Code to Qed, the Project Manager's Guide to Proof Engineering

Nicolas Dejon, Chrystel Gaber, Gilles Grimaud, Narjes Jomaa

Code to Qed, the Project Manager's Guide to Proof Engineering

NICOLAS DEJON* and CHRYSTEL GABER, Orange Innovation, France
GILLES GRIMAUD and NARJES JOMAA, Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL - Centre de Recherche en Informatique, Signal et Automatique de Lille - UMR 9189, France

Despite growing efforts and encouraging successes in the last decades, fully formally-verified projects are still rare in the industrial landscape. The industry often lacks the tools and methodologies to efficiently scale the proof development process. In this work, we give a comprehensible overview of the proof development process for proof developers and project managers. The goal is to support proof developers by rationalizing the proof development process, which currently relies heavily on their intuition and expertise, and by facilitating communication with the management line. To this end, we concentrate on the aspect of proof manufacturing and highlight the most significant sources of proof effort. We propose means to mitigate the latter through proof practices (proof structuring, proof strategies, and proof planning), proof metrics, and tools. Our approach is project-agnostic, independent of specific proof expertise, and computed estimations do not assume prior similar developments. We evaluate our guidelines using a separation kernel undergoing formal verification, driving the proof process in an optimised way. Feedback from a project manager unfamiliar with proof development confirms the benefits of detailed planning of the proof development steps, clear progress communication to the hierarchy line, and alignment with established practices in the software industry.

CCS Concepts: • **Software and its engineering → Formal software verification**; **Software development process management**; *Software design engineering*; *Reusability*.

Additional Key Words and Phrases: proof development, proof strategy, proof metrics, industrial development, project management

ACM Reference Format:
Nicolas Dejon, Chrystel Gaber, Gilles Grimaud, and Narjes Jomaa. 2024. Code to Qed, the Project Manager's Guide to Proof Engineering. 1, 1 (May 2024), 50 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In the realm of software development, an informal specification serves as a critical foundation for grasping the desired functionality and behaviour of the software. However, the inherent ambiguity of informal specifications can lead to misinterpretations between the author of the specification and the developer implementing the software. As such, despite their utility, informal specifications can sometimes be understood in multiple ways, underlining the need for rigorous formal methods to ensure unambiguous and precise interpretation of software requirements. To demonstrate high-assurance, one can establish guarantees through mathematical proofs derived from logical deduction. Although computers assist in machine-checking proofs, humans still have a preponderant role in leading and managing the proof development process, which includes writing proofs, developing strategies, optimizing resources and planning a long-lasting effort. In a context where strong guarantees are increasingly demanded, such as in the military, the avionics and the medical fields [6], industry actors without specialized expertise are showing a growing interest in

*Corresponding author.

Authors' Contact Information: Nicolas Dejon, nicolas.dejon@orange.com; Chrystel Gaber, chrystel.gaber@orange.com, Orange Innovation, Châtillon, Ile-de-France, France; Gilles Grimaud, gilles.grimaud@univ-lille.fr; Narjes Jomaa, jomaa.narjes@gmail.com, Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL - Centre de Recherche en Informatique, Signal et Automatique de Lille - UMR 9189, Villeneuve d'Ascq, Hauts-de-France, France.

Manuscript submitted to ACM

partially or fully formally verifying their systems. But the formal verification process has long been recognized as challenging, particularly for large software systems as summarized by De Millo *et al.* in 1979 [14]: *the absence of continuity, the inevitability of change, and the complexity of specification of significantly many real programs make the formal verification process difficult to justify and manage.* Since then, programs and systems have scaled up and new software management methodologies like agile practices [2] have emerged. As formal verification sees wider adoption, surrounded by more complex software, it is essential to support the process. To this aim, several works investigated ways to maintain [36, 37, 42], extend [17] and automate proofs of correctness [39], in parallel to continuous improvements on tools used for formal verification. Nonetheless, leading projects in the last decades, such as seL4 [28] and CertiKOS [18], have reported substantial costs and human effort in the proof development process, showing room for progress. Furthermore, the proofs are heavily reliant on the expertise of proof developers. The articles [8, 13, 31, 40] highlight that practices in Proof Engineering are not as mature as in Software Engineering and they call for progress in the area of proof understanding, proof guidance and conduct, and relevant proof metrics, for better formal verification project management, especially when starting a new project with non-specialised experts in formal verification. Indeed, like regular software development, proofs must cope with daily developments, requirements changes, unexpected tasks, resource attribution, turnovers, priority changes, and non-expert individuals (*e.g.* project managers and hierarchical line). As Andronick *et al.* asserted in [8], the management of projects involving program proof is in dire need of further study and control. Specifically, the authors assert that "*Better metrics are required. Related to this, better estimation models are required for formal verification projects. These are both open questions. Answering them will help to bring about decision-making tools for formal verification project management,[...]*". These are still open questions nowadays [7]. The observed gap in the literature is that managers of proof development teams require clear visibility and understanding of the proof process along with quantitative measurement progress upon the inception of the project, independently of past experiences, and should account for the project's uncertainties. This will enable them to provide better support to proof developers in prioritising their tasks and meeting expected deliveries. To this end, this work provides guidance to non-expert managers in charge of a project undergoing formal verification, by bridging the gap between proof engineering techniques associated with interactive theorem proving and software development management, and by proposing several quantitative metrics to finely track and report the proof progress.

This paper presents a proof development framework tailored for software projects, that is responsive to changes, is aware of the proof effort, is agnostic to the proof assistant, aids proof engineers in prioritizing their tasks in the best effort, and provides visibility on the proof's completion with quantitative metrics and visual representations suitable to managers also used to adjust and optimize the proof strategy. The provided methodology enables rapid identification of the scope of properties that need to be proven and then the establishment of an optimised proof path. The main benefit of the approach is to limit the scope creep risks (*i.e.* the risk of discovering new properties or reformulating properties at a late stage of the project). This has a direct impact on the proof effort overhead as described by Bourke *et al.* [13]: "*the faster the discovery, the smaller the overhead*". The result is a set of metrics and practices from the beginning to the end of the formal verification approach which can be used to drive the formal proof process and communicate its progress to non-acculturated stakeholders. This paper primarily targets teams with knowledge in Proof Engineering who start working on medium-scale long-term software proof projects rather than proof engineers specialised and dedicated to proofs in one specific domain.

In particular, our propositions are inspired by Software Engineering to build on the experience of proof developers and managers already acquainted with these practices. Our contributions are as follows:

- A description of the proof construction process and the identification of the major sources of the proof workload;
- The presentation of our proof/code co-design approach;
- The introduction of a proof management framework composed of proof structuring, proof strategies and proof planning methods, with associated tools and metrics;
- The illustration of the framework usage on a medium-scale practical software project with a formal verification approach;
- The use of these practices and metrics to build an agile project management dashboard.

This article is rooted in the context of a long-term, multi-year, collaboration between an academic laboratory with expertise in formal verification and an industrial research centre proficient in agile project management for software development but no slightest competency in project management for software proofs. The joint works started with a European Celtic-Plus collaborative project which led to the creation of Pip [26] and continued with the advisory of a PhD student which led to the creation of Pip-MPU [15], an adaptation of Pip for constrained microcontrollers. From the start of the collaboration, we observed a gap in the project management methodologies. Specifically, the progress and difficulties related to the proof were difficult to convey and led to an impression of a tunnel effect where it was impossible to strategically drive the project and to report to stakeholders not acculturated to formal proofs. This created tensions with other partners of the EU-funded collaborative project. The second encountered difficulty was related to the fact that the teams specialised in proof engineering were involved in a wide span of projects not necessarily related to the formal proofs of Operating Systems. Added to the fact that team members come and go, it is difficult to maintain a team which has a very deep understanding of the desired properties and behaviours, which is paramount to developing the intuitions and informal processes for medium to large-scale proof projects. This context largely differs from the teams involved in companies such as Proofcraft [33] or ProvenRun [34] which are deeply specialized in formal proofs for Operating Systems and dedicate their effort towards a limited number of products and applications.

The practices proposed in this paper lead the proof development process with the goal of planning the project's milestones to reach full formal verification with increased added value in a systematic way, with little reliance on the intuition of proof experts. Particularly, we deal with the aspect of unexpected changes in the project scope. This is usually eluded by estimation models computed *a posteriori*, not because they do not appear, but because the computation is based on past experiences and they rely on proof experts in similar projects to drive the proofs. However, we consider this aspect as central because they are sources of non-negligible proof effort drifting that could explode project costs. Hence, we *do not* assume proof experts already experienced in this specific proof conduct, nor the existence of similar past projects. The methods described in this paper result from internal projects targeting the formal verification of invariant security properties implemented within the Coq Proof Assistant [22] and represent the learnings from nearly 200.000 lines of proof.

The rest of this document is structured as follows. Section 2 presents underlying background information about two of our team's projects which underwent formal verification with associated knowledge around formal verification. In Section 3, we present our way of reasoning to prove a code block or instruction from start to end (from **Code to Qed**). Subsequently, we detail our proof/soft co-design approach specifically tailored for this proof process. In Section 4, we introduce proof management practices to reduce the sources of extensive proof workload identified in the previous section. We provide metrics, tools, and strategies to control the proof effort and to master the proof development process. Section 5 illustrates the practices introduced earlier for the formal verification of the security properties for an Operating System (OS) kernel. The project management dashboard is presented and its use is illustrated in Section 6.

The document ends with related works 7 and a conclusion 8. All presented elements are discussed in the sections they are introduced in.

2 BACKGROUND

The projects underlying this work aim to achieve full formal verification of the security properties of OS services. This section introduces the concepts of correctness and formal verification in the context of software development and presents tools to conduct formal proofs. It also introduces Pip and Pip-MPU, two practical software projects that demonstrate strong guarantees through formal verification of security properties, which are used to illustrate the work. Then, the section explains the use of Hoare Logic in formally verifying the kernel services of Pip and Pip-MPU. Lastly, the section describes the tracking of software development project progress using Burnup and Burndown charts.

2.1 Correctness and formal verification

Correctness is a critical aspect of software development, ensuring that a software system adheres to its formal specification and satisfies the intended relationship between inputs and outputs. The notion of correctness encompasses an infinite set of properties derived from the specification, rather than relying on a predefined set as in the case of static analysis. These properties can be broadly categorized into functional and non-functional properties. Functional properties pertain to the expected outcomes of software execution, such as producing a correct result (*e.g.*, returning an approximation of π). On the other hand, non-functional properties relate to conditions that should hold true regardless of the software's specific output or behaviour (*e.g.*, ensuring the software terminates its execution before a specified deadline). By rigorously evaluating these functional and non-functional properties, developers can build confidence in the correctness of their software.

The correctness of the implementation, or the satisfiability of the specification, is demonstrated through methods of different rigour. One of them is software testing, but as a software's state space is combinatorial, correctness is only demonstrated on singular events and a typical code coverage of 95% is considered high-quality in production environments. This masks the rest of the reachable states that can cause troubles, especially concerning security and safety. Formal methods are on the far end of the most rigorous methods. They are the only way to reach strong guarantees of the system's properties by supplying a mathematical proof of correctness. In other words, they replace the demonstration of confidence from testing singular events by universal statements. These are conducted on a mathematical model where proven properties reflect those in the real system. The model is always a simplification of the real system but must be as close as possible to real behaviours.

Formal verification is a process where the goal is to demonstrate correctness using formal methods. Verification can consist in verifying invariants that are properties that should always be satisfied despite transformations of the mathematical objects they refer to. For example, a loop invariant is a logical condition that is true at the start of the loop and still true at the end of each iteration of the loop. To facilitate the proof conduct, proof developers rely on theorems and lemmas.

Formal specifications and implementations also play a vital role in identifying the **chain of responsibility** in the event of a software malfunction. Proofs, and by extension program proofs, demonstrate that if the premises (assumptions) are correct, then the theorem (the result of the reasoning) is also valid. The notion of responsibility is distinct from specification, as specifications primarily aid in the software development process. Responsibilities, on the other hand, are crucial for defining and managing the software's operating environment to ensure it satisfies specification requirements. This may involve making assumptions about the hardware, where the relationship between hardware and software

behaviours is defined, or considering the libraries being used, with developers providing these libraries tasked with guaranteeing the validity of the underlying assumptions. By carefully delineating responsibilities and assumptions, developers can better understand and address any potential issues, enhancing the overall reliability and robustness of the software system. This clear allocation of responsibility facilitates swift identification and resolution of problems.

2.2 Formal verification tools

In the area of computer-assisted proofs, tools can be categorized in a spectrum ranging from *interactive* (requiring substantial human manual intervention) to *automated* (no or limited human intervention), with hybrid systems lying in between.

Common examples of automated tools are model checkers (*e.g.* TLA+, Spin, or Alloy [1, 3, 20]), some of them connected to SAT/SMT solvers. However, proofs generated by fully automated tools are typically very large, domain-specific, and subject to resource exhaustion, as observed in the state explosion problem which usually over-approximates the results [35]. Some automated tools tend to enlarge their capacity though (*e.g.* Dafny [29]), but tools suitable to express complex high-level properties, like security properties, delivered (mostly) in human-readable proofs, are usually interactive. Examples of interactive tools are interactive theorem provers, also called *proof assistants* (*e.g.* the Coq Proof Assistant [22], or Isabelle [5]). They can be used independently, as part of a verified toolchain (*e.g.* the Verified Software Toolchain [9]) or embedded in automated platforms to discharge verification conditions (*e.g.* Why3 [4]). They work by formal deduction (or deductive verification).

This study focuses on formal verification of a system using a proof assistant, which consists of writing the specification of the properties to prove, and then proving that the specification is satisfied, with regular human inputs to drive the proof. Hence, they require intense manual guidance and prover expertise, so they are time and resource-consuming, leading to heavy **proof effort**.

In formally verified projects, the proof effort is usually given in person-time (days, months, years). In what follows, we take the definition:

Definition 1 (Proof effort). The proof effort gives a measure of the amount of work (human effort) necessary to complete a proof.

For example, the proof effort can be instantiated to compute statistics on formal verification projects, providing insights such as a ratio of 0.0066 person-month per line of code for the formal proofs of security properties in seL4 [27] and a ratio of 0.0036 person-month per line of code for the ongoing formal verification of the security properties in Pip-MPU [16].

2.3 Pip and Pip-MPU

Pip and **Pip-MPU** [15, 26] are high-assurance systems, demonstrating strong guarantees through hardware-rooted mechanisms supported by formal verification, *i.e.* with mathematical foundations. They are two distinctive minimalist separation kernels, *i.e.* they simulate a physically distributed environment on a single machine [38]. Indeed, they both provide a hierarchical partitioning tree with strict memory space isolation, rooted in the root partition, as illustrated in Figure 1. That is, they ensure strict spatial memory partitioning (confidentiality and data integrity) between partitions (executable components) guarded by hardware mechanisms.

Pip-MPU derives from Pip, but with complete code refactoring because of a different hardware platform: Pip targets high-end devices and leverages the *Memory Management Unit (MMU)*, whereas Pip-MPU leverages the *Memory Protection*

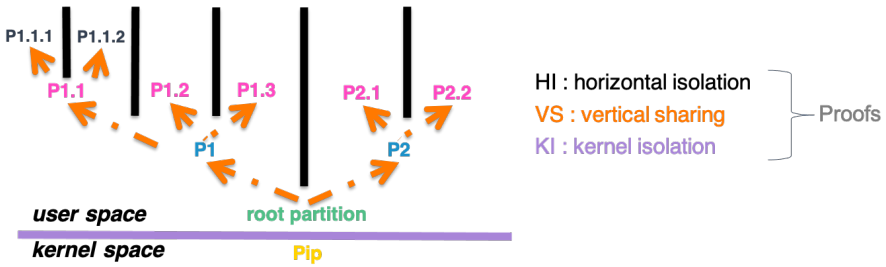

Fig. 1. Pip's hierarchical partitioning model.

