Empirical Study Towards a Leading Indicator for Cost of Formal Software Verification

Daniel Matichuk*†, Toby Murray*†, June Andronick*†, Ross Jeffery*†, Gerwin Klein*†, Mark Staples*†
*NICTA, Sydney, Australia
Email: firstname.lastname@nicta.com.au
†UNSW, Sydney, Australia

***Abstract*—Formal verification can provide the highest degree of software assurance. Demand for it is growing, but there are still few projects that have successfully applied it to sizeable, real-world systems. This lack of experience makes it hard to predict the size, effort and duration of verification projects. In this paper, we aim to better understand possible *leading* indicators of proof size. We present an empirical analysis of proofs from the landmark formal verification of the seL4 microkernel and the two largest software verification proof developments in the Archive of Formal Proofs. Together, these comprise 15,018 individual lemmas and approximately 215,000 lines of proof script. We find a *consistent quadratic relationship* between the size of the formal statement of a property, and the final size of its formal proof in the interactive theorem prover Isabelle. Combined with our prior work, which has indicated that there is a strong linear relationship between proof effort and proof size, these results pave the way for effort estimation models to support the management of large-scale formal verification projects.**

I. Introduction

Recent software attacks on critical systems such as cars [1] and pacemakers [2] have helped to increase demand for *formal software verification* [3], whereby the security, safety or reliability of software is demonstrated using mathematical techniques. Desired properties are given as *formal statements*, which are then shown to be satisfied using *formal proofs*. Formal verification is recognised as providing the strongest guarantees about software behaviour: it is mandated for the highest assurance level of the Common Criteria standard [4], and its importance for complementing software testing is recognised in the DO-178C certification scheme for avionics [5].

Some recent landmark formal verification projects include the certified compiler CompCert [6] and the microkernel seL4 [7]. These were multi-year efforts, producing hundreds of thousands of lines of machine-checked proofs. Such projects use interactive theorem proving, which requires human creativity and effort to guide the proof but is not constrained to simple properties or finite state spaces. This allows for proofs of strong properties about low-level software. Most of the effort in these projects is spent in writing machine-checked proofs.

Despite these successes, few projects have yet tackled the challenge of verifying the implementation code (source or binary) of sizeable real-world software. Because there is little experience with large-scale formal verification, it is difficult to predict its required effort, duration and cost. Earlier work [8] has showed the need for a better understanding of how to measure artefacts in formal methods, to inform costs and estimation models. An analysis of the existing literature [9] revealed a shortage of empirical studies to provide industry with validated measures and models for the management and estimation of formal methods projects. As Stidolph and Whitehead [10] state "experienced formal methodologists insist that cost and schedule estimation techniques are unsatisfactory and will remain so until a large body of experience becomes available". In short, more research is required in the field of *proof engineering*.

Our ultimate goal is to provide estimation models for proof effort. In prior work [11], we analysed proof productivity, and revealed a strong linear relationship between effort (in person-weeks) and proof size (in lines of proof script), for projects and for individuals. Proof size, however, is only known when the project is completed. So, in this paper we examine the inputs to software verification projects: formal statements of the properties to be proved about programs, and formal specifications of the programs. Our goal is to identify measures of these formal statements that relate to the final size of their interactive proofs.

We present results of an empirical analysis of a large number of proofs written in the Isabelle interactive theorem prover [12]. We measure the size of each lemma, in terms of the total number of concepts needed to state it, and compare that to the total number of lines used to prove it. We analysed four large sub-projects of the seL4 verification work [7] as well as two proofs from the Archive of Formal Proofs (AFP) [13], an open collection of Isabelle proofs. From these 6 projects, we analysed a total of 15,018 lemma statements and associated proofs, covering a total of more than 215,000 lines of proof.

We find a consistent *quadratic relationship between statement size and proof size*, with the R^2 for the quadratic regressions varying from 0.154 to 0.845. One of the four seL4 sub-projects stands out with a lower R^2 and a significant collection of outliers, with proof sizes much smaller than would be expected given the statement size. Investigation revealed that these outliers were caused by over-specified lemma statements (see Section III-E), with large constants mentioned unnecessarily, effectively inflating their statement size. To test this hypothesis, we defined an idealised measure for statement size that is an approximation of its minimum size. Using this measure greatly strengthens the relationship between statement size and proof size across all the projects, with R^2 between 0.73 and 0.937. This implies that there is a very strong quadratic relationship between statement size and proof size, when statements are not unnecessarily over-specified.

This confirms an early hypothesis formulated by the leader of the seL4 verification project. Towards the end of the project,

he was required to provide an estimation (and justification) of time, effort and cost needed to complete the project. The experience from the proofs done by that time suggested that "for microkernel refinement proofs, proof size scales roughly quadratically with code size". Microkernel code is highly non-modular by nature, and so verification is dominated by proving invariants. Each invariant needs to be preserved by each feature, which in turn relies on and modifies data structures used by other features. Our current work provides empirical evidence for this hypothesis, by correlating proof size to statement size.

Our main contribution in this paper is the identification of a concrete measure for the size of formal lemma statements that we show has a strong, quadratic relationship with proof size. A formal statement of program correctness must be known before beginning its verification. Thus its size can serve as a leading indicator of proof size. Proof size has in turn been shown to be strongly correlated to proof effort [11]; the measure presented here is therefore the first potentially useful leading indicator for estimating proof effort for interactive verification projects.

II. Related Work

Research to date on formal methods measurement has concentrated on the measurement of formal specifications of programs and also on the relationships between these measures and system implementations. In Olszewska and Sere [14] the authors report on their use of Halstead's software science model [15] for the measurement of Event-B specifications. They used this framework to measure the "size of a specification, the difficulty of modelling it, as well as the effort". The specification metrics developed were seen as useful descriptors of the specifications studied when applied in the DEPLOY project [16]. Some research has also been carried out on other specification metrics. In 1987 Samson et al. [17] investigated metrics that might aid in cost prediction for software developed. They use McCabe's cyclomatic complexity metric [18] and lines of code to measure the implementation of a small system and measures of operators and equations to measure the HOPE formal specification. Although their sample size was small, they found a relationship between their measures of the specification and implementation. Tabareh's masters thesis [19] contained an investigation of relationships between specification and implementation measures. A number of specification metrics were defined for Z specifications. These were size-based metrics such as lines of code and conceptual complexity; structure based metrics such as logical complexity; and semantic based metrics such as slice-based coupling, cohesion and overlap. In a more recent paper Bollin [20] evaluated the use of specification metrics of complexity and quality in a case study comprising more than 65,000 lines of Z specification text. In King et al. [21] an investigation of Z and the use of testing in a commercial software development project revealed that, the Z proof was the most "efficient phase at finding faults" followed by the system validation phase.

