, and also for detailed project planning. As with software development, proof productivity is likely to be affected by rework. We have seen initial evidence for this in the differences between sum of *lines-work* and final size. Deeper study is required on the impact of dependencies on proof rework and concurrent work.

Improvements to proof engineering may derive from proof automation, structures for reuse, refactoring, or proof patterns. A key question will be: do new tools or techniques improve on existing levels of proof productivity? The results in this paper provide an initial benchmark of proof productivity for future research. However, as discussed, Lines of Proof is a poor size measure for this, and improved size measures are required.

7. ACKNOWLEDGMENTS

Rafal Kolanski carried out this work while at NICTA and UNSW, but is now at Purdue University. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

8. REFERENCES

[1] Aitken, J. S., Gray, P., Melham, T., and Thomas, M. 1998. Interactive theorem proving: an empirical study of user activity, *Journal of Symbolic Computation*, vol. 25 (2).

[2] Andronick, J., Jeffery, R., Klein, G., Kolanski, R., Staples, M., Zhang, H., and Zhu, L. 2012. Large-scale formal verification in practice: a process perspective. *ICSE 2012*. 1002-1011.

[3] Curzon, P. 1995. The importance of proof maintenance and reengineering. *Int. Workshop on Higher Order Logic Theorem Proving and Its Applications: B-Track*, 17-32.

[4] Kaivola, R. and Kohatsu, K. 2003. Proof engineering in the large: formal verification of Pentium4 floating-point divider. *Int J Softw Tools Technol Transfer*, vol. 4 (3), 323-334.

[5] Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., and Heiser, G. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comp. Sys.* Vol. 32 (1), 2:1-2:70.

[6] Nipkow, T., Paulson, L., and Wenzel, M. 2002. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. LNCS vol. 2283.

[7] Runeson, P. and Höst, M. 2008. Guidelines for conducting and reporting case study research in software engineering, *Empirical Software Engineering*, vol. 14 (2), 131 – 164.

[8] Staples, M., Kolanski, R., Klein, G., Lewis, C., Andronick, J., Murray, T., Jeffery, R., and Bass, L. 2013. Formal specifications better than function points for code sizing. *ICSE 2013*. 1257-1260.

[9] Syme, D. 1998. *Interaction for declarative theorem proving*. Available from http://research.microsoft.com

[10] Trendowicz, A. and Jeffery, R. 2014. *Software project effort estimation: Foundations and best practice guidelines for success*, Springer.

[11] Trendowicz, A. and Münch, J. 2009. Factors influencing software development productivity – State of the art and industrial experiences, *Advances in Computers*, vol. 77.

[12] Wenzel, M. M. 2001. *Isabelle/Isar – a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München.

[13] Wright, H.K., Kim, M., and Perry, D.E. 2010. Validity concerns in software engineering research, *Proc. of FoSER 2010*, 411-414.