Unit (MPU) that is adapted to constrained devices (scarce memory and power resources). Both allow the definition of protected memory regions but for different types of devices.

The formal verification process aims to verify the hierarchical partitioning model of the separation kernel, by verifying the so-called **security properties**. These properties are composed of one memory-sharing property (Vertical Sharing VS ensuring memory owned by a child partition must be shared from its parent partition) and two isolation properties ensuring strict spatial partitioning (Horizontal Isolation HI ensuring strict isolation between sibling partitions and Kernel Isolation KI ensuring strict isolation between user and kernel memory).

While Pip and Pip-MPU share the same proof goals of verifying the security properties, the code difference forced complete refactoring of the proofs. The proof workflow is nevertheless kept, mirroring the project structure and naming conventions. In both projects, code and proofs are written within the Coq Proof Assistant, which ensures the correctness of the proofs and enforces the utilization of well-formed types and structures. The code is written in monadic style and subsequently transpiled into equivalent ComperCert C language, which is then compiled. This approach enables the construction and utilization of data structures based on integers and pointers, such as integer sequences or complex data types, within the proof assistant. Multiple other projects use the *refinement* proof technique, which adds multiple abstraction levels to verify complex properties on the most abstracted level and preserve them down, like seL4, ProvenCore, and mCertiKOS [18, 28, 30]. In Pip and Pip-MPU, formal proofs are directly conducted on the source code, which corresponds to the lowest specification level in the other systems.

The code and proofs of both Pip [1] and Pip-MPU [2] are open-source and available online.

2.4 Hoare logic

Pip and Pip-MPU are coded in imperative programming style, which makes **Hoare logic** [19] suitable to formally verify the kernel services. Hoare logic gives a formal system to conduct the proofs of properties of a program, *i.e.* rigorous deductive reasoning rules of inference. In Hoare logic, a program Q is encapsulated between a pre-condition P and a post-condition R and formally expressed as the following **Hoare triple** $\{P\}Q\{R\}$. It reads as the post-condition R is met whenever the program Q executes and terminates with pre-condition P. The pre- and post-conditions are general assertions (properties). Hoare triples follow the rules of inference involving consequence (for $\{P\}Q\{R\}$: if the post-condition assertion R logically implies assertion S, then $\{P\}Q\{S\}$, and inversely, if the pre-condition P is implied by an assertion O, then $\{O\}Q\{R\}$), composition (break down a program sequence by sequence and connect Hoare triples on each sequence to prove the full program, *i.e.* for a program of sequence ($Q1$; $Q2$) connecting the post-condition

[1] https://github.com/2xs/pipcore
[2] https://github.com/2xs/pipcore-mpu/

of the first sequence's Hoare triple with the pre-condition of the second sequence's Hoare triple: $\{P\}Q1\{R1\}$ and $\{R1\}Q2\{R\}$ then $\{P\}(Q1;Q2)\{R\}$), iteration (the controlling condition of a loop is false when leaving the loop). Pre- and post-conditions can hold the same properties: these are **invariants**. Although the adoption of separation logic [32] — an extension of Hoare logic — could enhance the formal verification efforts for Pip and Pip-MPU, it is not employed in these projects. Additionally, Pip and Pip-MPU are designed exclusively for single-core systems; the sole form of concurrency present is through interrupts, which are systematically disabled during the execution of kernel services to maintain sequential consistency and simplify the verification process.

2.5 Quantification of software development projects progress

One of the agile software development processes is Scrum which is an iterative incremental process commonly used in the industry. In this process, the work is delivered on a periodic basis, called *sprints*. As described in [10], Scrum teams use Burndown Charts to follow the progress of their work done during each sprint and use Burnup charts to follow their overall progression over time as well as the evolution of the overall scope of work.

An example of Burndown chart is presented in Figure 2. The horizontal axis shows each sprint. The vertical axis shows the amount of work remaining expressed in *story points*, a unit of measure which expresses an estimate of the overall effort to fully implement a work item. Teams assign story points relative to work complexity, amount of work in days, risk or uncertainty for example.

Unlike the Burndown chart, which plots remaining work, the Burnup chart depicted in Figure 3 illustrates the work completed. The dotted line represents the targeted scope of the work agreed with the customer. It is especially useful to materialize any increase of work that could stem from additional features required by the customer or reactions required to events (for example an external dependency changes its licence and the team needs to develop a replacement).

However, the Scrum methodology requires adaptation to the specific domain in which it is applied. For example, Tona *et al.* [41] adapted the Scrum framework to more effectively incorporate software quality requirements and quality processes. They highlight that adapting the quantifying of progress in terms of quality is a challenge. Our work aims to address a similar issue within the domain of proof development.

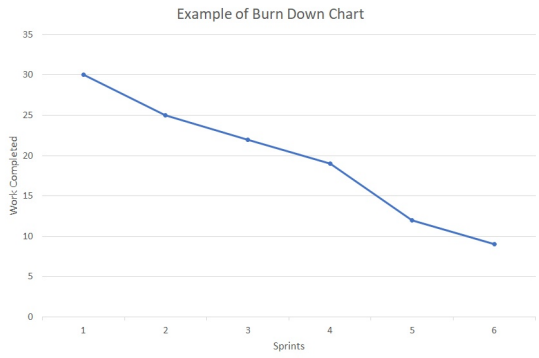

Fig. 2. Example of Burndown Chart

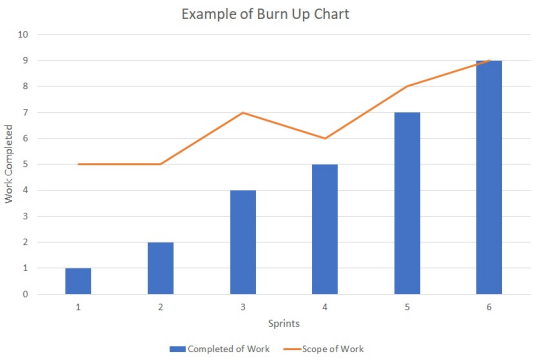

Fig. 3. Example of Burnup Chart

3 THE PROOF CRAFTMANSHIP

3.1 Software formalisation

Unlike informal specifications, formal specifications eliminate the need for interpretation, offering a single, unambiguous way to understand the requirements. This clarity prevents inconsistencies and contradictions from arising in the specification, which in turn helps guarantee the software's validity according to the defined behaviour. Moreover, formal specifications go beyond simply aiding comprehension; they function as a mechanism for validating the developed software. By employing formal specifications, developers can ensure not only a thorough understanding of the software's requirements but also a robust verification of its correctness, instilling greater confidence in the final product.

The use of formal methods in software development comes with its own set of challenges and costs. Specification complexity can arise as the engineer expresses the specification through a formal language; but also while he works to eliminate contradictions between specified properties. Merely writing the specification is not sufficient; careful attention must be paid to avoid any inconsistencies. Nevertheless, proof complexity is the most challenging aspect to consider. While formalisation is a prerequisite for establishing proof, it does not entirely eliminate the possibility of incorrect interpretations of the specification. In this context, software proofs play a crucial role in debugging the specification. A software system is deemed correct with respect to its specification, and conversely, the specification is considered correct with respect to the software. If it becomes impossible to prove a formal property within the software, the specification may be called into question, requiring corrections or removal of ambiguities. Moreover, there is no straightforward relationship between the number of lines of code and the number of lines of proof. A linear correlation does not exist, adding another layer of complexity to the process: progression in the proof is very hard to estimate.

In the following, we focus on proofs of software. Given that we examine properties extracted from the initial state or any subsequent, the code must exhibit a well-defined control flow (*e.g.* read/write instructions, conditional branches, function calls and subroutines, independent services) with no external events potentially altering the state and associated properties (*e.g.* interrupts). Hidden properties are unacceptable due to the indirect risk they pose of resulting in an inconsistent state.

3.2 Our approach: proof/soft co-design

The primary challenge in using formal methods lies in the associated high costs. This raises the question: how can we effectively manage the development and proof cycle?

To remain efficient, it is essential to address the following three key questions:

- What is the formal property that we aim to establish?
- Which fragment of the software should we reason about in order to verify this property?
- Which fragment of the software is not relevant to this particular line of reasoning?

By carefully considering these questions, developers can target their efforts to the most critical aspects of the software and its corresponding formal properties. This focused approach enables a more efficient verification process.

Our approach, therefore, involves iterating through the following steps:

(1) Identifying a subset of formal properties that are essential to the software's correctness;
(2) Identifying and verifying the software fragments that could affect these properties;
(3) Ensuring that the remaining parts of the software do not impact these properties.

The adoption of this approach offers a stronger guarantee than merely selecting and verifying the functional correctness of arbitrary software fragments. It is crucial to recognise that it is impossible to verify everything; the engineers must prioritise the most critical properties and the corresponding software fragments, considering the mutual dependency between them.

As a result of our previous works [26], we have developed a software formalisation strategy known as **proof/soft co-design**. This innovative approach fosters a more focused and efficient method for formal verification of software development, enabling developers to capitalise on the advantages of formal methods while effectively managing the costs and complexities involved.

3.3 Deeper in the proof process

Proof/soft co-design primarily relies on feedback throughout the various software development stages, ranging from the definition of its specification to the formal verification of its properties.

In order to reason about programs using Hoare triples, it is essential to distinguish between three types of properties:

Specification properties These properties stem from the formal specification and define the expected behaviour of the program. These can be functional properties (*e.g.*, a particular function returns a specific result) or non-functional properties (*e.g.*, a certain machine state is always true).

Consistency properties These properties establish invariants that are not expected *a priori* but are identified as necessary for reasoning about the software. For example, a consistency property may express that a field is never null, data within a data structure is always positive, or a specific operation is never requested from the hardware, regardless of the software function being considered. While there is an infinite number of potential consistency properties, only those that aid in reasoning about the program to prove specification properties are relevant.

Internal properties Internal properties are locally or temporarily true within a function. For instance, in the "then" branch of an "if" statement, the property tested by the "if" can be assumed to be true. Similarly, if within a function 'v' is defined as twice 'u', an internal property can declare that 'v' is even. Once again, there is an infinite number of possible internal properties, but only those that are relevant for reasoning about a given function should be considered, and only for the software fragment where they hold true.

The Specification properties are provided by the formal specification step, establishing the foundation for reasoning about the software. However, the other two types of properties - consistency properties and internal properties - emerge as a result of the proof process. By working through the proof, engineers identify these additional properties that are necessary for establishing the formal specifications.

In Section 3.2, we defined an iterative process with three looping steps. It is now necessary to outline the verification process for steps 2 and 3 of this general activity. This engineering process relies on the careful and sequential application of our Hoare triples on elementary operations composing the function. Fig. 4 illustrates the general schema of verification steps. It describes how a particular property is verified after an instruction execution and shows the various decision paths that exist. Choosing an adequate path is important to reduce verification costs.

Indeed, the verification of a set of properties after an instruction execution classifies them into two categories. Each one can also be decomposed into several categories as follows and as illustrated by Fig. 4.

When a property can be proven after the execution of an instruction, one question remains:

A. If the property is a general one, originating from the specification or global consistency, we must be able to provide justifications that validate the property's continued applicability after the instruction has been executed.

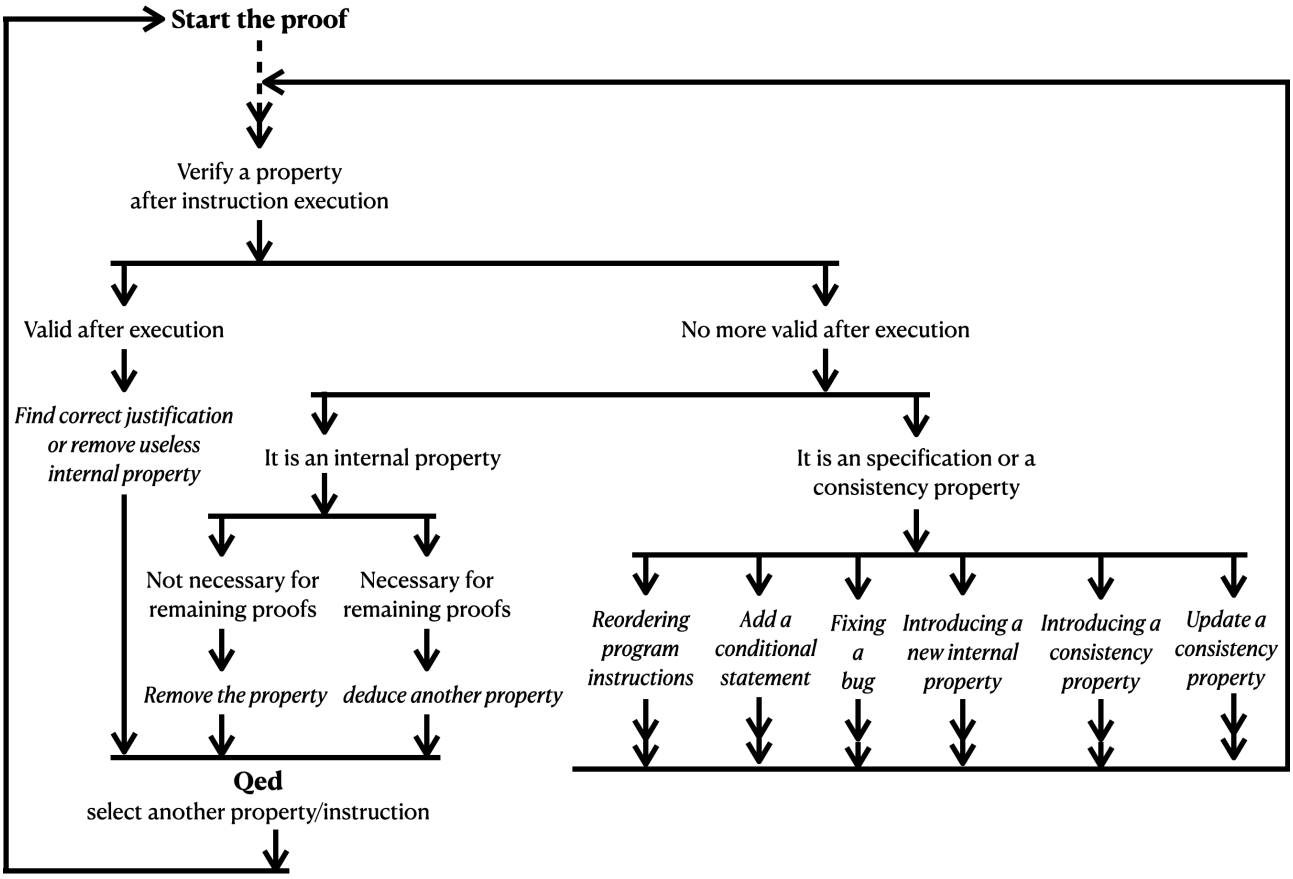

Fig. 4. The general schema of verification steps.

B. If the property is an internal one, the first question to consider is: Do we still need it? Internal properties are established for local purposes, and after they have served their function, they may no longer be necessary. In such cases, these properties can simply be forgotten.

When a property cannot be proven after the execution of an instruction, complications arise.

A. If the property is an internal one:

1. If it is not necessary to reason about the remaining instructions, the property should be removed from the set of properties to propagate by updating the post-condition of the current instruction.
2. If it is necessary to reason about the remaining instructions and the property is internal, it can always be locally replaced with a deduction from the existing one. It is important to note that the specification of internal properties about an executing instruction is initially based on an intuition of some local invariant that could be expressed in different ways. Therefore, it is possible that the definition might not be exactly as anticipated. Changing the property where it has been defined can lead to updates in several existing proofs. For this reason, it is more convenient to locally deduce a new property that is better suited to the current reasoning. In most cases, this transformation is not difficult to establish.