In summary, previous research has investigated relationships between specification measures and implementation measures. Often this has been motivated by the desire to predict effort from implementation characteristics (e.g. size). Research has also investigated the types of measures that can be used for formal specifications. However we have been unable to find any research investigating the relationship between specification and proof measures, particularly size of proof. Our work aims to fill this gap.

III. Approach and Measures

A. Formal Verification - Background

A software verification project aims to establish that a program satisfies some property. A property might state that a program meets a high-level specification, or satisfies an invariant, or enforces security mechanisms. In an interactive theorem prover (ITP), such a property must be given as an unambiguous mathematical statement S, which may refer to the (formalised) program code and its formal specification. To prove S, we provide a proof P which appeals to the inference rules of the logic of the ITP, and may also appeal to intermediate lemmas. The set of all the definitions, intermediate lemmas and final proof P of S form the proof development establishing that the program satisfies the property expressed in S. S is then referred to as the top-level statement or property of the proof development. This P, known as a *machine-checked* proof, is the strongest known support for the assurance of S. Ultimately we wish to estimate the effort required to prove P for a given S. To this end, we chose to investigate the relationship between the complexity of S and the size of P.

We make the simplifying assumption that proof developments can be divided into three stages. (1) All definitions are written. (2) Statements about definitions are hypothesised as *lemmas*. (3) Each lemma is proved correct by writing an interactive proof. In practice, proofs never proceed quite this neatly. The validity of this assumption will be discussed in Section VII.

B. Proofs and Specifications

A proof development in an ITP is comprised of *definitions*, lemma *statements* and *proofs* of those lemmas. Definitions are used to introduce new *constants* and give them semantics, such as function specifications, program invariants or data structures. These definitions are given in the *term language* of the theorem prover, which has a precise syntax and semantics. Lemma statements relate constants, positing a fact that is then proved. Similar to definitions, lemma statements are written in the term language. Term and proof languages vary significantly between theorem provers, however they all share similar traits. A term can express logical statements, with connectives (e.g. conjunction, implication) and quantifiers (e.g. universal, existential). Proof languages have syntax for appealing to automated reasoning tools and previously proven results. A proof, in this context, is a sequence of appeals which eventually demonstrate that a lemma statement is true.

As a running example, consider the following definitions of two new constants C and E, which mention some propositions A and B from a previous proof development. They also depend on the exclusive-OR operator $\oplus$ and the usual logical connectives.

$$\mathsf{C} \equiv (\neg \mathsf{A}) \vee (\neg \mathsf{B})$$
$$\mathsf{E} \equiv \mathsf{B} \oplus \mathsf{C}$$

The constants implicitly form a *dependency graph*. A constant c directly depends on another constant c' if c' appears in the definition of c. This represents an edge in the constant

dependency graph from node c to node c'. In our example, E directly depends on B, C, and the $\oplus$ connective. We say that c depends on c'' if c'' is reachable from c in the constant dependency graph. In other words, c'' must be defined at some point in order to define c. In our example E depends on all of A, B, C, and the three connectives $\oplus$, $\vee$ and $\neg$.

Similarly lemmas also implicitly form a *dependency graph*: a lemma l directly depends on another lemma l' if l' is used to justify some step in the proof of l. Lemma l is then said to depend on l'' if at any point l'' had to be proved for the proof of l to hold. In our example, one can state the following lemmas to be proved (where the $\longrightarrow$ operator is logical implication).

$$\mathsf{E} \longrightarrow (\mathsf{A} \vee \neg \mathsf{B}) \quad (1)$$

$$\mathsf{E} \wedge \mathsf{B} \longrightarrow \mathsf{A} \quad (2)$$

The truth of Statement 1 and Statement 2 simply relies on the definitions of E and C and standard propositional logic. Their Isabelle proofs might look like the following:

```
lemma Eq1: E ⟶ (A ∨ ¬B)
  unfolding E-def C-def
  apply (rule HOL.impI)
  apply (elim xorE HOL.disjE HOL.conjE)
    apply (subst (asm) HOL.de-Morgan-disj)
    apply (subst (asm) HOL.not-not)
    apply (rule  HOL.disjI1)
    apply (elim  HOL.conjE)
   apply assumption
  apply (rule  HOL.disjI2,assumption)+
  done

lemma Eq2: E ∧ B ⟶ A
  apply (metis Eq1)
done
```

It is not necessary to understand these proofs. We observe, however, that the proof of the lemma **Eq1** refers only to facts from the **HOL** proof development, which defines the Higher Order Logic of Isabelle, as well as the facts that capture the definitions of C and E, **C-def** and **E-def** respectively, plus the fact **xorE** that in this example has already been proved in an existing proof development on which it builds.

C. Proof Size

In our analysis we consider a given lemma and relate its statement size to its proof size. Formally we define the size of the proof of a lemma l to be:

> **Proof size of lemma l:** the total number of source lines used to state and prove l, excluding definitions.

Note that this includes the proofs of all lemmas that l depends on. We exclude definition declarations (such as **C-def**) because, although they are part of the proof source, we are interested in the total number of *new* source lines written to complete stages (2) and (3) of a proof development; definition declarations are all written in stage (1).

We refine this notion of size slightly by only counting source lines from a particular proof development D. We call this the proof size of l with respect to D, defined as follows.

> **Proof size of lemma l with respect to proof development D:** the total number of source lines required to state and prove l, excluding definitions, as well as lemmas and proofs outside of D.

The reason that we contextualise proof size this way is because proof development is cumulative, and new proofs often build on old ones. For example, a proof development D, proving the correctness of Dijkstra's algorithm, would appeal to lemmas and definitions from an existing proof development G, a formalisation of graph theory. G would provide a definition of a graph, edges, paths, and would have lemmas proved about reachability. When measuring the size of the proof of a lemma from D in order to gauge its effort, one would consider the lemmas in G to have come at zero cost, and not take their size into account. More precisely, in general we measure the size of any proof of a lemma from a proof development D *with respect to* D, in order to exclude from its size any pre-existing lemmas on which it depends.

In our example, all the facts used in the proof of **Eq1** come from an pre-existing proof development except **C-def** and **D-def** that are the definitions of C and D respectively. Thus none of the direct dependencies of the proof of **Eq1** are counted when computing its size. The size of the proof of Statement 1 therefore is just its immediate size (i.e. 11 lines), while the proof of Statement 2 would be measured as its immediate size summed with the size of Statement 1 (i.e. $3 + 11 = 14$ lines).

D. Raw Statement Size

Here we define a measure for the statement of l that we found correlates well with the proof size of l as defined above:

> **Raw statement size for lemma l:** the total number of unique constants required to write the statement for l, including all of its dependencies, recursively.

"Unique" specifies that each constant is counted at most once per statement. This measure, importantly, is computable after stage (1) in the proof development as it only requires definitions to have been written, and does not depend on proofs. We refer to this as a statement's *raw* size. This is distinguished from the statement's *idealised* size, introduced later.

Similarly to proof size, it often makes sense to measure statement size with respect to some proof development D. Doing so excludes all constants that fall outside of D. However, while we measure the size of a *proof* in proof development D with respect to D itself, it often makes more sense to measure *statements* in D with respect to a larger proof development D' that includes D. In the example of the previous section, when measuring the size of Statement 1, we might choose to count the sizes of A and B, even though these constants have been defined in a pre-existing proof development, and even though proofs about A and B in this pre-existing development will not be counted in the size of the proof of Statement 1. The reason is that the effort of proving a new fact about a constant c (here e.g. A) might be highly impacted by the size of c even though it has been defined in a pre-existing development. In the context of the seL4 proofs we observed this effect to be extremely strong, where most statements referring to seL4's abstract specification would have proofs which follow the structure of the abstract specification and carry the complexity of reasoning about it.

In the example of the previous section, assume we choose to measure the size of Statement 1 and Statement 2 with respect to the entire proof development down to the axioms of the logic, i.e. including the definitions of A and B and all the operators. Assume that A and B are complex constants with sizes 100 and 200 respectively. For simplicity we assume their dependencies are disjoint, so C would have a size of $1 + 100 + 200 + 1 + 1 = 303$, where we add 1 for C itself, the size of A, the size of B and then the size of $\neg$ and $\vee$ (we assume for simplicity that they are defined axiomatically). We can then calculate the size of E as $1 + 1 + 303 = 305$, by adding 1 for E itself, 1 for $\oplus$ (also assumed to be defined axiomatically) and then the size of C. Note that we do not add the size of B, as it already has been considered when calculating the size of C, and we are only counting unique constant dependencies. Then Statement 1 would have a size of $305 + 1 = 306$, where we count the size of E and the size of $\longrightarrow$ (the other constants being already considered in the size of E). Similarly Statement 2 would have a size of 306.

Note that both proof size and statement size are inherently recursive; the proof size of l includes the size of its dependent lemmas, and the statement size of l is based on its dependent definitions. No attempt is made to measure the immediate size of any lemma, as this is far too susceptible to fluctuations in individual proof style. In our example, Statement 2 has a small immediate proof size (3 lines), but would be given the same statement size as Statement 1. By considering each in terms of all of its dependencies we get a much more robust measure.

E. Idealised Statement Size

Most lemmas make stronger assumptions than are actually necessary. In particular, a lemma might have a concrete term where an abstract one will suffice: the statement "1 is odd" is less general than the statement "$2n + 1$ is odd", which has an abstract term "$2n + 1$" in place of the concrete term "1". It is the job of the proof engineer to decide the appropriate level of generality for a lemma, based on its intended use. In cases where a constant with a large definition is included unnecessarily, we observe a large discrepancy between statement and proof size.

In our example, consider the statement $\neg \mathsf{C} \vee \mathsf{C}$. Suppose it was proved in one step by appealing directly to the *law of excluded middle*, an axiom of HOL in Isabelle. The raw size of this statement is $303 + 1 + 1 = 305$. This is an over-estimation of the statement's complexity because the statement mentions C unnecessarily — C could be abstracted without affecting the proof. Doing so (by replacing the constant C by a variable x) would yield a statement with size of just 2 (1 for each logical connective), a much better indication of its proof complexity.

In practice, over-specificity can save effort in provers like Isabelle, as it can aid automated reasoning by simplifying *higher order unification*, a primitive procedure in Isabelle. Additionally it is not often worth the effort to generalise a lemma that will only be used once. As a result, it is common to see many over-specific lemmas in large proof developments.

To address this, we introduce *idealised statement size*. The idealised size for the statement of some lemma l is the size it would have been given had it been stated in its most general terms. More precisely, it is defined as follows.

> **Idealised statement size for lemma l:** the raw size of the statement of l had it been abstracted over all possible constants such that l's proof remains valid.

Note that we apply this recursively, conceptually removing mentions of redundant constants in all dependant constants of l's statement. The idealised statement size of l is always smaller or equal to the raw size of l, as it may only remove unnecessary constants from measurement. Unfortunately, computing this size is undecidable as it would require a precise analysis of why l is true. We show how it can be approximated in Section IV-C.

IV. Measures in Isabelle

The definitions given in the previous section abstracted away from any specific theorem prover. Here we explain how we compute these measures for Isabelle.

A. Measuring Proof Size

The interactive proofs in Isabelle we are interested in begin with the keyword **lemma** or **theorem**, followed by a statement in Isabelle's term language. This is the *lemma statement* as described in Section III-C. This is followed by an Isabelle proof, which consists of structural proof elements and invocations of automated tools known as *proof methods*. Once the statement has been shown true, the proof ends with the keyword **done** or **qed**. This statement is now a *fact* and is usable in other proofs.

To measure proof size, we distinguish what we call Isabelle *facts* from Isabelle *lemmas*. A *lemma* is explicitly stated by the proof engineer and then explicitly proved in Isabelle. A *fact* is more general: it is a statement that has been proved by any means. This includes lemmas, but also all the statements automatically generated and internally proved by Isabelle. For example, defining a recursive function f requires a proof of its termination. In many cases this termination proof can be done automatically with no manual invocations of tools. This fact is implicitly used in any lemma l that reasons about f, but it does not have a proof size that can be measured in such a way that would correspond to the effort required to prove it (since it is automatic). Therefore, when computing the size of the proof of a lemma l, we will only count the proof sizes of used *lemmas*. Lemmas are the only facts whose proofs require substantial human effort, therefore they are the only ones relevant to our overarching goal of effort estimation.

For a given lemma l in Isabelle we say that the lines between the beginning and ending keywords (inclusive) constitute the proof of l. The immediate size of l is therefore simply its line count. Then we compute all the lemmas which l recursively depends on and add their sizes to get l's total size. Here we only count unique lemmas: if multiple dependencies of l depend on some l', we only count the size of l' once. We compute lemma dependencies by examining proof terms produced by Isabelle [22], where a proof term is the internal formal representation of a proof. The total size of l with respect to some proof development D considers all lemmas outside of D to have size 0. Simply put, for each proof development, lemmas not from that development (e.g. pre-existing library lemmas), do not contribute to the measured size of proofs.

This measure approximates the proof size of l as described in Section III-C. It is incomplete, however, as it does not include

source lines which must exist for l to be valid, but for which this dependency relationship is not easily found. For example, the proof of l may depend on certain syntax existing, declared with one line using the **notation** keyword. This is not factored into the total size of l. The impact of this on the validity of the measured proof size is assumed to be minimal, as the size of Isabelle proof developments is dominated by proof text.

B. Measuring Raw Statement Size

Simple definitions can be added to Isabelle using the **definition** command, where an equation is given in Isabelle's term language, introducing a new constant name and body. Additionally there are commands for installing (co)inductive datatypes and both partial and complete recursive functions [23]. Internally these create simple definitions based on the user specification, proving canonical facts for interpreting them. In our running example, the definition of C introduces the new name C with body $(\neg \mathsf{A}) \vee (\neg \mathsf{B})$, and a new fact **C-def** for the statement $\mathsf{C} = (\neg \mathsf{A}) \vee (\neg \mathsf{B})$.