B. However, if the property in question is a specification or consistency property, the difficulty can be addressed in various ways:

Reordering program instructions Sometimes, a consistency property or even a specification property can be temporarily invalidated. This can occur when the software alters several elements in a global state, for example, and doing so step-by-step results in a temporarily invalid data structure. As the reasoning is highly dependent on the order in which the instructions are executed, simple reordering of instructions can sometimes provide the easiest way to manage the proof. Our verification experience indicates that creating a temporarily inconsistent state during the execution of a piece of code can complicate the proof. This code modification can lead to one of two possible outcomes:

1. Move the current instruction forward to a more convenient position for later execution. This requires modifying its invariant to follow the program's control flow;
2. Execute the instruction earlier in the code, which necessitates updating the Hoare triples of the subsequent instructions by integrating the corresponding internal property.

Code updates can, in some cases, result in significant modifications to existing proofs. However, the extent of the changes largely depends on the nature of the modification. Therefore, it is essential to carefully organise the code before beginning the verification of the program, as this helps creating a more structured program and preventing temporarily inconsistent data structures.

Introducing a new internal property In cases where global properties cannot be temporarily preserved, an internal property can be used to capture the partial consistency of the failing global property. Propagating transformations can then continue until a new consistent global state is reached and the global property is reinstated. As previously mentioned, internal properties are defined during the proof of instruction streams, and their formalisation involves anticipating what may be important for the proof. Consequently, it is quite possible to miss some internal properties. In such cases, the corresponding instruction needs to be identified, and the new property specified and propagated. This type of local adjustment is only possible with local Hoare triples because global properties are expected to hold true at both the beginning and end of a function.

Introducing a consistency property Sometimes, incorporating a new consistency property is necessary for completing the current proof. It provides additional information about the global system state or data structure that aids in completing the verification of the property. This step demands more effort as it entails significant updates to existing proofs. In our case, existing proofs are robust when adding new consistency properties, and generally, we only need to make changes to include missing proofs regarding the validity of the new consistency property.

Add a conditional statement Global system states and data structures often have interrelated and mutual dependencies. As a result, it's possible to infer properties and relationships between them. However, without an explicit consistency property expressing this relation, making such inferences isn't possible. As mentioned earlier, integrating a new consistency property can be a challenging task. In some cases, this can be avoided by explicitly testing the validity of certain properties, although minor code changes may be necessary. It's important to note that the goal of code changes is to facilitate proof without causing performance degradation.

Update a consistency property As previously discussed, consistency properties are gradually integrated during the proof. Consequently, it's possible to integrate a misleading property that may not be accurate enough. Such modifications are rarely performed, and any corresponding ambiguity is usually captured. Indeed,

in some cases, verifying the validity of a property is impossible, especially when it depends on a modified global structure or system state.

Fixing a bug In this situation, both source code and proofs need to be synchronised. If the issue involves adding missing checks, updating the existing proof is usually not difficult, as we failed back in the "Add a condition before the instruction" item. However, when the problem relates to a design issue, corrections may invalidate a significant portion of the proof. Thus, it's beneficial to eliminate obvious bugs before starting the verification process. Based on the lessons we have learned from our practice in program verification, it is always preferable to eliminate bugs before attempting to prove a program, as debugging a partially proven program can be laborious. For this reason, we recommend thoroughly testing the software prior to proving it. The time spent on testing is always negligible compared to the time lost revising a proof due to changes in the program. This is particularly relevant because the debugging process introduces modifications to the instructions, which can jeopardise some or all of the existing proofs.

If the progression of the proof in the lines of code of a complex software, relies, in one way or another on one of these solutions, to find which one, remains an exhausting work.

3.4 Illustration

In this section, we aim to clarify the different strategies for overcoming proof difficulties in software as detailed in Fig. 4. We illustrate the iterative nature of proof development and the complexity of reasoning by examining a prevalent challenge in multi-threaded software: ensuring the correct use of mutual exclusion mechanisms to prevent deadlocks.

A deadlock occurs when a set of threads are blocked, each waiting for a resource held by another thread in the set, creating a cycle of dependencies that cannot be resolved. Informally, the *specification property* we need to establish is that no deadlock scenario arises. Specifically, it means no situation where the unlocking of one thread depends on another thread, which in turn, directly or indirectly, relies on the first thread for unlocking (*i.e.* a locking cycle). Formally, the specification property involves defining the concept of threads, either in relation to an application's thread management plan or through Hoare triples related to thread creation and activation API. The concept of "unlocking" is tied to Hoare triples focusing on the acquisition (*i.e.* `lock`) and release (*i.e.* `unlock`) of mutexes. After establishing these specification properties, it's crucial to prove that the property holds true across all possible execution paths of the threads.

To demonstrate the specification property, and following Fig. 4, we state a *consistency property* asserting: "a thread cannot acquire a second mutex until it has released the first one". Thus, a thread cannot be in a state of waiting to unlock while simultaneously being responsible for locking another thread, which could potentially lead to a deadlock. By adhering to one mutex at a time, a thread either waits for a lock or, having acquired one, does not hold any other, thereby avoiding a locking cycle.

Consider first the following pseudocode fragment, illustrating a function that alternates between locking mutexes v1 and v2 within a loop:

```
1: while a do
2:    if b then
3:       lock v1
4:    end if
5:    if c then
6:       lock v2
7:    end if
8:    ... code fragment without further locks and without modifying b and c...
9:    if b then
10:      unlock v1
11:   end if
12:   if c then
13:      unlock v2
14:   end if
15: end while
```

The proof of this code fragment must establish that it does not lock both v1 and v2 simultaneously. One strategy is to *introduce a new internal property* such as "$b = \neg c$". Moreover, it should be verified that b and c remain unchanged between the conditional locking and unlocking statements. Doing so, only one mutex in {v1,v2} is locked during a loop and unlocked before the next loop. However, some scenarios could complicate the proof:

- *'it is an internal property' -> 'not necessary for the remaining proof'*: If it concerns the internal property '$b = \neg c$', it is possible that this property cannot be proven, either before or after the while loop, because it only holds in the `while` block.
- *'it is an internal property' -> necessary for remaining proofs*: But if this property is still useful afterwards, yet unprovable, it might be that another property could be used instead, which implies *'deduce another property'* on Fig. 4. For example, '$\neg(b \wedge c)$' because sometimes, neither b nor c are true, and this does not pose any issue for proving the internal property.

As explained in the previous section, it can also happen that a consistency property, or even a specification property, turns out to be impossible to prove. In the case of the stated consistency property, the following program fragment cannot be proven because the two locks are acquired simultaneously:

```
1: lock v1
2: ...
3: lock v2
4: unlock v1
5: ...
6: unlock v2
```

However, this does not imply there is no deadlock. Indeed, if this is the sole location where two locks are reserved simultaneously, no other thread can lock v2, fail to unlock it, and then request v1. To address this, we can apply different strategies from Fig. 4:

- *'it is a consistency property' -> 'reordering program instructions'*: In fact, it is common to construct programs, either intentionally or unintentionally, where a consistency property is "temporarily wrong". But when possible, the simplest approach is to change the order of instructions so that this is no longer the case. Here, it could be to

swap the two instructions `lock v2` and `unlock v1`. Based on our experience, this loophole should be preferred when applicable, and it really aligns with the idea of 'proof/code *co-design*'.

- *'it is a consistency property' -> 'add a conditional statement'*: Sometimes, the solution to advance the proof when we encounter a property that is impossible to prove is to adopt what is called defensive programming. In this approach, we test a condition in the software that we are unable to establish through proof. For example:

```
1: if no_mutex_lock then
2:     lock v1
3: end if
```

  Of course, this involves replacing a property that is verified once and for all with a piece of code that will be executed every time a thread passes through this line. However, there are situations where this remains the only solution, in the absence of being able to establish the required property "statically".

Now consider the following program:

```
1: if v1 != null then
2:     lock v1
3: end if
4: if v2 != null then
5:     lock v2
6: end if
7: ...
```

For this code fragment, applicable strategies from Fig. 4 are:

- *'it is a consistency property' -> 'introducing a new internal property'*: In this case, it might be relevant (if it is assumed to be true) to add a new internal property: 'if v1 is not null, then v2 is null, and vice versa.' However, this new property would need to be verified throughout the entire program.
- *'it is a consistency property' -> 'introducing a consistency property'*: If v1 and v2 are global references throughout the program and not internal to a specific function, then the previously mentioned internal property is no longer local, but a global property of the entire program (consistency property), which must be verified for each function in the program.

Ultimately, it's possible that, after extensive proof work, the software engineers working on the critical software realise that the initial consistency property, namely "a thread never reserves more than one lock at a time", turns out to be impractical. They might then decide to *'update a consistency property'* with another like the one once proposed by Dijkstra: "commit to reserving multiple locks always in the same order". But then, all the proof work done so far will need to be revisited, and Dijkstra's proof will have to be reexamined to establish that this consistency property implies the absence of deadlock, significantly encumbering the proof and substantially increasing the proof effort.

The examples taken in this section are limited and quite simple. When dealing with more complex programs, such as a real-world OS kernel, each iteration of the proof development process significantly increases the proof effort. This increase is due to the multiplication of the possible strategies and the necessity to address incoherent proof states and bugs intrinsic to complex programs and proofs, which are depicted by all paths of Fig. 4.

3.5 Lead a code to Qed

When a property cannot be proven, it is necessary for the engineer to choose the order in which to attempt the various solutions that have just been presented. These solutions differ in terms of proof effort and probability of success. We can distinguish two categories of solutions: those that act on the code and those that act on the proof. The solutions that act on the code are "**Reordering program instructions**", "**Add a conditional statement**", and "**Fixing a bug**", while the solutions that rely on the proof are "**Introducing a new internal property**", "**Introducing a consistency property**", and "**Update a consistency property**".

In the next paragraph, we present our feedback on the best order in which possible solutions should be tried to deal with the difficulty to prove a global property.

Solutions that act on the code often generate less co-design effort and are therefore generally preferred, and sometimes even absolutely necessary.

If there is a bug, it must be fixed. The "**Fixing a bug**" solution should always be prioritised if possible, and it is the first area to explore. However, since the software being proven is co-designed during a formal specification process and has been written in a formal language, the likelihood of a bug being present is rare, according to our experience. But precisely because it is improbable, it must be kept in mind that if we cannot prove a consistency property, or one resulting from the specifications, it is perhaps simply because as it stands, it is not true.

The second preferred solution is "**Reordering program instructions**". When applicable, this solution has numerous advantages. As explained earlier, it can have a very low impact on the engineer's proof effort if well organised, and it has a localised impact. Furthermore, when this solution addresses internal system states, it may indicate a possible "race condition" in the hardware.

The third preferred solution, according to our experience, is a tie between "**Introducing a new internal property**" and "**Add a conditional statement**". While once again, the solution that involves changing the code rather than the proof may seem better in terms of engineering effort, adding runtime checks – that is, switching from static to dynamic verification – can ultimately impact software performance. After all, the same guarantee can often be obtained statically, specifically through the introduction of a new internal property for the function being reasoned about. Moreover, this static solution has only a minor impact on the proof effort since it only affects the function fragment being considered.

If none of these solutions is possible, the remaining two solutions imply a significant proof effort. Indeed, "**Introducing a consistency property**" introduces a new global property that must be verified throughout the whole code that has already been proven, not just the code fragment being proven, resulting in a considerable proof effort.

The solution to "**Update a consistency property**" may seem somewhat less daunting, but in practice, even with significant efforts to organise proof scripts, modifying a property involved in a proof requires modifying the proof itself, necessitating substantial rewriting of the proof.

3.6 Discussion

Managing industrial projects involving the integration of formal methods is challenging not only because the cost of proving software is high (in terms of man-months), but also because it is difficult to plan. As we have seen, the proof workload is proportional to the number of global properties to be proven in the code, and after the formal specification phase, this number is not yet determined. The number of global properties is equal to the number defined by the formal specifications, *plus* the number of consistency properties discovered during the proof construction process itself. We

later provide an illustration of this fluctuating proof workload from the proof developer's point of view in Section 5 and from the project manager's point of view in Section 6.

Furthermore, the iterative proof process is not boundable *a priori*, as adding new global consistency properties or modifying these properties requires engineers to revisit code they have already reasoned about.

Domain experts, however, often overlook or neglect this situation. After several years of working on a specific type of software (an allocation algorithm, a scheduling algorithm, *etc.*), they acquire an expertise in the data structures and services they reason about and, consequently, tend to anticipate the appropriate consistency properties required by the upcoming proof. Similar obstacles in revising consistency properties, unfavorable for proof production, have also been observed in other approaches such as that of seL4 [43]. Therefore, in general, we believe it is necessary to assume that not all consistency properties can be identified *a priori*. It is important to anticipate the consequences of revising these properties in the most unfavorable situations.

Finally, when software or hardware evolves, the proofs shatter, and the process must start over. It is therefore essential to identify the sources causing additional proof effort and to provide metrics to measure progress. These metrics serve multiple purposes: they give professionals a sense of progress in the task, provide indicators to quantify costs and resources to commit to project managers, and enable them to report on the advancement of the proof. Lastly, these metrics should also help in factoring and steering proof efforts.

4 PROOF MANAGEMENT

From the sections above, it is reflected that the proof development process is a complex and intellectually demanding task, requiring a high proof effort.

Intuition is the leitmotiv at each stage, from the proof term construction to the proof planning, where "efficiency" and success depend on highly skilled proof experts. The consequences include lack of systematic proof development strategies, inability to communicate the advancements of the proof especially for people used to software engineering, inability for a project manager to plan necessary resources and manage the efforts in time accordingly, a long and steep learning curve for newcomers to master tools and proof techniques for formal verification and thus heavily rely on senior proof developers.

Furthermore, the proof backtracking loop, represented on the right-hand side of the Figure 4, shows the sources of additional proof effort ("**Introducing a new internal property**", "**Introducing a consistency property**") and wasted proof effort ("**Reordering program instructions**", "**Fixing a bug**"). Consequences involve higher project costs, reconsideration of the project planning, proof developer frustration by reiterating the proof process, production costs due to the discovered bugs with potential malfunctioning or security breaches.

Therefore, we need to deepen our understanding of the internals of the proofs and their relationships with the code, as well as organising and optimising the proof process described in the previous section. We introduce in the following tools, metrics, and strategies to substitute intuition and counter the listed consequences, with the goal to elevate the reusability of proofs, avoid proof backtracking, explicitate the proof development progress and plan the proof development in reasoned progressive steps, with the ambition to reduce the likelihood of additional and wasted proof effort. The context is the formal verification of a software system within the boundaries announced in the previous section, *i.e.* using interactive theorem proving and leveraging Hoare Logic on a piece of software with a clearly defined control flow.

This section is voluntarily made theoretical, while the presented material is illustrated on a real-world system in Section 5.