To measure the size of a given lemma statement S as described in Section III-C we recursively inspect all definitions mentioned in S. The number of unique definitions traversed will be the size of S according to this measure.

C. Approximating Idealised Statement Size

In Section III-E we introduced idealised statement size as a refinement of raw statement size. Although computing the idealised size of S is, in general, undecidable, it can be approximated by examining the finished proof of S.

Intuitively, if S refers to C but the defining fact **C-def** of the constant C does not appear in the dependency graph of the proof of S, this means that the truth of S does not depend on the definition of C, and that S could be rewritten by replacing the constant C by a variable x. We therefore exclude the size of C when computing the approximate idealised size of S. For instance, the approximate idealised statement size of the statement $\neg \mathsf{C} \vee \mathsf{C}$, proved by appealing directly to the law of excluded middle without using **C-def**, is 2.

More generally, for each constant c, we have a set of *defining equations*. These are Isabelle facts which are the canonical interpretation of c. In the case of simple definitions, this is just the equation that was given when c was defined (such as **C-def**). To compute the idealised size of a statement S, we exclude all the constants whose defining equations are never used in the proof of S. This will always be an over-approximation of the idealised size of S, but is at worst the original statement size.

This approximation of idealised statement size cannot be a leading predictor of proof size, as it requires stage (3) of the proof to be complete. However, it is useful when trying to build an explanatory model for understanding the relationship between statement size and proof size. The implications of using this measure are discussed in Section VI

V. Data Collected

We applied our exploratory analysis to six projects: (1) four top-level statements of the seL4 verification work, and (2) two proof developments in the Archive of Formal Proofs.

A. seL4 Project Proofs

The seL4 verification project [7] produced a formal, machine-checked proof of the full functional correctness of the seL4 microkernel, down to the binary level, followed by proofs of security properties about the kernel. seL4 is a small operating system kernel, designed with explicit goals of high performance, formal verification, and secure access control. It is comprised of approximately $10,000$ lines of C code. The proof of its functional correctness shows that the binary correctly refines its high-level abstract specification. This proof was done in stages, with a proof that binary refines C, that C refines a design representation, and finally that this design representation refines the abstract specification. This last stage, called *Refine*, is our first target proof. The binary verification is not suitable for this analysis because it is not done in Isabelle [24]. The C to design verification is not suitable for technical reasons: the translation from C source to a semantic representation in Isabelle does not create a constant dependency graph like one would expect. The *Refine* proof builds on a proof, called *AInvs*, expressing global invariants satisfied by the abstract specification. This is the second target of our study. The proof of functional correctness was followed by a proof that seL4 enforces integrity [25] (called *Access*) and confidentiality [26] (called *InfoFlow*). These are the two remaining targets of our study. We present the main characteristics of these four proofs below, taken from the public release of the seL4 proofs [27]. The statements measured for each proof development are all the dependencies of its top-level statement, computed with respect to that development. The statement sizes for these proofs are computed with respect to the whole seL4 verification development, where the large kernel specifications are defined.

AInvs: this proof shows that a global invariant *invs* is preserved by every possible execution of the abstract specification. The top-level statement in this proof therefore depends on both the abstract specification and *invs*. We measured 2,790 lemmas from *AInvs*, including the top-level statement with a raw size of 1,292, ideal size of 949 and proof size of 32,214 lines (measured as described in Section IV).

Refine: this is a refinement property, showing that the design specification correctly refines the abstract specification. Its top-level statement depends on these two specifications, in addition a corresponding global invariant for each of them. *Refine* builds on *AInvs*, but we compute the proof sizes for *Refine* with respect to itself. That is, proofs from *AInvs* do not contribute to the size of proofs from *Refine*. We measured 4,143 lemmas from *Refine*, including the top-level statement with a raw size of 2,398, ideal size of 1,746 and proof size of 67,856 lines.

Access: this proof shows that seL4 preserves the integrity of components' data according to a security policy *p*, i.e. it shows that seL4 prevents unauthorised writes. Its top-level statement depends on the abstract specification, the definition of integrity as well as *invs*. Similar to *Refine*, we only measure *Access* proofs with respect to itself, taking proofs from *Refine* and *AInvs* for granted. We measured 724 lemmas from *Access*, including the top-level statement with a raw size of 1,395, ideal size of 1,083 and proof size of 8,116 lines.

InfoFlow: this proof shows that seL4 preserves the confidentiality of components' data, i.e. seL4 prevents unauthorised reads. Its top-level statement depends on the abstract specification, the

TABLE I. R^2 AND EQUATIONS FOR FIGURE 1 AND FIGURE 2

Proof Name	Size Used	R^2	Equation $f(x) = a \cdot x^2 + b \cdot x + c$		
			a	b	c
AInvs	**raw**	**0.845**	**0.02579**	**-8.181**	**394.9**
	idealised	*0.937*	*0.04325*	*-4.399*	*100.4*
Refine	**raw**	**0.724**	**0.01198**	**-5.355**	**519.1**
	idealised	*0.799*	*0.01737*	*2.365*	*<2.2E-16*
Access	**raw**	**0.735**	**0.0032**	**-1.112**	**86.45**
	idealised	*0.889*	*0.006039*	*-0.915*	*57.36*
InfoFlow	**raw**	**0.154**	**-0.0003736**	**1.743**	**<2.2E-16**
	idealised	*0.73*	*0.007893*	*-3.652*	*260.4*
JinjaThreads	**raw**	**0.457**	**0.05631**	**-16.46**	**472.9**
	idealised	*0.694*	*0.1166*	*-16.04*	*281*
SATSolver Verification	**raw**	**0.798**	**1.711**	**-65.43**	**375.7**
	idealised	*0.802*	*4.123*	*-77.52*	*223.7*

definition of integrity, and *invs*. Additionally it includes a more involved discussion of program execution traces. As previously done, proof sizes from *InfoFlow* are measured with respect to itself. We measured 1,665 lemmas from *InfoFlow*, including the top-level statement with a raw size of 2,029, ideal size of 1,323 and proof size of 19,579 lines.

B. Proofs from the AFP

The Archive of Formal Proofs (AFP) [13] is a collection of proofs in the theorem prover Isabelle, aimed at fostering the development of formal proofs and providing a place for archiving proof developments to be referred to in publications. The AFP counts over 100 entries. We investigated the two largest software verification proofs from the AFP, *JinjaThreads* [28] and *SATSolverVerification* [29].