4.1 Motivating example

Full formal verification entails a thorough examination of each component within the codebase, with the proof assistant ensuring correctness independently of the verification sequence or a specific proof script. In a naive approach, discussed in previous Section 3, the consistency properties are discovered incrementally at each new step in the proof, making it difficult to **assess the proof effort from the beginning of the project**. In contrast, the ultimate goal for the manager of formal verification teams is a **long-term planning** that **minimizes the overall proof effort** by allocating resources effectively and optimizing the work of the proof engineers. For example, the maximisation of the proof reusability and the early discovery of the consistency properties help to reach this goal by mutualizing proof efforts and reducing the likelihood of the proof loopbacks depicted in Figure 4. To illustrate this, consider a software component composed of 3 functions *A, B, C*, represented in Figure 5. These functions are themselves composed of the instructions and internal functions indifferently referred to as *1, 2, 3, 4, 5*. From the functional point of view, we observe that element 1 is both used by *A* and *B*. In other words, the code is **reused**, avoiding code duplication and reducing maintenance necessities to keep both in sync. The same principle applies to proofs: we aim to **compute and maximize the reusability factor** of proven theorems and lemmas. Additionally, discovering consistency properties at an early stage enables early property sharing extracted from different contexts, thereby lightening the burden of proof loopbacks. For instance, when starting the proofs of *A*, *B* and *C* simultaneously, this property sharing occurs at the earliest stage, significantly streamlining the overall proof process. However, from a managerial perspective, partial proofs are not sufficient for reporting progress. Ideally, managers would like to **gradually observe convergence** to full formal verification according to a **predefined agenda**. This is tracked on a sort of **higher-level overview with status progress** including meaningful **views on the proof status**. But the **selection order of the functions to prove** significantly affects the proof effort and the proof outcomes. Indeed, while each function is a separate context with its own set of consistency properties, all properties will eventually merge into a unique statement for the entire system under formal verification. Therefore, functions spanning accross multiple contexts are more informative, and processing them first reduces the overall proof effort. For example, in Figure 5, we can prove *C* in two steps (by proving elements 4 and 5), but we can also prove *A* in two steps as well (by proving elements 1 and 2). However, due to the interleaving of proof elements (existence of reused elements), proving *A* also brings a lot of information on the proof of *B*, since it reuses elements 1 and 2. By finishing the proofs of *A* by encompassing a bit of *B*'s context, the overall proof context is richer than *C*'s context alone, thereby reducing the heaviness of proof loopback corrections. Furthermore, by leveraging again the reusability, the proof of *B* is terminated in one extra step by proving 3, while if we had proven *C* first, neither *A* nor *B* could have been proven in three steps, and so we **minimize the proof effort at each step of the proof while pursuing to cover most of the remaining proof**. Just as importantly, by starting to prove *A* and *B*, instead of *C*, more functions are **challenged earlier in the process** with the properties to prove, thus comforting the overall verification process (properties and absence of bugs endangering the properties). Naturally, the motivating example assumes uniform proof units, which is rarely the case with functions more complex than others. To refine prioritization, long-term planning requires an accurate measurement of the proof costs in terms of proof effort for all instructions and internal functions. This entails addressing two aspects: 1) **computing the proof effort**, and 2) **obtaining this accurate measurement as early as possible**. In addition to that, **short-term planning** is as important as the long-term planning discussed until now when considering proof statements within instructions and internal functions. Indeed, some code fragments are more important to challenge earlier in the verification process to find potential bugs within the functions themselves, which means **early identification and prioritisation of the proof of the most critical parts**.

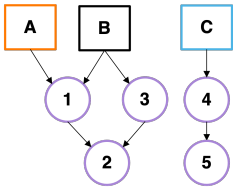

Fig. 5. Toy example of a software component composed of functions A, B and C, each of them composed of primitives and internal functions $1-5$.

By leveraging this approach, teams can effectively reduce redundancy and optimize their proof efforts, leading to more efficient and effective formal verification. This intuition is formally stated in the next sections.

4.2 Proof structure

The foundational aspect of formal verification is the synergy between code and proofs. It intrinsically binds portions of code with their corresponding proof elements. The major activity is to construct these proof elements (or proof artifacts), which are the building blocks to conduct the formal verification.

Definition 2 (Proof element). A proof element is a proven statement, *i.e.* a theorem, lemma or proposition, with respective proof.

Similar to how **code elements** (function, module, instruction) are composed in software, **proof elements** are systematically organized during the proof construction process. Indeed, proof elements interconnect to leverage previously established statements, aiming at proving a global statement, such as specification properties. Each function or primitive correlates with a primary proof element, supported by auxiliary proof elements that help the proof development, such as property extraction or property equivalence. At the end of the verification process, the total number of code elements (functions and primitives) indicates then the minimum number of proof elements.

Theorem 3 (Minimum number of proof elements). *For the set of code elements composed by the subset of functions* $F = \{F_1, F_2, ..., F_f\}$ *and the subset of primitives* $P = \{P_1, P_2, ..., P_p\}$*, and for the set of corresponding proof elements* $PE = \{PE_1, PE_2, ..., PE_l\}$*, then*

$$|F| + |P| <= |PE|$$

The interrelations between code elements and proof elements are represented as separate dependency graphs. A dependency graph is a polytree, a type of directed acyclic graph in which the underlying undirected graph is a tree. It delineates the primary code element's inner functions (subroutines) and excludes low-level primitives, *i.e.* functions comprise at least two low-level primitives. It is a subset of the call graph and control-flow graph, excluding recursive calls and the routes back from an inner function to the primary function. Figure 5 illustrates a typical (generic) dependency graph.

Definition 4 (Dependency graph). The dependency graph of code, respectively proofs, is defined as the set $G_{function} = (F, DC)$, respectively $G_{proof} = (PE, DP)$, where the set of functions $F = \{F_1, ..., F_n\}$ are the nodes in $G_{function}$, respectively where the set of main proof elements $PE = \{PE_1, ..., PE_l\}$ are nodes in G_{proof}, and the set of dependency DC are arrows (edges) in $G_{function}$, respectively DP in G_{proof}.

Two methods that structurally streamline the proof effort are design freeze and proof element reusability. Design freeze facilitates the analysis of proof dependency graph properties relative to code properties, with direct correspondence

between each function or primitive and their respective proof element. In that configuration, the code and proof dependency graphs match from a high-level perspective and the number of primary elements, *i.e.* functions and proof building blocks, is known in advance and feeds the planning. Reusability, akin to software development practices, aims to capitalize on previous efforts by reemploying elements in new contexts. Next, we introduce metrics to quantify the extent of reusability within the system.

4.2.1 Code and proof reuse. Embracing code reuse is a strategic approach to fostering the creation of similarly reusable proof components at a lower cost. Of course, proof components must be general enough to cover most of the cases where the corresponding code element is utilized. Proof reuse circumvents the redundancy of crafting similar proofs at different points within the proof script. The pursuit of developing reusable lemmas, or heuristics such as tactics (proof hints), to reduce the proof effort is an active area of research [21].

To assess proof reusability, we consider the dependency graph $G = (E, D)$, where $E = \{E_1, ..., E_n\}$ is the set of (code or proof) elements and D the set of dependency arrows (edges).

Definition 5 (Main reused element subgraph). Main reused elements are nodes in the dependency graph G that receive multiple incoming edges (indegrees). The subgraph comprising these reused elements is denoted by $R = \{G' \in G | \forall E_i \in G[E], deg^-(E_i) > 1\}$.

Definition 6 (Full reused element subgraph). Inner functions of reused elements are inherently reused, even if they may not be included in R due to a single incoming edge. The full reused element subgraph, therefore, extends R to encompass all elements within the subtrees $S = \{S_1, ..., S_s\}$, $s \in \mathbb{N}$, that originate from a main reused element. Hence, $G_{element\_reused} = \{G' \in G | \forall G_i \in G, \forall R_j \in R, \forall S_k \in S_{R_j}, G_i[E] \in S_k \wedge deg^-(G_i[E]) = 1\} = (E_{element\_reused}, A_{element\_reused})$.

Ensuring that the modularity of proof elements mirrors the modularity of code elements is key to minimizing proof effort. When proofs deviate from the code's structure, an additional proof effort is incurred for each developed proof component that *potentially* could have been reused. On the contrary, an increased number of reused elements leads to reduced development needs, as we consider that the benefits of reusability outweigh the resources invested in generalizing elements for reuse. Consequently, it is valuable to determine the actual proportion of reused elements across all nodes in the dependency graph to gauge efficiency, termed the **reused elements ratio**.

Definition 7 (Reused elements ratio metric). The reused elements ratio *RR* is the metric that quantifies the proportion of reused elements relative to all elements in the dependency graph G. It is calculated as the quotient of the number of reused elements in $E_{element\_reused}$ by the total number of elements in G:

$$RR = \frac{|E_{element\_reused}|}{|E|}$$

A high *RR* value indicates a significant presence of reusable elements within the dependency graph.

Note that root nodes, such as service entry points in an operating system invoked from user space, are excluded from $E_{element\_reused}$ because they are independent and thus lack incoming edges.

4.2.2 Reused proof discrepancy. In some situations, code and proof dependency graphs diverge, such as when establishing function-level lemmas proves challenging or when functions are so trivial that the effort to formalize proofs is unjustified. This divergence leads to a discrepancy, referred to as the **reused proof discrepancy**, between the potential

full reusability of elements, assuming matching dependency graphs (equivalent to full code reuse, RR_{code}), and the actual proof reuse, RR_{proof}.

Definition 8 (Reused proof discrepancy).

$$RR_{code} - RR_{proof}$$

A larger discrepancy points to missed optimization opportunities, such as redundant proof efforts, unnecessary additional lemmas, repetitive patterns in the proof script, and a more complex design making the maintenance operation more cumbersome. While the ideal is proper alignment between dependency graphs, a mismatch does not undermine the verification process, since proof assistants ensure completeness of proofs, regardless of proof structure.

4.2.3 Effective reusability. The counterpart metric to the reused ratio is the metric of effective reusability, which is a measure of the entanglement of all elements. Indeed, while functions may be reusable, the extent of their actual reuse directly reduces development and proof efforts. In other words, it compares the scenario where each element incurs a unique proof cost against the worst-case scenario where the dependency graph is fully expanded (unfolded) and each reused element necessitates additional proof effort by developing a similar proof.

Definition 9 (Unfolded dependency graph). The unfolded dependency graph is defined as $G_{unfolded} = (F', A')$, representing G devoid of modularity, with F' and A' being the new set of functions and dependencies, respectively. This graph is constructed by tracing each dependency path from the root nodes and *duplicating* all reused elements for each path, representing the most unfavorable case for code/proof development and maintenance. The total number of elements requiring development is $|F'|$. $|F'|$ is equal to the total number of elements in each subtree $SG' = \{SG'_1, ..., SG'_s\} \subseteq G$, so $|F'| = \sum_{i=1}^{s} |SG'_i|$.

From the perspective of the services' entry points, the total number of proof elements in the first scenario (a reused element incurs a single, one-time global verification cost) can be categorized into two groups: 1) the count of reused elements ($|G_{element\_reused}|$), and 2) the count of elements that are *not* reused. The latter is determined by pruning away the reused elements, resulting in the no-reused-element-pruned subgraph G_{pruned} defined below.

Definition 10 (No-reused-element-pruned subgraph). The no-reused-element-pruned subgraph G_{pruned} is derived from the code dependency graph G pruned away from reused elements: $G_{pruned} = \{G \setminus G_{element\_reused}\}$. We refer to its constituent subtrees as $SG'' = \{SG''_1, ..., SG''_s\} \subseteq G_{pruned} \subseteq G$. The total number of non reused elements is $|G_{pruned}| = \sum_{i=1}^{r} |SG''_i|$.

From the definition, $|G| = |G_{element\_reused}| + |G_{pruned}|$.

Finally, we can define **effective reusability**, which quantifies the degree of element reuse within the polytree G, and so equivalently the rate of interleaving in the polytree G. The effective reusability metric increases as more elements are actively reused.

Definition 11 (Effective reusability metric). Considering the polytrees $G_{element\_reused}$, G_{pruned} and their respective subtrees $SG'' = \{SG''_1, ..., SG''_r\} \subseteq G_{pruned} \subseteq G$ ($r \in \mathbb{N}$), as well as $G_{unfolded}$ with its respective subtrees $SG' = \{SG'_1, ..., SG'_s\} \subseteq G$ ($s \in \mathbb{N}$), and acknowledging that these subtrees share the same root nodes, we have $r = s$.

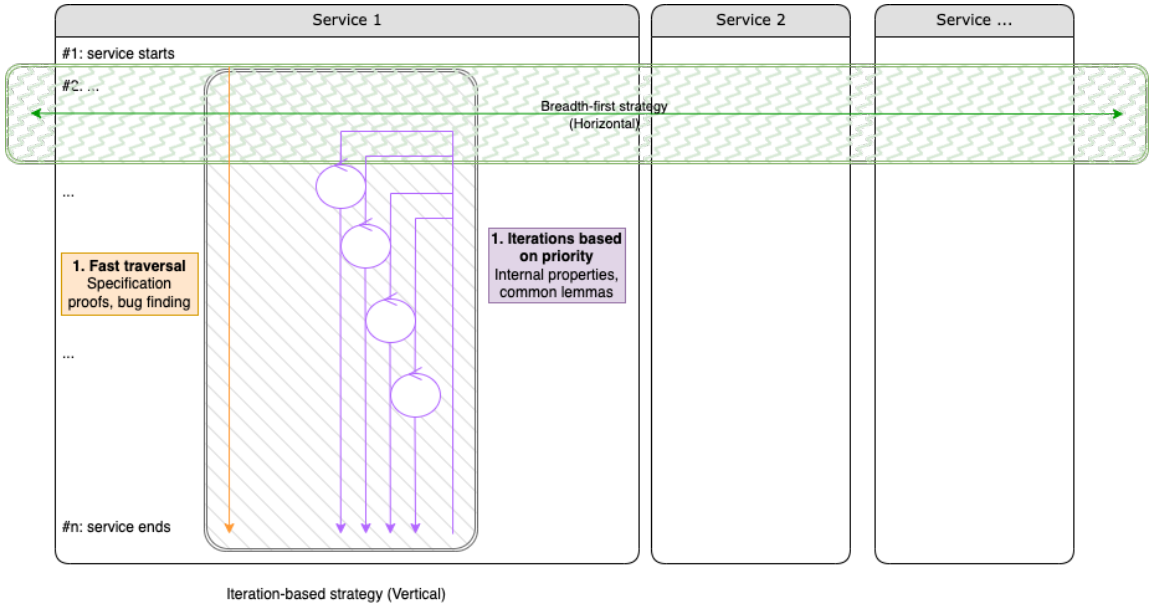

Fig. 6. Illustration of the proof strategies for the verification of OS services. In hatched green, the breadth-first exploration strategy covering all the services. In hatched gray, the priority-based iterative strategy for a specific service.

Then, the effective reusability *ER* metric is defined as:

$$ER = \frac{|G|}{|F'|} = \frac{|G_{element\_reused}| + |G_{pruned}|}{|F'|}$$
$$= \frac{|G_{element\_reused}| + \sum_{i=1}^{r} |SG_i''|}{\sum_{i=1}^{r} |SG_i'|}$$

ER indicates the rate of proof elements to prove with reuse compared to without reuse. Consequently, $1 - ER$ represents the proportion of proof effort saved thanks to reuse.

Note that the sequence in which each dependency path from the root nodes in G_{pruned} is considered (each subtree) does not affect the outcome of the metric.

4.3 Proof development strategy

Recall from Section 3.3 that consistency properties are invariants required to prove the specification properties and discovered *during* the proof process. As seen later in Section 3.5, we have several solutions to lead the code to Qed, however, we don't necessarily have the choice on the ones to apply and some are major sources of proof effort, like "**Introducing a consistency property**", "**Update a consistency property**" and "**Fixing a bug**".

We address these concerns with a proof strategy based on two movements: 1) the **breadth-first exploration strategy**: a fast and as complete as possible identification of consistency properties, and 2) the **priority-based iterative strategy**: an efficient traversal of the code element with iterations progressively increasing trust in the proof script while diminishing risks of inconsistencies. They have respectively a horizontal and a vertical movement criss-crossing the code elements, as illustrated in Figure 6, in the case of formally verifying OS services.

4.3.1 Breadth-first exploration strategy. The intuition of this strategy is that the more code is covered, the more consistency properties are unveiled. All code elements contain a check code portion, made of conditional statements (cf Section 3), which returns a result or rejects any unsatisfactory conditions like incorrect user parameters. This strategy explores first the check code portions in all code elements, in parallel, until reaching the modification phase. The check code portion requests drastically easier proofs than the modification phase, as attested in the proof development [24].

Furthermore, the check code portion brings up low-level structural properties that can't be easily discovered by deduction, for example an invariant property that a flag is set if and only if a field contains a meaningful pointer.

After applying this strategy, most consistency properties are revealed. The proof goal is detailed very rapidly by spanning the system, such as the proof workload is understood almost from start and the proof developers do not wander into unknown grounds for very long.

4.3.2 Priority-based iterative strategy. The intuition behind this strategy is to challenge the specification properties as fast as possible on the code to be proven. The first iteration cuts straight to the final state where the specification properties must be proven (*i.e.*, "*straight to the point*"). This implies temporarily dismissing relatively difficult proof obligations. Then, consecutive iterations solve the dismissed proof obligations by risk order, meaning properties with a likelihood to be false and tightly coupled with the consistency properties assuring the specification properties must be proven first. In such a way, trust increases drastically in the first iterations, while leaving less risky proof obligations to the end of the development (*i.e.* a false reasoning becomes exponentially less and less "probable"). Indeed, the objective is to quickly identify inconsistent properties, which reveal either bugs in the code or missing consistency properties that would help to resolve the specification properties and that cannot be deduced from the constructed proof context, assuming all dismissed proof obligations. After applying this strategy, a complete code element is proven.

4.4 Proof planning

The previously presented metrics and strategies help the proof manufacturing process. While the verification goal is usually set in advance, *i.e.* the full or partial verification of the code, proof management also needs a project view on the overall process to define milestones, prioritise, plan resources, communicate about the achievements and be sure the formal verification is on track.

We propose **short-term and long-term proof planning strategies** based on computed estimated proof effort, as well as a **proof dashboard** showing off the progress for each primary proof element (*e.g.* corresponding to the entry points of services provided by the OS) and a fine-grained analysis of the proofs to estimate the proof effort.

4.4.1 Short-term planning and proof effort estimation by proof complexity computation. In any project, planning requires meeting deadlines. Estimated effort is key to adjusting project management, like selecting the right amount of resources, retroplanning, or setting up sprints in an agile style. The short-term work organisation benefits from a fine-grained analysis of the work effort, specifically here the proof effort. The latter depends on the difficulty to undertake the proof. However, proofs are not of equal complexity. Indeed, a write instruction might be hard to prove, but harder is to prove a function with several write instructions.

This section introduces definitions to compute a **proof complexity score**, used as a basis to compute the **proof effort**. The intuition here is that the complexity of a proof, *i.e.* the **proof complexity**, depends on the impact of the code instructions on the properties to prove (some instructions disturb more the properties than others), *i.e.* the **proof impact score**, and how complex these properties intrinsically are, *i.e.* the **property complexity**.

Proof impact score at the instruction level. A first major categorisation of code instructions is whether they are read or write operations. Read instructions extract information from the context while write instructions modify it. In the perspective to verify properties, and in particular the specification properties, reads are tremendously easier than writes, because they are logical deduction from the same context instead of a changing one. Notably, invariants spare the logical deduction of proving the equivalence of states before and after the read operations, which is trivial.

In the case of write instructions, the impact on the proofs depends on the modification. Some operations, like a flag set required in the specification properties, might impact the properties significantly and could result in contradiction with the specification properties. On the contrary, modifications of elements of less importance do not disturb the properties very much or at all. Thus, each modification operation is analysed against each specification property and consistency property, and a disturbance score is systematically issued for each impacted property, *e.g.* if a modified structure field is present in the definition of a property. Hence, the more write instructions disturbing the properties, and the more impacted properties, the higher the impact score.

The matrix containing the impact scores of the instructions per property is called IIP.

Proof impact score at the function/service level. At the function level, the impact score depends directly on the instructions and their own impact score. Hence, we sum all the instruction impact scores given the instructions of the function and the properties to prove.

Functions might call other functions, the score being the sum of all scores of their own inner elements. We do not count the recursive calls if any, because we are only interested in the presence of a disturbance, not its occurrence.

Definition 12 (Function impact score). For a function F that has independent instructions I ($|I| = n$) in its own main function, the properties P ($|P| = p$) to prove, composed of the inner set of functions E ($|E| = e$), and considering the matrix $IIP = (I, P)$ relating the instructions' impacts on the properties, then the impact score ISS of F is defined as

$$ISS_F = \sum_{i=1,j=1}^{n,p} IIP[I_i, P_j] + \sum_{i=1}^{e} ISS_i + C$$

$C = 1$ is an additional impact score added to each element because there are always local adjustments to apply.

Property complexity. The impact score is not sufficient to describe the difficulty to conduct the proofs. In fact, the properties themselves participate in that difficulty given their intrinsic characteristics (*e.g.* their size, their variables, the proof goals).

The sum of all characteristics per property is reported in the column "Total" of the matrix called the **property complexity matrix** PCM.

Proof complexity. The proof complexity reflects the overall difficulty that requires problem-solving capabilities. It is a combination of the instruction impacts and the property complexities.

Definition 13 (Instruction proof complexity). The proof complexity PC of an instruction I is defined as the multiplication of the transposed corresponding column in the impact matrix IIP for I by the total column of the property complexity matrix PCM.

$$PC_I = IIP[I]^T * PCM[Total]$$

At a function level, proof complexity indicates the difficulties brought by each instruction to prove the properties.

One particularity is the type of the entry which is modified, and not only type modification of the entry. Indeed, types are included in the field change, whatever the change, in the sense that the type must be known to modify a value. For that, we compute the type impact score matrix *TIS* as the number of properties where the type intervenes, thereby matching a type to its global impact score on all properties. However, when proving a property, both the type and the address of any field change present in the modified state are projected on the statement to prove. The case discrimination to handle the proof then depends on which type is modified *and* the different modified memory addresses: the more types combined with modified addresses, the more cases to handle. For example, when the content of an address modified previously keeps its type, the same case discrimination occurs as previously, and so the proof difficulty stays the same. However, if the type changes, or when considering another address, even of the same type, it ends in an additional case to discriminate in the proof. In other words, the proof cares about each unique association between a type and an address, and not their total number computed independently. Hence, the impact of types on the proof must be considered with the modified addresses from a function's perspective, and counted separately.

Definition 14 (Type function proof complexity). Consider the map $M : I \rightarrow (T, A)$ mapping the instructions I to their type in the set of types T and the modified memory address from the set of memory addresses A.

The type function proof complexity $TFPC$ of function F is defined as the sum of the type impact score TIS for each type projected from the unique tuples in $TA = \{(t, a) | t \in T \wedge a \in A\}$ of size s corresponding to M applied to all instructions of F:

$$TFPC_F = \sum_{i=1}^{s} TIS[TA_i(t)]$$

Finally, we can define the function proof complexity encompassing the instructions and the types used in the function.

Definition 15 (Function proof complexity). The proof complexity PC of a function F composed of n instructions is defined as the sum of each instruction's proof complexity added to the type proof complexity of F.

$$PC_F = TFPC_F + \sum_{i=1}^{n} PC_i(F)$$

At a system level, the **overall proof complexity** depends on the proof elements that are the most difficult to prove.

Definition 16 (Overall proof complexity). The overall proof complexity OPC is the highest proof complexity in the set of proof complexities PC found in the service-level proofs:

$$OPC = max(PC)$$

Proof effort. It is important to differentiate between proof complexity, as discussed here, and proof effort, as defined in Section 1.

There are many ways to compute the proof effort. For example, it can be retrospectively computed as the person-hours or person-months invested in the formal verification activity. Unfortunately, this measure does not consider the difficulty of the task and the experience of the proof developers. As such, man-months is an excellent standard measurement of the amount of work and can be used for financial tracking, however, little does it say if the same work could be done faster and how difficult it was.

Hence, we introduce the notion of **estimated proof effort**.

Definition 17 (Estimated proof effort). The estimated proof effort *EPE* relies on the proof complexity *PC* of the n ($n \in \mathbb{N}$) proof elements to prove.

It is calculated as:

$$EPE = n + \sum_{i=1}^{n} PC(i)$$

n again represents the one-time cost to locally adapt the proofs to the reusable elements.

With *EPE*, one can schedule work in advance and gauge the task's complexity. *EPE* can be adapted to any project, provided there is a clear definition of the expected proof complexity.

4.4.2 Long-term proof planning strategy and associated tool. Without proof planning at a holistic level, the verification process is blind to higher-level optimisations. For example, there are multiple ways to reach the ultimate proof goal; like proving the services of a system by alphabetical order or by proving first all internal functions and primitives and later the entry points of the services. At first sight, the proof goal, *i.e.* the proofs of all code elements with respect to the specification properties, hides the relationships between these elements. Indeed, inner elements of the proofs are reused, just like functions are in the global context. In other words, the selection of a code element to verify covers more or less other elements, *e.g.* the reused functions in other services.

We propose a proof planning based on a strategy that solves in priority the proof element requiring the *minimal proof effort to be completely proven while maximising the impact on other proof elements*. In other terms, this strategy defines an order in fully verifying code elements from the lowest to the highest proof difficulty, so with a gradually increasing confidence. The inclusion of the impact on the other proof elements, in the selection order, progressively decreases the proof effort of the most difficult proofs. As such, our previous effort from Section 4.2.3 to structure the proof with high reusability benefits the convergence speed of a formally-verified system. Furthermore, verifying a code element and its dependencies together, *versus* randomly choosing a code element to prove before connecting the dots with the rest, seems more reliable because it challenges the specification properties the fastest by reaching the conclusion of the code element.

The order in which the services must be proven is computed thanks to Algorithm 1 where the goal is to minimize the current proof effort and to maximise the proof coverage. Indeed, we favour a service whose proof effort is lower than others, but enables to lower the most the proof effort of the remaining elements thanks to reused proof elements.

4.4.3 Proof dashboard. The proof dashboard is a dynamic tool that reports the proof status of a long-lasting activity. It modifies the perspective on proofs as proof development progresses. The dynamism is inherited from the presented proof strategies which constantly adjust with incoming events like newly discovered consistency properties or design modifications. It disposes of **three views**: the **dependency graph coverage**, the **instruction-level progress**, the **resolution**.

Dependency graph coverage At the beginning of the proof development, we have a complete view of the code dependency graph. The goal is to develop all the proof elements from the related proof dependency graph. We can then present the proof status by a graph coverage, *i.e* which code elements have been proven.

Instruction-level progress However, having coverage at the proof-element level is not precise enough and the proof status would not change for a long time. For example, for a primary element, it would mean waiting for the proofs of all inner proof elements which could take a long time. Thus, we would like an instruction-level rough estimation to understand the current focus.

Algorithm 1 Proof path best effort strategy algorithm.

set *global_dependency_graph* **with** global dependency graph of all services
repeat
  for *service* **in** *global_dependency_graph* **do**
    set *proof_path* **with** proof path for *service* from the *global_dependency_graph*
    set *current_proof_effort*[*service*] **with** current proof effort for *proof_path*
    set *spared_proof_effort*[*service*] **with** spared proof effort on the other services (reusability)
  end for
  set *services_min_current_proof_effort* **with** all *services* scoring minimum in *current_proof_effort* (lowest proof effort for the developer)
  if several services have the same current proof effort **then**
    set *services_max_spared_proof_effort* **with** all services in *services_min_current_proof_effort* scoring maximum in *spared_proof_effort* (highest reusability, so most impact on the remaining proof)
    set *selected_service* **with** first service in the *services_max_spared_proof_effort* list
  else
    set *selected_service* **with** only service in *services_min_current_proof_effort*
  end if
  add *selected_service* to *service_order_list*
  remove all elements of the *selected_service* proof path from the *global_dependency_graph*
until *global_dependency_graph* is empty
return *service_order_list*

The initial estimate is based on the number of instructions that have been visited during the Hoare instruction breakdowns. The current proof pointer gives the ratio of proof coverage depending on the number of remaining instructions in a service.

However, among the remaining instructions for a given primary element, some might be similar to previously encountered instructions, such as functions or read and write instructions, for which proof elements have already been developed. In such cases, the proof is facilitated, only requiring local adjustments. Thus, it discriminates proof elements which have never been visited before from reusable elements with reduced proof effort. This view should then be combined with a precise assessment of the proof effort required for the remaining instructions to understand the extent of the remaining challenges.

Resolution Upon reaching the final instruction, the proof goal must be verified. The dashboard then breaks the latter property by property, for both the consistency properties and the specification properties. The amount of proven properties compared to the complete set of properties reports the advancement. Furthermore, in line with the priority-based iterative strategy described above, some lemmas may remain unaddressed. The number of the remaining admitted lemmas also reports the progress of the proof status (or in a way, how weak the proof currently is).

Although the *instruction-level progress* and *resolution* views are computed per code element, they give a global appreciation of the proof progress when considered across the entire system.

4.5 Discussion and future works

The proposed framework is based on experience gathered on two projects with formal verification approach totalising nearly 200.000 Lines of Proof in The Coq Proof Assistant (more than 180.000 Coq *tactics*). In the following, we discuss limitations and improvements.

Table 1. Proof dashboard dynamism: views depending on the granularity.

View	Granularity	Metric
Dependency graph coverage	System	Global dependency graph coverage
Instruction-level progress	Main code element	Ratio of proven code instructions (excluding reusable elements)
Resolution	Code element	Ratio of verified properties and number of remaining admitted lemmas

The framework is compatible with external code dependencies (libraries). If the external code is known, can be imported, and compiled with the original code base, its verification follows the steps outlined in Section 3. However, if the code is not known, only assumptions about it can be made and weaken the proofs. The framework absorbs this in the "Resolution" view, where all admitted lemmas are referenced, and in the project scope, where the *admits* will be left aside as no proof effort is needed. Ideally, the code should be fully available and proven, with as few assumptions as possible, as targeted in our two projects [16, 25].

The framework provides planning methods, tools and strategies to give the necessary overview to the proof developers but also to managers who need to know the advancements of their teams to properly plan resources and help them as they need. It brings a close understanding of the proof process for non-expert entities, which is the essential condition to ensure the verification approach exists and is sufficiently nourished with proper resources. Also, similarities with project management practices bridge Proof Engineering and Software Engineering and ease its adoption, like simplicity, factorisation, the best effort proof path and a backlog prioritisation or sprint planning, the proof complexity score and effort planning, constant and progressive added value, branching workflows serving the development and work organisation. Also, a high reusability score entails plenty of proof bricks to assemble for further development, for example if the API is extending to new services. However, while the framework advocates a high code modularity that can be inherited by the proofs, it is usually not expected in software development where the benefits of factorisation might not systematically cover the cost of development like for proofs with a longer design time and refactoring. Nevertheless, it seems the framework should be tested more broadly and evaluated against the received feedback, especially for larger proof development teams. While the computation of the proof complexity score might burden its use, all other tools and strategies are lightweight and ready to use for broader adoption. Indeed, at the initial stage, the dependency graphs can be easily recovered by simple analysis and metrics directly computed which fed the proof path best effort strategy algorithm. The proof dashboard requires manual investigation to collect the values to compute the ratios, but these are of limited number and inserted in a spreadsheet for the computation. The dashboard and the proof dependency graphs should then be updated at each project milestone (or more often if that is required by the hierarchical line to track the progress). But even the proof complexity computation should be put into perspective. The proof complexity computation does not require expert knowledge but for the moment being, the instruction-level proof impact score matrix IIP, as well as the property and type function complexity matrixes PCM and $TFPC$, need manual investigation. In the case of Pip-MPU, this represented a relatively negligible effort of 1.5 person-day. Even though the matrixes must be locally updated at each newly discovered consistency property or added functionality, the effort is not significant thanks to the breadth-first exploration strategy which collects most of the properties from the beginning and so fills in most of the matrixes during the primary effort. The rest is computed automatically in a spreadsheet. In addition to that, we have seen the proof development strategy enabled to spot bugs endangering the specification properties early in the proof process. In particular, the priority-based iterative strategy writes the guiding thread for the

proof which suits very well the life of a development team where turnovers are expected for such long development. The first steps in the proof process lay down the principles of the proof, testing the intuition to resolve the specification properties, while an external proof developer could catch the development on the go while following this thread. In that, it enables to split of the design, testing, and development phases of the proof.