JinjaThreads: *Jinja* is a Java-like programming language formalised in Isabelle, with a formal semantics designed to exhibit core features of the Java language architecture, and formal proof of properties such as type safety and compiler correctness. *JinjaThreads* extends this development with Java-style arrays and threads, and shows preservation of the core properties. We measured 5,215 lemmas from *JinjaThreads*, including the top-level statement with a raw size of 579, ideal size of 453 and proof size of 39,821 lines.

SATSolverVerification: this is a proof of correctness of modern SAT solvers. We measure the proof with respect to the *FunctionalImplementation* theory, an implementation of a SAT solver within Isabelle's HOL. We measured 481 lemmas from *SATSolverVerification*, including the top-level statement with a raw size of 99, ideal size of 57 and proof size of 21,788 lines.

VI. RESULTS AND DISCUSSION

For each of the six projects analysed, we computed the statement size and proof size of all of its lemmas, using the measures described in Section IV. As explained, we used two variants for the statement size: a raw measure and an idealised measure, where the latter represents what the size of the statement would be if stated in its most general form.

The results of the analysis are given in Figure 1 for the raw measure and in Figure 2 for the idealised measure. The analysis demonstrates a *consistent quadratic relationship* between statement size and proof size across all the projects, with a stronger relationship when using the idealised measure. The respective R^2 and equations of the regression lines are given in Table I. We now discuss the results in detail.

A. Results using the Raw Measure

For the first three results the regression for the raw size fits the data nicely, as we can see in the plots and confirmed by the R^2 results. However, even in *Access* some outliers can be seen in the lower right corner of the graph. In *InfoFlow* this effect is even stronger: a cluster of points with a large statement size ($\sim 2,000$) and a negligible proof size. This has the effect of flattening the regression line and obfuscating the relationship. After a thorough examination we determined that these are statements that have been over-specified (see Section III-E), resulting in a larger statement size than expected.

There likely exist many other over-specified lemmas, but they have a less apparent effect on the overall shape of the graph. The lemmas identified in this case were pathological: they mentioned the entire abstract specification but were stating a general property. The abstract specification is one of the largest constants in this development, so including it in a statement makes its size completely dominated by the size of the abstract specification. All of the lemmas in *AInvs* are similarly over-specified, but more subtly so. Indeed the presence of this over-specification was only made clear after performing the idealised size analysis. This over-specification is a result of abstract specification being *extensible* [30], embedding an optional deterministic implementation of certain operations, which can be used in place of non-deterministic ones when necessary. However, no proof in *AInvs* appeals to this deterministic implementation. This is simply because, by design, the standard invariants do not discuss the program state required to resolve this determinism. As a result, these deterministic operations unnecessarily inflate the statement size as measured in *AInvs*.

B. Effectiveness of the Idealised Measure

The idealised size, introduced in Section III-E, is meant to capture the idea that redundant constants do not contribute to the difficulty of a proof. Analysis using the idealised size (shown in Figure 2 and Table I) show improved results across all projects. In particular, the *InfoFlow* results are now much more aligned and consistent with the others. R^2 now varies from 0.694 to 0.937. We can also see from the graphs that all of the statement sizes shrunk, reflecting the fact that the idealised size is always smaller or equal to the raw size.

We made no attempt to investigate the outliers in the proofs from the AFP, simply because we do not know their proofs well enough to perform the same level of analysis. Despite this, applying the idealised size measure had significant improvement on the clarity of the data in *JinjaThreads*, with a lesser effect in *SATSolverVerification*, which was already quite clear.

Previously [8] we showed a linear relationship between the size of formal specifications (as measured in source lines of Isabelle) and the number of lines of source code. Although not measured, we argue that the specification size (as described in Section III-D) will scale linearly with source lines. Combined with the result above, this indicates a quadratic relationship between code size, property size, and eventual proof size. This confirms our intuition based on our experience developing the seL4 proofs: the correctness of each property depends on its interaction with the entire program, resulting in the observed quadratic relationship. Despite this intuition, and the apparent shape of the data, we investigated other regressions in the

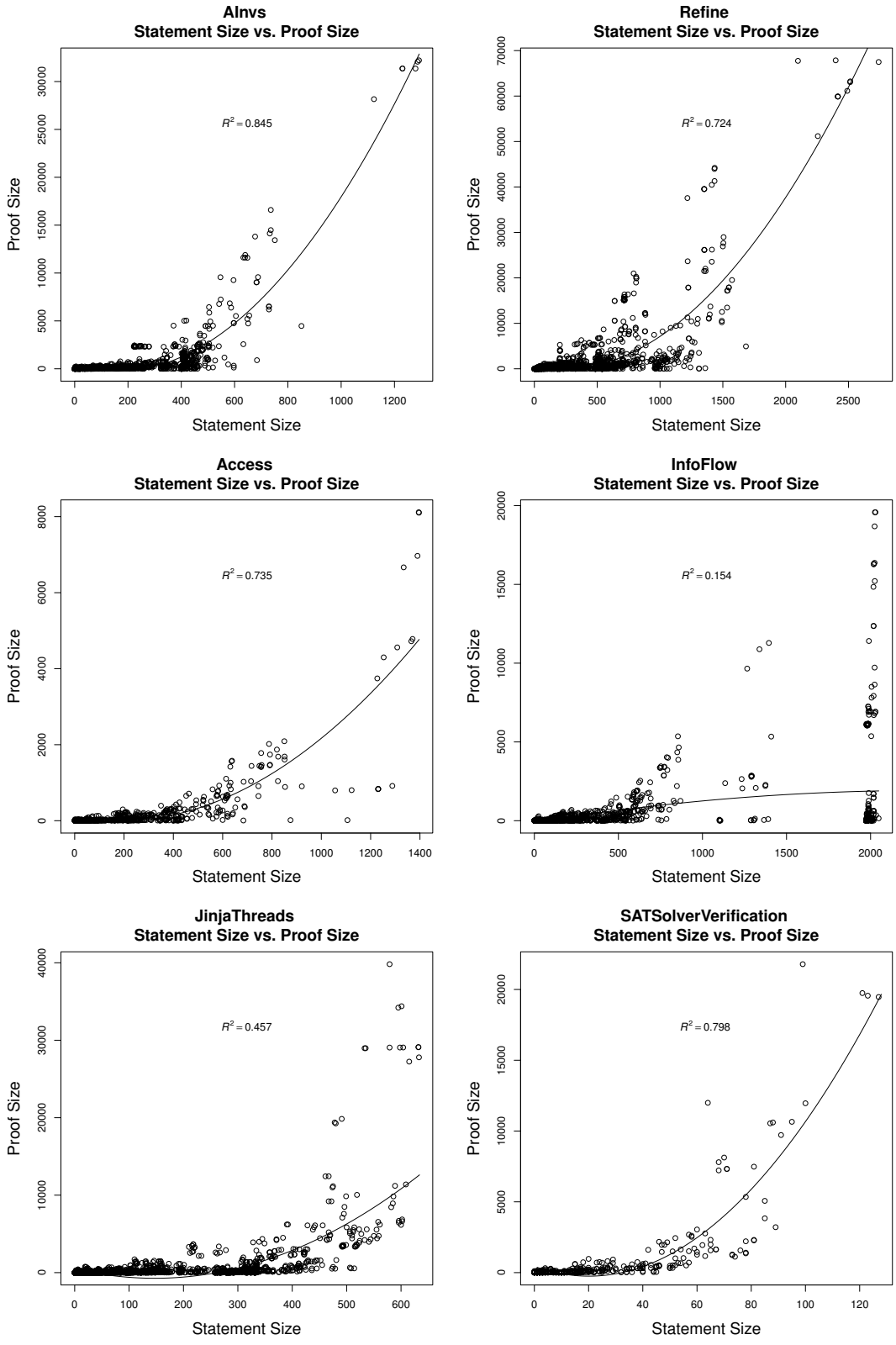

Fig. 1. Relation between raw statement size and proof size (measured as described in Section IV) for the six projects analysed.