With the framework, uncertainties about the estimated proof effort are reduced. The impact scores give an idea of the impacts on the current set of properties, however are not yet final. Indeed, while the breadth-first exploration strategy identified plenty of the consistency properties of the system, we expect some modifications of the set, and in turn the scores, along the verification process. The minimal set of properties which gives a sense of the final result is not yet determined. Other proof strategies might enhance the framework as well. *However, its efficiency is not the primary goal of this work, but the sole existence of a systematic description of the proof process and its rational planning*. We believe Proof Engineering has its own specificities that must be experienced for broader adoption, with this proposition to push and encourage projects to embrace proof development.

The framework does not consider a specific proof assistant. It relates instead to Hoare Logic and the higher level of properties, not related to proof instantiation, and so without consideration about the proof assistant. We provide a baseline score that the manager then adjusts based on the team's attributes (experience, similarities with other projects, simplifications by automated hints thereby improving the velocity of the proof developers, and other relevant factors) to convert into a measure of an amount of work, similarly to story points in Software Engineering as presented in Section 2.5.

The framework is not exclusive to a specific proof methodology, however, might not be fully applicable like in the case of refinement. It says nothing about equivalence proofs and the dilution of complexities between different refinement layers. Although, from the refined layer perspective, the framework could help to some extent.

The framework showed useful for our projects, but might not be suitable for all software projects. Indeed, the proof effort compared to the design and implementation effort is much larger and limits the feasibility of full formal verification. While our projects were designed for full formal verification from scratch, nowadays, proofs arrive late in the software development process, and despite automated proof tools, open research questions are the best manners to erase the barrier and reduce the cost between the world of code and proof. Our code/proof co-design approach (explained in Section 3.2) adapts to the software's lifecycle (and so to the inherent instability of such a project) with adaptative metrics and evolutive strategies proposed in the framework. Parallel activities for software and proof developers can still be performed without the need to wait for the other team (*e.g.* refactoring, unit tests, specification formulation, functionality extension). More than that, it is essential that they are involved together from the start to form a code basis suitable for formal verification (in fact, the same people worked both on the code design and the verification in both projects we report). In our projects, very few instruction reorderings took place during the co-design phase. However, for a current project not designed with a formal verification process in mind, our framework is still applicable but would certainly need structural and procedural simplifications (refactorings) to ease the formal verification process. Such cases might also require formal verification of only a subset of the initial project.

5 ILLUSTRATION OF THE MANAGEMENT OF PIP-MPU'S FORMAL VERIFICATION FROM THE PROOF DEVELOPER'S PERSPECTIVE

As an illustration of the theoretical foundations proposed in the previous section, we apply the framework to Pip-MPU. Pip-MPU provides a real-world example with a running prototype, whose formal verification had not started before this work. We give insights about the formal basis of Pip-MPU in Section 5.1. The actual proofs are out-of-scope of this

work, but available online[3] and have been presented in an earlier work [16]. We describe then the preliminary analysis of the proof structure (Section 4.2), the proof development strategy (Section 4.3) and the proof planning (Section 4.4).

5.1 Pip-MPU's formal basis

The formal verification process aims to verify the so-called security properties (the **specification properties**) which state the hierarchical partitioning model of the separation kernel. The security properties are *invariants*, *i.e.* properties that must be verified in whatever state that the system currently is. They extract information from Pip-MPU's metadata structures to construct the partitioning model on which they can reason about. Consequently, they can only be proven on a sane/correct partitioning model, which implies the metadata structures are correctly formed. **Consistency properties** ensure the latter. They check the correctness of the configuration of the metadata structures. The consistency properties are also invariants. The proof goal is then to verify the security *and* consistency properties.

As a reminder of Section 2, the security (specification) properties are composed of one memory-sharing property (Vertical Sharing *VS* ensuring memory owned by a child partition must be shared from its parent partition) and two isolation properties ensuring strict spatial partitioning (Horizontal Isolation *HI* ensuring strict isolation between sibling partitions, and Kernel Isolation *KI* ensuring strict isolation between user and kernel memory).

Pip-MPU has been designed with formal verification in mind. Several simplifications aiming to ease the verification process happened during the software development, and continued afterwards, as part of our code/proof co-design approach. Pip-MPU uses **Hoare logic** (cf Section 2) to formally verify the (currently) eleven kernel services, each of them composed of instructions and internal functions (sometimes shared between services, *i.e.* dependencies). Proofs are conducted in the **Coq Proof Assistant** [22]. The services are implemented in a shallow-embedded C subset in Gallina (the specification language of the Coq proof assistant). For that, it uses a monad, an abstract data type which represents computations (side-effects) by encapsulating a state and provides functions to access it. The monadic style is better suited for formal verification using the Hoare logic. The code of the services is later translated word-by-word from Gallina to compilable C.

From the architectural point of view, Pip-MPU dynamically creates a partition tree and manipulates kernel metadata structures to set it up. Each partition is defined by two types of metadata structures: 1) the *PDT* structure (*a.k.a* Partition Descriptor) holding most importantly a reference to its parent partition and 2) a super-structure composed of *BE* entries (*a.k.a* Block) each related to a configurable MPU region, *SHE* entries (*a.k.a* Shadow 1) each referencing a child partition, and *SCE* entries (*a.k.a* Shadow Cut) each keeping track of an MPU region division. Our memory model only deals with these types (*PDT*, *BE*, *SHE*, *SCE*) and the address type (*PADDR*).

The partition tree is extracted and used in the proofs as chained references between parents and children in the form of a generated list, which is abstract because they are not defined at verification time. Many other abstract lists are used during verification in the form of chained structures to represent linked lists or contiguous memory addresses, such as arrays and sublists. The link between the abstract lists and their concrete counterparts, *e.g.* from the correctness in the world of proofs to the verified code, is ensured by the verification of the consistency properties.

5.2 Proof structure

The formal verification intervenes after the design and implementation of Pip-MPU. We conduct a first analysis of the structure of the code and the projected derived proof structure.

[3] https://github.com/2xs/pipcore-mpu/tree/master/proof

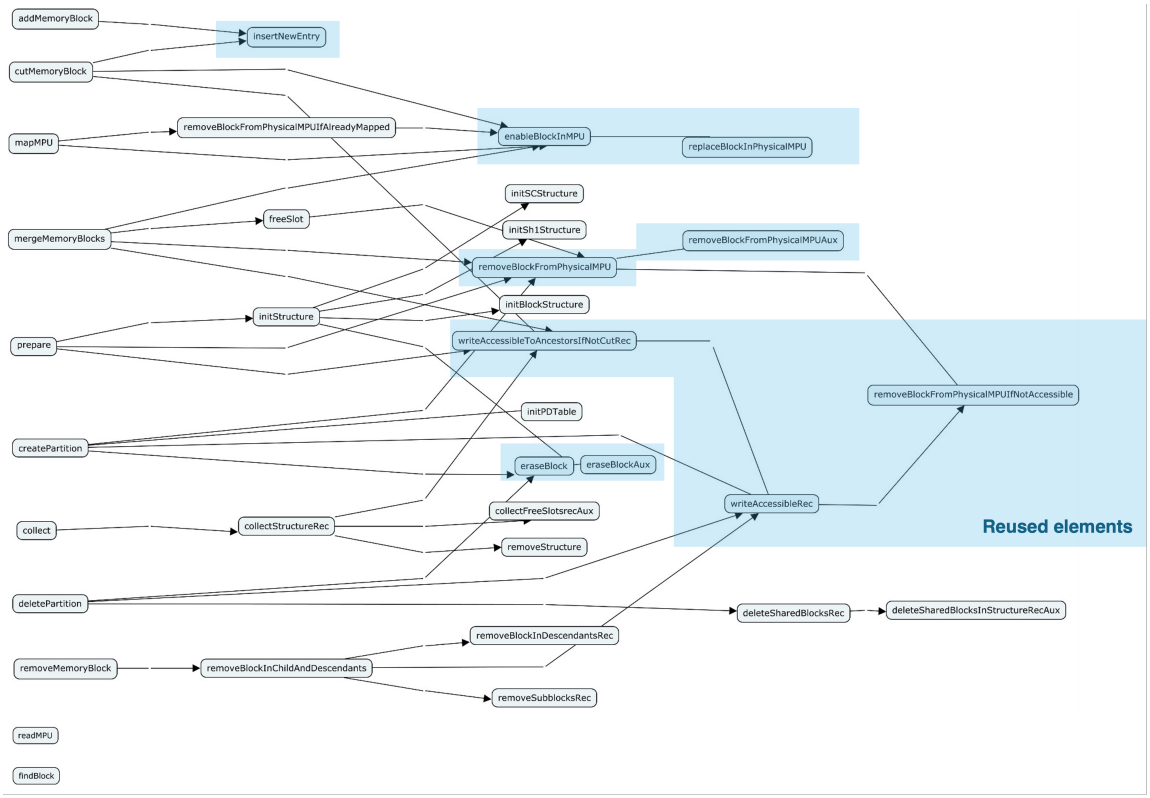

Fig. 7. Pip-MPU dependency graph. Service entry points are on the left. Reused elements are highlighted in the blue boxes.

5.2.1 Reused elements ratio in Pip-MPU. For the sake of clarity and readability, we only model in Figure 7 elements containing write instructions (the full dependency graph should also contain any read instructions-related elements). In this figure, reused elements are highlighted with blue boxes. Thus, the more blue boxes, the more reuse.

For Pip-MPU, the reused elements ratio, given the simplified dependency graph depicted in Figure 7, is:

$$RR = \frac{10}{36} \approx 27.7\%$$

It means between a fourth and a third of all code elements are reused.

5.2.2 Reused proof discrepancy in Pip-MPU. In Pip-MPU, the modularity of code is translated in the proofs, so the proof dependency graph is equivalent to the code dependency graph ($G_{function} = G_{proof} = G$). Thus, there are no differences between the actual proof reuse RR_{proof} and the code reuse RR_{code}, and so the reused proof discrepancy is 0. Equivalently, and likewise the previously computed code reused elements ratio, it means more than a fourth (27.7%) of the lemmas developed for the main proof elements (*e.g.* to prove the functions and primitive read and write operations) are reused. Thus, the proof process leverages the full potential of modularity. Further analysis of the code did not show any factorisation worth reducing even more the proof effort.

5.2.3 Effective reusability in Pip-MPU. We illustrate the benefit of the effective reusability (*ER*) metric with the Pip-MPU memory management services: `createPartition`, `deletePartition`, `prepare`, `collect`, `addMemoryBlock`, `removeMemoryBlock`, `cutMemoryBlocks`, `mergeMemoryBlocks`, `mapMPU`, `readMPU`, `findBlock`. Recall that the effective

Table 2. Effective reusability metric table for Pip-MPU.

Service ($r = 11$)	$\lvert SG_i'' \rvert$	$\lvert SG' \rvert$
addMemoryBlock	1	2
cutMemoryBlock	1	7
mapMPU	2	6
mergeMemoryBlocks	2	13
prepare	5	13
createPartition	2	9
collect	4	7
deletePartition	3	7
removeMemoryBlock	4	6
readMPU	1	1
findBlock	1	1
Total sum	26	72

U = $\lvert G \rvert$ = $\lvert G_{element\_reused} \rvert$ + 26 = 10+26 = 36

E = $\lvert F' \rvert$ = 72

Effective reusability : $\frac{U}{E} = \frac{36}{72} = 50\%$

reusability shows the actual amount of spared efforts compared to the naive approach without code or proof factorisation. In the following, we relate to the *ER*'s dividend as U and *ER*'s divisor as E; so $ER = \frac{U}{E}$.

For Pip-MPU, we study each service subtree and compute the values reported in Table 2. We obtain an effective reusability of 50%, which indicates a reduction of half of the number of proof elements to prove compared to the naive approach. In other words, Pip-MPU largely relies on reusable elements, which directly benefits the proof effort by reusing formerly proven lemmas.

5.3 Proof development strategy establishment in Pip-MPU

In this section, we describe how we used the tools and metrics described in this article to define and decide the proof development strategy to prove Pip-MPU. After the breadth-first exploration stage, 19 consistency properties were found. After completion of the first priority-based iterative exploration on service `addMemoryBlock`, 10 more properties were discovered (full set of properties given in Appendix A). Notably, the proof of this latter service only relies on half of the 29 discovered consistency properties. This means that without the breadth-first exploration strategy, only fourteen consistency properties would have been discovered, thus misleading developers and project managers on the scope of work. The gap between the properties discovered on the flow and the final total number is a strong source of uncertainty as it may impact heavily the proof effort. It is also a source of frustration for managers to see the workload growing bigger without an idea of completion (see Section 6 for the project manager's perspective). Furthermore, Pip-MPU derives from Pip and a certain degree of similarity is expected in the consistency properties. The parent project Pip almost completed full verification and features 25 consistency properties, with some spared properties compared to Pip-MPU due to the lighter requirements of the underlying hardware. As the number of properties is similar, the exploration strategy managed to capture the majority of consistency properties as expected.

Furthermore, the rapid priority-based iterative strategy allowed us to discover a software bug endangering the specification (security) properties very early in the proof process. While bugs were not expected at that point because of thorough development and reduced code, it had the positive aspect to validate the purpose of this strategy.

More than that, the establishment of the strategy leveraged a specific code and proof branching workflow, close to the *Gitflow* branching model. The general idea is to introduce an equivalent proof *master* branch, rooting the proof

branches of all services. Each time a set of common lemmas have been developed, they are merged into the proof *master* branch, which is then dispatched in all proof service branches to retrieve the fresh and complete set of common lemmas there. Once a service has been fully formally verified, the corresponding proof branch is merged into the proof *master* branch, which in turn is finally merged into the code *master* branch to keep both *master* branches in sync. This workflow enables parallelising the work and eventually assigning it to several (code/proof) developer teams.

5.4 Proof planning for Pip-MPU

The application of the framework for Pip-MPU enables to plan the proof development in advance and reveals the proof attributes for this particular system.

5.4.1 Proof complexity computation. A fine-grained analysis of the program instructions is essential to compute the proposed proof complexity, because they differ in terms of impact on the properties. For example, in Pip-MPU, a modification of the access permission rights is not decisive for the security properties. Indeed, solely the access, whatever the permissions, is considered. On the contrary, setting up the flag which references a new child, impacts both the consistency properties and the security properties because it modifies the partition tree.

We developed a heuristic to approach the proof complexity by knowing four function characteristics that give a minimal sense of the proof effort: 1) the number of instructions, 2) the instruction modifies the memory state, 3) the instruction calls an inner function, 4) the function is recursive (because a loop invariant must additionally be found). The list is not exhaustive and additional factors impacting the proof effort can easily agglomerate with the framework described in previous Section 4.

In this section, we finely analyse the proof complexity of the `addMemoryBlock` service of Pip-MPU.

Type and field impact scores on properties. We classify which modifications impact the consistency and security properties the most by an analysis of the fields used in the properties.

If the modification modifies a value used in the property, then a score of 1 is attributed to the corresponding entry of the *IIP* matrix. If the modified value lies within a list, then more efforts are expected to trace the modification compared to the initial list, requiring the finding of a loop invariant (lists are defined recursively) and the generalisation for abstract values, which can be very cumbersome and which complexity is list-dependent. However, our experience shows that it is possible to create a reusable lemma which reports the list's modification for a single instruction step and to apply the lemma at the modification instruction to propagate the new list. Hence, *given some amount of work to create this lemma*, the modification does not impact as much as foreshadowed anymore. Thus, modification instructions involving lists are counted *separately* with a "*" (*star*) per list, which informs that a certain amount of proof work must be done once, but can be immediately reused each time it is encountered and then "*" instantiates in a score of 1. Furthermore, according to our experience, the number of modified lists used across different properties is limited. There will still be some proof work to adapt to local situations and locally prove consistency properties, however on a unique state modification (contrariwise to the proofs of the consistency properties at function-level that could include many state modifications).

Table 3 resumes what impact is expected on the proof of properties by the modification of a single instruction. This table sums up the individual tables present in Appendix A.

From Table 3, we easily identify the most determinant fields for the security properties, notably the `PDflag` (stating that the entry refers to a partition descriptor), the memory types (*PDT*, *BE*, *SHE*, *SCE*), the `structure` field (the pointer

Table 3. Illustration of the type *TFPC* and field impact score matrix on the proof of Pip-MPU.

		Impact score on consistency properties	Impact score on security properties
Type	PDT type	14 + 5*	3*
	BE type	16 + 10*	3*
	SHE type	6 + 2*	3*
	SCE type	3	0
	PADDR type	7 + 11*	3*
Field	pdentry.(firstfreeslot)	2 + 3*	0
	pdentry.(structure)	3 + 9*	3*
	pdentry.(nbfreeslots)	1 + 3*	0
	pdentry.(parent)	2	0
	pdentry.(MPU)	1	0
	blockentry.(blockindex)	5	0
	blockentry.(blockrange).(startAddr)	4 + 5*	3*
	blockentry.(blockrange).(endAddr)	2 + 7*	3*
	blockentry.(present)	6 + 7*	3*
	blockentry.(accessible)	4 + 2*	1*
	blockentry.(read)	2	0
	blockentry.(write)	2	0
	blockentry.(execute)	2	0
	sh1entry.(PDflag)	5 + 1*	3*
	sh1entry.(PDchild)	3	0
	sh1entry.(inChildLocation)	3	0
	scentry.(origin)	2	0
	scentry.(next)	2	0

to the super-structure from the *PDT* structure) and the `accessible` flag (the *BE* entry can be used). Considering as well the consistency properties, we conclude that any modifications of the `present` flag (the *BE* entry exists) and the `structure` field (the pointer to the super-structure from the *PDT* structure), consolidated by the score on *BE* and *PDT* types, as well as the `PDflag`, impact the properties the most from a global perspective, and by extension the proofs. On the contrary, as expected, modifications of the access permissions do not disturb the properties very much (low scores).

Impact score of `addMemoryBlock`. We illustrate the metric with the `addMemoryBlock` service. The service consists of 29 instructions in its main entry point, which include the inner function `insertNewEntry` and two single modification instructions, the remaining being instructions not modifying the state.

First, we compute the impact score of `insertNewEntry`. It has 14 instructions, of which 10 are modification instructions impacting the fields *pdentry.(firstfreeslot)*, *pdentry.(nbfreeslots)*, *blockentry.(blockrange).(startAddr)*, *blockentry.(blockrange).(endAddr)*, *blockentry.(accessible)*, *blockentry.(present)*, *blockentry.(read)*, *blockentry.(write)*, *blockentry.(execute)*, *scentry.(origin)*. By summing all impact scores of these modification instructions given Table 3, we obtain a total score of: (2+3*) + (1+3*) + (4+5*+3*) + (2+7*+3*)+ (6+7*+3*)+(4+2*+1*)+2+2+2+2=27+37*.

Then, we compute the impact score of the main service function `addMemoryBlock` by calculating the impact score of the remaining two single modification instructions (so excluding `insertNewEntry` computed independently above). These two instructions impact the fields *sh1entry.(PDchild)* and *sh1entry.(inChildLocation)*. The impact score of the main service function only is then 3+3=6.

Finally, we get the cumulative impact score of 33+37* for `addMemoryBlock`. This score is definitely higher than a service without any modification instructions, like `findBlock`, which has an impact score of 0.

Property complexity matrix. We compute the property complexity matrix depending on its intrinsic properties. We retain four property characteristics that elevate the property complexity: 1) the size of the property (the number of sub-propositions) 2) the number of different lists 3) the number of variables involved 4) the number of final proof goals (the size of the last sub-propositions). Concerning the size of the property, it has a direct influence on the proof context, which must be adapted and transformed to solve the proof goal. About lists, they induce a higher difficulty level because they abstract a set of linked elements that must be monitored at any time, and not only a single one like a field modification. Hence, the modification of the state could produce a significant disruption in the initial set. However, our experience shows that the proof impact could be reduced by setting up reusable lemmas which state the expected list modifications for a given operation. After the creation of the reusable lemmas, only the number of different lists implies additional proof work to obtain the modified state with the corresponding updated lists. Regarding the number of variables, they imply additional combinations in the proof script. For example, if the proof concerns two elements, the different combinations of these elements must be explored. Finally, *à propos* the number of final goals, it states the number of proofs to provide to solve the global proof goal.

The matrix is represented in Table 11 of Appendix A. The table shows heterogeneous complexities, with higher complexities for the security properties, and for the properties *freeSlotsListIsFreeSlot*, *DisjointFreeSlotsLists*, *accessibleChildPaddrIsAccessibleIntoParent* and *sharedBlockPointsToChild* (details about the properties are not required for the reading flow, but the curious reader will find their definitions in the available proofs[4]), which means more case discriminations in the proof script, more lists, richer proof context and more proof goals that complicate the proof.

Type complexity. The function `insertNewEntry` manipulates three different types: *PDT*, *BE* and *SCE*. For each type, only one memory address is associated, so three different addresses. Hence, the part of the function proof complexity due to the types is $TIS[PDT] + TIS[BE] + TIS[SCE]$ =33 + 21*. The computation is reported in Table 4.

Proof complexity. Finally, we are able to compute the proof complexity of `addMemoryblock`. Recall from Section 4 that the proof complexity of an instruction is the corresponding transposed column of that instruction of the impact score matrix multiplied by the total column of the property complexity matrix.

The function `insertNewEntry` has a proof complexity of $PC_{insertNewEntry} = 253 + 439^*$ as computed in Table 4 from *IIP* Tables [7, 8, 9, 10], *TFPC* Table 3 and *PCM* Table 11, if we consider all consistency properties (again sum of transposed columns in the instruction impact matrices multiplied by the total column of the property complexity matrix, summed with the type function proof complexity, as defined in Section 4). Each instruction corresponds indeed to a single field modification. The service's modification instructions in its main function are `writeSh1PDChildFromBlockEntryAddr`, which modifies the PDchild field of the Shadow 1 entry (a part of the kernel structure holding the references to the child partitions), and `writeSh1InChildLocationFromBlockEntryAddr`, which modifies the InChildLocation field of the same Shadow 1 entry. They have, respectively, a proof complexity of $PC_{sh1entry.pdchild} = 6 * 1 + 12 * 1 + 19 * 1 = 37$ and $PC_{sh1entry.inChildLocation} = 6 * 1 + 6 * 1 + 12 * 1 = 24$ (*i.e.* respectively the transposed columns *sh1entry.PDchild* and *sh1entry.inChildLocation* of Table 10 multiplied by the column "Total" of Table 11). `addMemoryBlock`'s main function proof complexity is then the sum of the two computed proof complexities: $PC_{addMemoryBlock} = 37 + 24 + 253 + 439^* = 314 + 439^*$.

[4] https://github.com/2xs/pipcore-mpu/tree/master/proof

Table 4. Illustration of the computation of `insertNewEntry`'s proof complexity.

Instruction proof complexity	
Instructions	Proof complexity
writePDFirstFreeSlotPointer	13 + 33*
writePDNbFreeSlots	7 + 33*
writeBlockStartFromBlockEntryAddr	35 + 80*
writeBlockEndFromBlockEntryAddr	16 + 110*
writeBlockAccessibleFromBlockEntryAddr	33 + 29*
writeBlockPresentFromBlockEntryAddr	44 +96*
writeBlockRFromBlockEntryAddr	18
writeBlockWFromBlockEntryAddr	18
writeBlockXFromBlockEntryAddr	18
writeSCOriginFromBlockEntryAddr	18
Sub-total	220 + 381*
Type function proof complexity	
Number of different PDT type addresses: 1	14+8*
Number of different BE type addresses: 1	16+13*
Number of different SHE type addresses: 0	0
Number of different SCE type addresses: 1	3
Number of different PADDR type addresses: 0	0
Sub-total	33 + 21*
Total	253 + 439*

We have now a fine-grained understanding of the proof complexity in Pip-MPU for the `addMemoryBlock` service. The service's proof complexity score serves as a temporary goal, allowing for the computation of estimated proof effort, which can then be translated into time and tasks.

5.4.2 Long-term proof planning. Long-term proof planning provides managers with an optimised roadmap for the formal verification process of their software projects. We illustrate the Algorithm 1 with Pip-MPU's proof path strategy in Table 5. It eventually selects the order in which to prove the elements.

For the sake of simplicity, we approximate the proof effort calculations. In this section, we define the current proof effort on a proof path as the total number of proof elements included in that path. The next proof effort is also simplified and computed as a sum of all incoming edges on a proof element minus its own incoming edge on the proof path (only keeping the additional incoming edges). Indeed, as described earlier, incoming edges in a proof dependency path represent reusable lemmas that also benefit other proof elements.

With each iteration, the current proof effort decreases for the other services only if they share reusable lemmas with the last selected service, which would have been proven at that earlier stage. Note that this illustration serves as an example; a more accurate measurement of the proof effort, such as the estimated proof effort *EPE* introduced in Section 4 and detailed in Section 5.4.1, could provide deeper insights. In such a case, for instance at iteration 9, we would have selected `removeMemoryBlocks` instead of `collect` because the proof complexity analysis would have shown a higher computed proof effort.

5.4.3 Proof dashboard. We illustrate the dynamic nature of the proof dashboard and its three distinct views (as introduced in Section 4.4) by presenting a snapshot during the proof development. For the example, we take the instant

Table 5. Proof path best effort strategy iterations of Algorithm 1 applied to Pip-MPU. Each iteration reveals a new score depending on previously selected services. The score is given as a tuple: *current_proof_effort*/*next_proof_effort*. A service is marked with '°' at the iteration when it is selected. Final service order: [`readMPU`, `findBlock`, `addMemoryBlock`, `mapMPU`, `cutMemoryBlock`, `mergeMemoryBlocks`, `createPartition`, `deletePartition`, `collect`, `removeMemoryBlock`, `prepare`]

Iteration	addMemoryBlock	cutMemoryBlock	mapMPU	mergeMemoryBlocks	prepare	createPartition	collect	deletePartition	removeMemoryBlock	readMPU	findBlock
1	2/1	7/11	4/3	9/10	12/8	8/9	7/7	7/6	6/4	1/0°	1/0
2	2/1	7/11	4/3	9/10	12/8	8/9	7/7	7/6	6/4	0/0	1/0°
3	2/1°	7/11	4/3	9/10	12/8	8/9	7/7	7/6	6/4	-	0/0
4	0/0	6/10	4/3°	9/10	12/8	8/9	7/7	7/6	6/4	-	-
5	-	4/7°	0/0	6/1	12/8	8/9	7/7	7/6	6/4	-	-
6	-	0/0	-	4/3°	7/3	6/4	4/0	5/1	4/0	-	-
7	-	-	-	0/0	5/0	4/1°	4/0	5/1	4/0	-	-
8	-	-	-	-	5/0	0/0	4/0	3/0°	4/0	-	-
9	-	-	-	-	5/0	-	4/0°	0/0	4/0	-	-
10	-	-	-	-	5/0	-	0/0	-	4/0°	-	-
11	-	-	-	-	5/0°	-	-	-	0/0	-	-
12	-	-	-	-	0/0	-	-	-	-	-	-

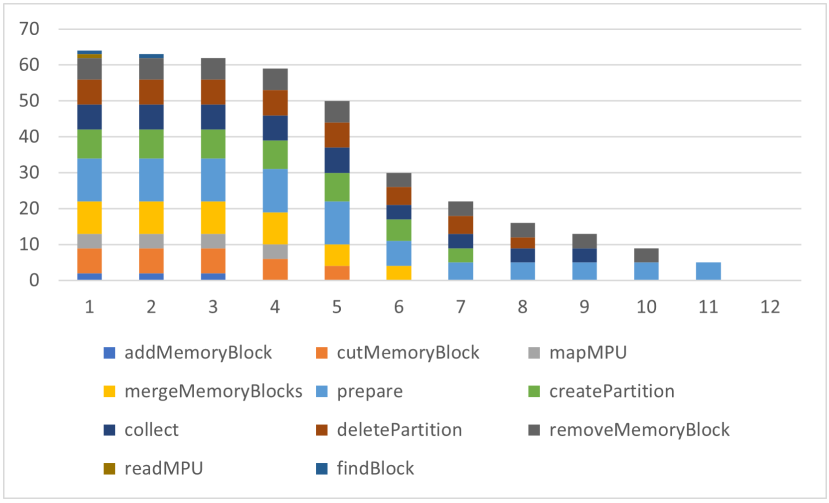

Fig. 8. Planned evolution of the remaining proof effort at each iteration of Algorithm 1 applied to Pip-MPU (snapshot of the remaining current proof effort at each iteration from Table 5). Observe the smooth curve progressively decreasing the proof effort. Furthermore, note how the proof of the services `mapMPU`, `cutMemoryBlock`, and `mergeMemoryBlocks`, respectively at the fourth, fifth and sixth steps, do not show significant proof effort individually but drastically reduces the overall proof effort.

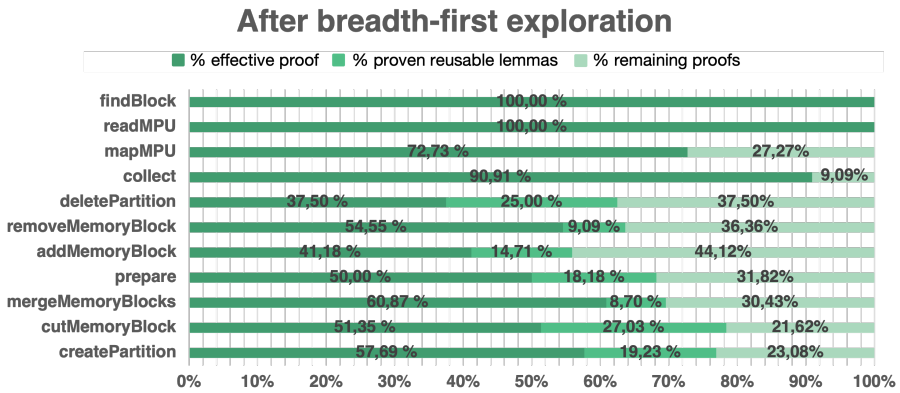

Fig. 9. *Instruction-level view* of the proof dashboard after the breadth-first exploration stage approximated by proof elements (not considering the proof complexities).

Table 6. *Resolution view* of the proof dashboard of the `addMemoryBlock` service and its internal element `insertNewEntry` after the breadth-first exploration and after the complete priority-based iterative exploration for this service.