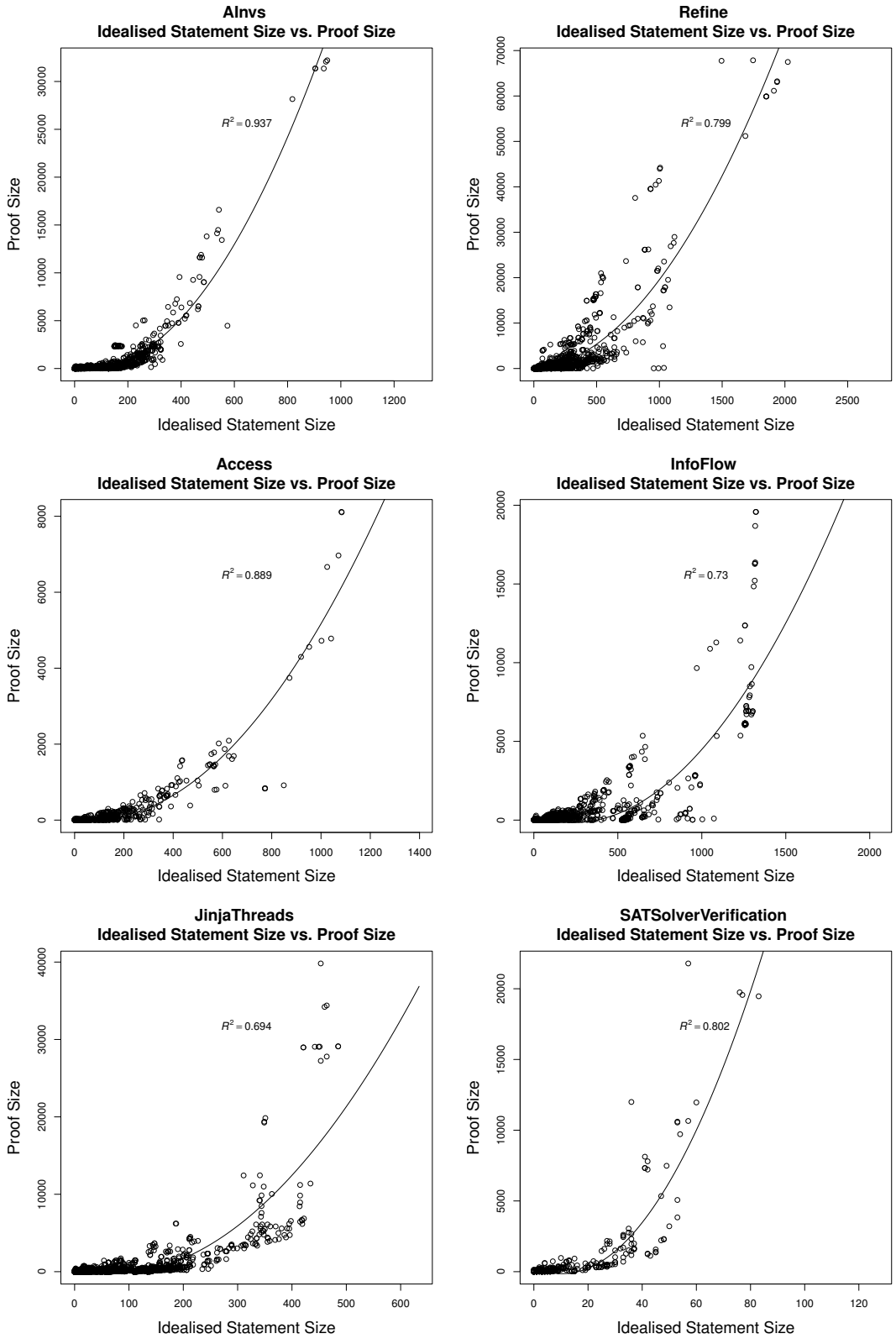

Fig. 2. Relation between idealised statement size and proof size (measured as described in Section IV) for the six projects analysed.

course of this study. Specifically we performed linear, cubic and exponential regressions against the measured proof size and ideal/raw specification sizes. Linear and exponential regressions were less compelling than quadratic, with extremely low R^2 values and clearly not fitting the data. A cubic regression yielded a marginal increase in R^2, but with an extremely small leading coefficient, indicating that the relationship is indeed quadratic.

Overall, our measures of statement size, while primitive, are a promising leading indicator of proof size, with a quadratic correlation that is strengthened with the idealised measure.

C. Limitations

More work can still be done to refine the measures, tune them to specific application areas, and further analyse outliers. In particular, a set of outliers that can be seen on the graphs is the data points with very low proof size for large statement size, visible for instance in the graph for *InfoFlow* in Figure 1 and Figure 2 as the points that drag down the regression line. These points are a result of proof reuse: they correspond to statements proved largely by just using facts from *Access*; the size of these *Access* facts is not counted since they come from a development already in existence when starting the *InfoFlow* proofs (see Section III-C). This re-use could be captured by better measures, as discussed in Section VIII.

Although the relationship between proof and statement is consistently quadratic across the proof developments we analyzed, it is important to note that it is not yet necessarily predictive. It is not clear how to take a model from one proof and use it to predict proof sizes in another. The seL4 proofs all have quadratic coefficients within the same order of magnitude, indicating some potential for an overall model for seL4-style proofs. However the AFP results are quite different: The largest statements in *JinjaThreads* and *SATSolverVerification* have raw sizes of 579 and 99 respectively, which are both an order of magnitude smaller than any top-level statement in seL4. Their proof sizes, however, are comparable to that of *Refine* and *AInvs*. Without an in-depth analysis of these proofs it is difficult to confidently say what causes this discrepancy.

D. Other Measures

The measure for statement size was chosen based on its simplicity and because it was clear how an idealisation could be applied in order to address over-specified lemma statements. In the course of this work we investigated many other measures which attempted to capture our intuition about what makes a statement difficult to prove. For example, we gave extra weight to recursive functions with even greater weighting for mutual recursion, to capture the intuition that inductive proofs are inherently difficult. In practice however, none of these measures correlated better than the naive one. There is still significant room for exploration in this area, and in Section VIII we discuss measures which might yield a predictive model.

VII. Threats to Validity

Our results show a strong internal consistency amongst the projects analysed, whose validity is strengthened by two factors. Firstly the analysis of the data collected has been set-up by one individual and re-run, refined and checked by another. Secondly, the derivation of measures has been mechanised and fully automated using a custom tool, ProofCount [27]. Given the size of the data set, manual counting would have been impossible. ProofCount gives us the guarantee of determinism and ease of use on any formal proof in the same language. However, it is quite intricate and could still contain errors.

A threat to internal validity is the simplifying assumption made in Section III that proofs proceed in three neat phases: (1) definition writing, (2) lemma statement writing, and (3) proof writing. In practice, these phases tend to be iterated to varying degrees. For instance, during phase (3) one often finds a lemma unprovable, meaning that definitions and lemma statements written earlier need further refinement and existing proofs written may need to be updated. The noise introduced by this iteration would need to be taken into account when building an estimation model based on the results from this paper, and is inherent when using results obtained from *finished* proofs (which have already undergone the iteration mentioned above). Any such predictive model will therefore require experimental validation on *new* proofs, by running it at the beginning of phase (3) and comparing its predictions to the actual results obtained at the end of the proof.

An additional threat to internal validity is the justification of our idealised measure for statement size, as a refinement of raw statement size. This measure is shown to have stronger relationship to proof size across all projects. The threat comes from the fact that we use information from the proof to identify overly-specific statements. This makes the application of idealised size as a leading measure less straightforward. In practice however, when attempting to estimate the eventual size of the proof of a statement, we would apply domain-specific knowledge to manually identify which parts might be over-specified. In most cases the worst causes of over-specification are readily apparent, but require human intuition to identify. The idealised size, as we computed, is an automated substitute for the application of this intuition. With some manual effort, applying the expertise of the proof engineer to identify cases of over-specification, an approximation of idealised size can be made without requiring a finished proof. This approximation could therefore be applicable as a leading measure.

Threats to external validity include the ability to generalize the results to other verification projects, other project settings (e.g. team size), and other interactive theorem provers. In order to address the first two threats, we have applied our approach to three completely separate verification projects, including two independent open-source projects from the AFP. Results show that our raw measure for statement size does not have high correlation to proof size for *JinjaThreads*. However, our idealised measure does have a strong quadratic relationship in both AFP proofs, consistent with the results for the seL4 verification project. Application to additional projects from the AFP could also provide further data points, however we are most interested in projects with large top-level statements, as is seen in formal software verification. The largest proof in the AFP, apart from *JinjaThreads*, only has a top-level proof size which is comparable to the smallest proof development we examined.

The last threat, that of having applied our approach to only one interactive theorem prover, namely Isabelle, has been partially addressed by choosing measures whose general principles are prover-agnostic (e.g. number of constants, number

of proof lines) and that have equivalents in other popular theorem provers such as Coq. However, a notable difference between Coq and Isabelle is Coq's native support for easily writing tactics [31] (proof procedures). In Isabelle most proofs appeal solely to built-in automation, with only recent work [32] aiming to make writing custom automation more accessible. Investigating the effect of using custom proof tools on the observed relationship is planned future work.

To summarise, the generalisation to other settings is yet to be investigated but the approach has been designed to be generalisable and initial steps have been taken to validate it on different types of projects.

VIII. Future Work

Although we have only presented simple measures in this paper, we have a large degree of freedom in this area. Previously [11] we chose to measure proof size as lines of proof because it correlated well with effort, but only at the granularity of entire proof developments. A more in-depth analysis could establish a relationship between some size measure and effort for individual lemmas. For instance, we might try to incorporate some semantic information from proofs to calculate their size. The source text of a proof describes invocations to automated reasoning tools, which eventually must appeal to primitive inference rules. These appeals can be stored as a formal representation of the proof, often called a *proof term* [22], [33]. A naive size measure of a proof term is unlikely to be indicative of the effort required to create it. For example, a single invocation of a first-order prover might produce a large proof term, but would be a trivial exercise from the perspective of a proof engineer. Despite this, we anticipate that it would be possible to incorporate more semantic proof information in proof size measurement.

Grov et al. suggest that high-level proof strategies could be extracted from finished proofs, using statistical relational learning [34]. This was realised as ML4PG [35], an extension to Proof General [36], which gathers on-line statistics during the development of Coq proofs and uses machine learning to provide hints when writing subsequent proofs. They extract proof features such as *goal shape* at intermediate proof points as well as which automated tools were applied. A similar analysis could be done to more precisely measure effort for individual proofs and provide a more accurate measure of proof size.

It is difficult to pick a good single measure of statement size given their complexity. In general we could compute a *feature vector* which captures more fine-grained information from statements. One feature could be statement size as measured in this paper. Other features could include domain-specific information, such as the size of function and invariant statements separately, or the cyclomatic complexity of the programs. We could also attempt to capture information from previously finished proof developments that we build on. For example, as a feature of a statement, we might compute some measure of its similarity to statements of existing lemmas. This would allow us to anticipate the degree of proof re-use (as mentioned in Section VI-C) that would potentially appear in its proof. The set of features chosen would dictate the class of proofs for which such an analysis is meaningful (e.g. program verification). We hypothesise that well-chosen feature vectors would be robust against the fluctuations in statement design style that we see e.g. between seL4 and JinjaThreads.

Although our analysis did not result in a universal predictive model for eventual proof size, it does indicate the potential for iteratively building a project-specific model due to the strong correlation within a given proof development. The model could be updated as proofs are completed and the precise relationship between proof and specification size is better understood.

IX. Conclusion

In this paper we investigated the relationship between the size of *formal statements* and corresponding *formal proofs* in an interactive theorem prover. Our increasing trust on critical software systems has created a demand for software which is trustworthy. Formal software verification is the application of formal proof to guarantee the correctness of software behaviour. However, the cost of formal verification is not well understood due to the relatively low number of successful large-scale applications. The ability to accurately predict this cost is crucial to the wide-spread application of formal methods. In earlier work [11] we established a linear relationship between proof size and proof effort. Expanding on this, we sought leading indicators of proof size and thus turned to statement size.

We have established two measures for statement size, *raw* and *idealised*, and use them in an analysis of six Isabelle proof developments. Raw size is the number of unique constants required to write a statement, recursively including all dependencies. This measure was shown to be highly susceptible to over-specified statements having inflated sizes. This prompted the introduction of idealised size, a refinement of raw size, which removes redundant constants in order to reduce the impact of over-specified statements. In total we examined the size of 15,018 statements and compared them against their proof size.

Our analysis shows a quadratic relationship between statement and proof size, and that our idealised measure strengthens this correlation. However, the investigation did not yield a predictive model as the precise relationship changes between proofs. Combined with previous work [11], we believe this is an important step in establishing a leading indicator for the effort and cost required to perform formal verification.

Acknowledgment

The authors would like to thank Ihor Kuz, Gernot Heiser and Thomas Sewell for their feedback on drafts of this paper.

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

REFERENCES

[1] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive experimental analyses of automotive attack surfaces," in *Proceedings of the 20th USENIX Security Symposium*, 2011.

[2] D. Halperin, S. S. Clark, K. Fu, T. S. Heydt-Benjamin, B. Defend, T. Kohno, B. Ransford, W. Morgan, and W. H. Maisel, "Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses," in *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2008, pp. 129–142.

[3] K. Fisher, "HACMS: High assurance cyber military systems," in *Proceedings of the 2012 ACM Conference on High Integrity Language Technology*, ser. HILT '12. Boston, Massachusetts, USA: ACM, 2012, pp. 51–52.

[4] (2012) Common criteria for information technology security evaluation, version 3.1 revision 4. [Online]. Available: http://www.commoncriteriaportal.org/cc/

[5] Special Committee of RTCA, "DO-178C, software considerations in airborne systems and equipment certification," 2011.

[6] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[7] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM Transactions on Computer Systems*, vol. 32, no. 1, pp. 2:1–2:70, Feb. 2014.

[8] M. Staples, R. Kolanski, G. Klein, C. Lewis, J. Andronick, T. Murray, R. Jeffery, and L. Bass, "Formal specifications better than function points for code sizing," in *International Conference on Software Engineering*, David Notkin, Betty H. C. Cheng, Klaus Pohl, Ed. San Francisco, USA: IEEE, May 2013, Conference Paper, pp. 1257–1260.

[9] R. Jeffery, M. Staples, J. Andronick, G. Klein, and T. Murray, "An empirical research agenda or understanding formal methods productivity," *Information and Software Technology*, 2015, to appear.

[10] D. C. Stidolph and J. Whitehead, "Managerial issues for the consideration and use of formal methods," in *In Stefania Gnesi, Keijiro Araki, and Dino Mandrioli (eds.), FME 2003, International Symposium of Formal Methods Europe*, 2003, pp. 8–14.

[11] M. Staples, R. Jeffery, J. Andronick, T. Murray, G. Klein, and R. Kolanski, "Productivity for proof engineering," in *Empirical Software Engineering and Measurement*, Turin, Italy, Sep. 2014, Conference Paper.

[12] T. Nipkow, L. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2283.

[13] G. Klein, T. Nipkow, and L. Paulson, "The archive of formal proofs," http://afp.sf.net, 2003.

[14] M. Olszewska (Plska) and K. Sere, "Specification metrics for Event-B developments," in *Proceedings of the CONQUEST 2010: "Software Quality Improvement"*, I. Schieferdecker, R. Seidl, and S. Goericke, Eds. International Software Quality Institute, 2010, p. 112.

[15] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977.

[16] Deploy: Industrial deployment of system engineering methods providing high dependability and productivity. [Online]. Available: http://www.deploy-project.eu/

[17] W. B. Samson, D. G. Nevill, and P. I. Dugard, "Predictive software metrics based on a formal specification," *Inf. Softw. Technol.*, vol. 29, no. 5, pp. 242–248, Jun. 1987.

[18] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[19] A. Tabareh, "Predictive Software Measures Based on Formal Z Specifications," Master's thesis, University of Gothenburg - Department of Computer Science and Engineering, 2011.

[20] A. Bollin, "Metrics for quantifying evolutionary changes in Z specifications," *Journal of Software: Evolution and Process*, vol. 25, no. 9, pp. 1027–1059, 2013.

[21] S. King, J. Hammond, R. Chapman, and A. Pryor, "Is proof more cost-effective than testing?" *IEEE Trans. Softw. Eng.*, vol. 26, no. 8, pp. 675–686, Aug. 2000.

[22] S. Berghofer and T. Nipkow, "Proof terms for simply typed higher order logic," in *Theorem Proving in Higher Order Logics*, ser. Lecture Notes in Computer Science, M. Aagaard and J. Harrison, Eds. Springer Berlin Heidelberg, 2000, vol. 1869, pp. 38–52.

[23] S. Berghofer and M. Wenzel, "Inductive datatypes in HOL - lessons learned in formal-logic engineering," in *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, ser. TPHOLs '99. London, UK, UK: Springer-Verlag, 1999, pp. 19–36.

[24] T. Sewell, M. Myreen, and G. Klein, "Translation validation for a verified OS kernel," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*. Seattle, Washington, USA: ACM, Jun. 2013, Conference Paper, pp. 471–481.

[25] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein, "seL4 enforces integrity," in *Proceedings of the 2nd International Conference on Interactive Theorem Proving*, ser. Lecture Notes in Computer Science, M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, Eds., vol. 6898. Nijmegen, The Netherlands: Springer, Aug. 2011, pp. 325–340.

[26] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: from general purpose to a proof of information flow enforcement," in *IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2013, Conference Paper, pp. 415–429.

[27] Trustworthy Systems Team, "seL4 proofs for API 1.03, release 2014-08-10," Aug 2014.

[28] A. Lochbihler, "Jinja with threads," *Archive of Formal Proofs*, Dec. 2007, http://afp.sf.net/entries/JinjaThreads.shtml, Formal proof development.

[29] F. Maric, "Formal verification of modern SAT solvers," *Archive of Formal Proofs*, Jul. 2008, http://afp.sf.net/entries/SATSolverVerification.shtml, Formal proof development.

[30] D. Matichuk and T. Murray, "Extensible specifications for automatic re-use of specifications and proofs," in *10th International Conference on Software Engineering and Formal Methods*, Thessaloniki, Greece, Dec. 2012, Conference Paper, p. 8.

[31] D. Delahaye, "A tactic language for the system Coq," in *International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, ser. Lecture Notes in Computer Science, vol. 1955. Springer, Nov. 2000.

[32] D. Matichuk, M. Wenzel, and T. Murray, "An isabelle proof method language," in *Interactive Theorem Proving (ITP)*, Vienna, Austria, Jul. 2014, Conference Paper, p. 16.

[33] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[34] G. Grov, E. Komendantskaya, and A. Bundy, "A statistical relational learning challenge — extracting proof strategies from exemplar proofs," in *ICML12 Workshop on Statistical Relational Learning (SRL-2012)*, 2012.

[35] E. Komendantskaya, J. Heras, and G. Grov, "Machine learning in proof general: Interfacing interfaces," in *Proceedings 10th International Workshop On User Interfaces for Theorem Provers, UITP 2012, Bremen, Germany, July 11th, 2012.*, 2013, pp. 15–41.

[36] D. Aspinall, "Proof general: A generic tool for proof development," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2000, pp. 38–43.