	After breadth-first exploration	After complete priority-based iterative exploration
addMemoryBlock		
Property verification ratio (in %)	0	100
Number of remaining admitted lemmas	0	0
insertNewEntry		
Property verification ratio (in %)	0	100
Number of remaining admitted lemmas	0	0

after the end of the breadth-first exploration, so after all check code portions of the services have been visited (see Section 4.3). At this stage, the verification process for the services `findBlock` and `readMPU` is complete because they consist solely of check instructions (read operations).

In the first view of the dashboard, at the example snapshot, the dependency graph coverage would look a lot like Figure 7 completed with read instructions, where only `findBlock` and `readMPU` would be displayed as fully verified together with some check functions.

The second view of the dashboard is the instruction-level progress. The result is a global advancement report illustrated in Figure 9 for the example snapshot, which demonstrates the transversal effects of the breadth-first strategy. The amount of reusable lemmas is directly linked to the extent of proof reuse and the leveraged effective reusability metrics computed above.

The last view of the proof dashboard, the Resolution view, shows no progress *at the given snapshot* because the services have not yet reached their final instruction, as illustrated in Table 6 for the `addMemoryBlock` service, except for the two fully-verified services where the property verification ratio would be complete with no remaining admits to report.

5.5 Discussion

The fine-grained proof complexity scores could be improved to be as close as possible to a realistic expected proof effort. To take an example, in all our scores, we considered additional instructions to elevate the complexity. This might not always be true. As an illustration, the simple case where an instruction writes another value at some address and just after writes back the initial value at the same address again, does not modify the state at the end, however disturbs the proofs and properties. Furthermore, we base our property complexity matrix on selected characteristics. The framework welcomes new heuristic factors, should further experiences on verification projects identify additional ones.

The computed scores also reflect a particularity of our model which considers lists. The issue comes from low-level code verification and the discrepancy between the time of verification and the time of execution. Indeed, there is no other component like a garbage collector that is responsible for the consistency of the memory operations, which becomes consistency properties to prove. At execution time, lists are defined and finite. At verification time, lists are abstract types that are defined recursively to represent a set of contiguous memory addresses or chained pointers. In the memory model, whenever a value is stored, it is important to ensure that any modifications made to the defined lists are expected and do not erase data elsewhere. To achieve this, the proofs need to account for all impacts of any modification. However, the verification process involving lists can be made more efficient by making the proofs more generic, which we characterised by the "*" for each impacting list. The "*" refers to the significant one-time effort required to produce a generic lemma, which will have a minimal impact on the proof later on, similar to an atomic modification in a structure. To further improve the use of the framework, the recomputation of the proof complexities could occur after each completed proof of a service. As a result, the selection order of the algorithm will change over time to best fit the state of the proofs. In scenarios stemming from other use cases, other data objects could also require special treatment and be considered independently to manage complexity more efficiently.

6 ILLUSTRATION OF THE MANAGEMENT OF PIP-MPU'S FORMAL VERIFICATION FROM A PROJECT OVERSIGHT PERSPECTIVE

This section describes how we leveraged some of the metrics and tools defined in the previous sections to empower project managers with a familiar dashboard, namely a Burnup chart, that allows them to take a step back on the formal proof activities, to understand and convey the difficulties encountered, and to take strategic decisions for the formal proof perimeter and activities. First, we will describe how we built our dashboard. Second, we provide feedback on its use.

6.1 Building the management dashboard

We identified three major needs in terms of project management. The first one is to define the scope of the work to be done. The second one is to show and explain the progress of the proof process. Finally, the third one consists in analysing whether the observed progress is satisfactory and deciding whether an alert needs to be raised to request a delay, reduce the scope of work, or request additional resources.

We were able to address both first needs with the tools presented in the previous sections. To define the scope of the project, we used the code dependency graph and its mapping to the proof dependency graph, illustrated in Figure 7, because it captures the overall scope of the work to be done. It is also an intuitive tool familiar to developers and project managers. To track and communicate the progress made in the proofs, we used the proof dashboard views,

with the *dependency graph coverage view* giving the overall proof status, while the other views provide an in-depth view of the status of the proof with the *instruction-level progress view* illustrated in Figure 9 giving the proven code instructions and the *resolution view* giving the ratio of verified properties and the number of remaining admitted lemmas.

However, these metrics are not sufficient to determine whether the project is on track, ahead or behind schedule. They also do not capture the fact that new properties can be found during the proving activity which greatly impacts the visibility and effort required to complete the proof. This situation is similar to the scope creep problem which, in project management, refers to a situation where there is an uncontrolled growth of the work to be done for example when key project stakeholders continuously change requirements. To visualize and anticipate such issues, a typical solution is to use Burnup charts as they display at the same time the scope of work and the progress made to achieve this scope of work.

Figure 10 illustrates the Burnup chart for our project. The project spanned over 16 months, experiencing a three-month interruption during months 4, 5 and 6, followed by an additional pause in month 11. The orange curve labelled **Total Proof Scope** corresponds to the total number of properties that need to be proven to achieve full proof completion. We define it as $TPS = N_S \times (P_C + P_S)$, where N_S is the number of services, P_C and P_S respectively are the number of consistency properties and the number of security properties. The black curve, labeled **Project Proof Scope**, corresponds to the number of properties PPS that the project team commits to prove during the project's duration. We define it as $PPS = N_{S,Perim} \times (P_C + P_S) \times R_A$, where $N_{S,Perim}$ is the number of services in the perimeter of the proof activity and R_A is an *admit* factor expressed as $R_A = \exp(-\frac{N_A}{TPS})$. It captures how adding admits to the proof reduces the scope of what needs to be proven in the project. Its value stands between 0 and 1 depending on whether there are many or few admits. Therefore, it can also be seen as an evaluation of how the admits affect the proof. The blue and dark green curves, namely **Total Proof Ideal** and **Project Proof Ideal**, respectively correspond to the monthly targets for the number of properties that must be verified to fully complete the proof and to meet the project's defined proof scope within the allotted time. The green histograms labelled **Proven Properties** correspond to the number of properties fully proved (the Qed has been achieved). The blue histogram, labeled **Estimated Proven Properties**, is an estimation of the properties partially proven, taking into account the number of proven lemmas that can be reused, as detailed in Section 5.4.3. We define **Estimated Proven Properties** as $PPP = MPC \times (N_{PI} + N_{APIP})$, where MPC is the mean of the proof coverage of each service as visualized in the Instruction-level progress view from Figure 9, N_{PI} is the number of fully proven instructions and N_{APIP} is the number of already proven instructions as described in Section 5.4.3.

6.2 Retrospective

During the project, the dependency graph and the coverage metric were used to communicate the advances of the project towards the management of the industrial partner. The coverage metric was also integrated as a Key Performance Indicator (KPI) of the industrial project team and used to set the variable part of the compensation.

The Burnup chart was introduced later in the project and was not used to strategically drive the project. However, Figure 10 perfectly illustrates the story of the project and the decisions taken at key points of the project. The initial step of the methodology presented in this article involves exploring the proof in breadth to swiftly identify all the properties requiring verification. This phase corresponds to the first month of the project where marked by a sharp rise in the number of properties in the Total Proof Scope. Then, as the in-depth exploration of the proof of one service starts, the number of the properties in the Total Proof Scope evolves when consistency properties are merged (for

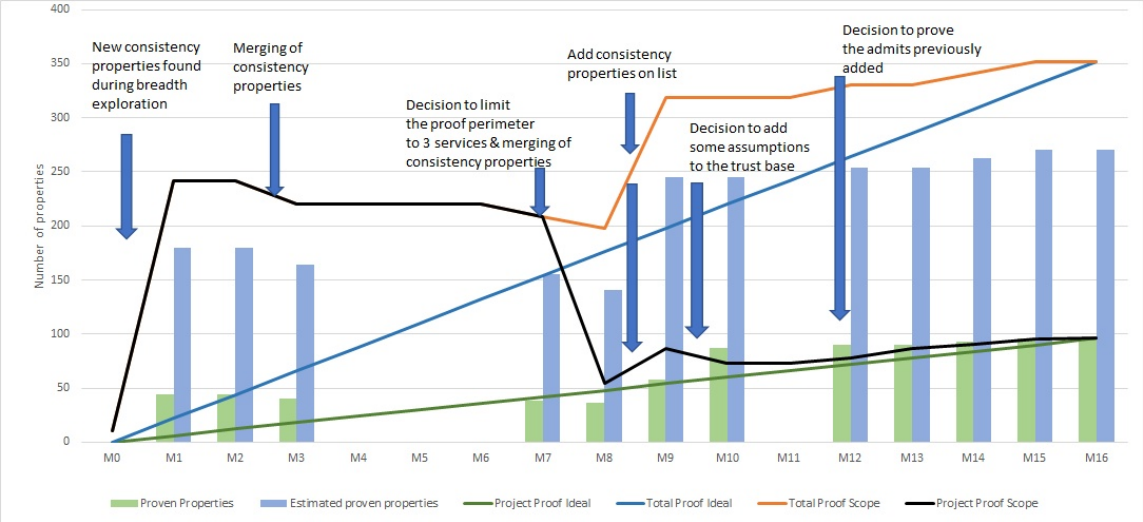

Fig. 10. Burnup chart of the project

example between months 2 and 3) or when new consistency properties are added (for example between months 8 and 9). Prior to month 7, the Total Proof Scope and the Project Proof Scope coincide. However, given that the project was initially due to month 10 and that the Proven Properties were below the Project Proof Ideal and that this gap was further exacerbated by the discovery of new consistency properties, we decided in month 7 to limit the perimeter of the code to be proven to 3 services instead of 11. This corresponds to the moment when the Project Proof Scope and the Total Proof Scope become distinct. As the month 10 deadline approached, we decided to increase the trust base of the proof by adding admits which helped achieve the Qed of the third service by the planned deadline, but also weakened the proof. Afterwards, we obtained an extension of the project to complete the proof of the third service, which was achieved in month 16.

7 RELATED WORK

The drive towards systematic procedures is ever-present in the proof development process of formal verification projects.

It is a time-consuming and tedious task, overwhelming the software development part, and there is great need and room for optimisations. The call for better empirical software engineering dates back from the 1970s, but it is only recently that optimisations, metric tracking, productivity and related research questions [23], to which outputs of this work partially contribute, have emerged alongside the formal verification of (small-to-large-scale) complex industrial projects.

Proof process understanding has been investigated before as an iterative trial and error process [8, 13]. However, to the best of our knowledge, no other work categorizes the consequences of decisions in the process so finely and uses them as guidelines to develop techniques and tactics to reduce the overall proof effort, even without proof expertise.

7.1 Metrics

Typical metrics for proofs in formally verified projects, like those presented in [12, 18, 23, 26, 28], include the number of lines of proof, number of theorems, number of invariants, number of discovered bugs, and amount of verified code.

In [11], the authors try to identify proof characteristics to quantify proof effort (number of theorems weighted by complexity, theorem size and number of dependencies) or to detect potential issues (depth of the dependency tree and number of child theorems that could complicate maintenance, similarity score between theorems revealing a bad design). Many of these metrics are sensitive to the interactive theorem prover used, the project, the proof style and the proof developer, which poses a risk of misinterpretation when making comparisons. Hence, the set of meaningful formal metrics has not been standardized and varies depending on the project. In addition to that, prior works [8, 26, 40] have already discussed the limitations of these metrics in accurately capturing the proof difficulty.

seL4 projects aim to build a reliable estimation model of the proof effort as described in *Simulation Modeling of a Large-Scale Formal Verification Process* [43], by combining a quadratic correlation between the specification size and the corresponding proof script presented in *Empirical Study Towards a Leading Indicator for Cost of Formal Software Verification* [31], with prior seL4 projects, notably *Productivity for Proof Engineering* [40] where they found a linear correlation between the proof size and the proof effort. In comparison, our leading parameters for proof effort estimation are distinguished by the granularity of the proof complexity computation and from the approach itself. Indeed, our experience has shown that we must introduce additional parameters to score the proof difficulty, in addition to the computed proof size or the raw statement size similar to cited works, which are the number of variables, the number of lists and the number of different types involved. Furthermore, we examine proof complexity not only from the perspective of the proof itself but also from the code. Therefore, we conceptualize proof complexity as a disturbance stemming from the code on the properties to prove and their inherent complexities, which directly translates i in a more challenging proof script. This approach combining proof and code complexity score is not considered in any works we are aware of. In addition to that, our fine-grained proof complexity analysis distinctly highlights proof artifacts at high risk of proof effort drifting, which we identify as a contributing factor to the quadratic correlation described above.

Moreover, these works analyze completed proofs with the assumption that the proof cost will remain consistent for new projects. Thus, computations are performed *post*-verification to help estimate future proof costs. Deltas with the estimation, caused by unplanned events such as new invariants to prove, are resolved with the adjunction of a "noise" [31]. While this fits well with a new project with a large common code base, for example porting to a close architecture, this makes it difficult to plan when much more changes are expected or because the expectations of a simple adaptation turn out to be vain. Unlike these works, our framework is orthogonal and takes place *before* any proofs are conducted, not considering or relying on the performances of other projects. Furthermore, our proof complexity metric can be computed independently from the interactive theorem prover, even applied on pseudo-code. Hence, our approach is agnostic to past experience and project-independent, making it well-suited for projects at their inception.

7.2 Management practices and tools for experts and non-experts

Beyond metrics, the framework proposes proof practices that are understandable to software developers and managers, so agnostic to proof expertise, while previously mentioned works essentially target proof developers or managers with skills in formal verification. As such, we do address productivity for proof developers and give practical solutions to some of raised concerns [8, 13, 23], with the intent to reach out more broadly with comprehensive know-how to non-experts. We believe the above metrics contribute to open research questions identified in *Large-Scale Formal Verification in Practice: A Process Perspective* [8], where the following quote is confirmed by the rest of our contributions and discussed below: *"Better metrics are required. Related to this, better estimation models are required for formal verification projects. These are both open questions. Answering them will help to bring about decision-making tools for formal verification project management,[...]"*.

In *Challenges and experiences in managing large-scale proofs*, Bourke *et al.* says *"the faster the discovery, the smaller the overhead"* when discussing disrupted proof processes due to the discovery of new invariants during the development. Our work directly responds to this desire with our breadth-first strategy, which intends to quickly identify the vast majority of invariants from the first stage of the verification. They also present the verification process as an iterative one, with wishes to fast-check deviating verification: *"Verification productivity hinges on this edit-check cycle being short."* We respond with our priority-based iterative strategy, which offers a solution for swiftly testing our assumptions and specifications by clearing off bugs and other highly risky inconsistencies, before delving deeper into the proof. This strategy also facilitates turnovers in the team members with a mix of long-lasting members and newcomers, highlighted as an important factor in [8, 13, 43]. Andronick *et al.* also identifies as important the ability to parallelize the process, which is eased by the strategy as well as the Git proof branching model we employ. This aligns with agile software development principles by facilitating continuous integration, and our proof complexity score resembles estimation points used to plan ahead a batch of work. To some extent, Ringer *et al.* [37] also point out similarities between the proof and the software development, by gathering hints and practices for Proof Engineering, that also extend to proof maintenance and proof repair [36]. They investigated proof development for proof developers, with a close focus on the interactive theorem provers and their interactions with them from broad projects. The issue of scaling-up verification projects is also previously pointed out by De Millo *et al.*, with difficulties arising from changing software (*Patches, ad hoc constructions, bandaids and tourniquets, bells and whistles, glue, spit and polish, signature code, blood-sweat-and-tears, and, of course, the kitchen sink*) [14]. However, we believe some critical pieces of software are worth formal verification and efforts, and our code/proof co-design approach, along with adaptable metrics, addresses evolutive software. Furthermore, the framework in our paper presents proof strategies not discussed yet in any other works we are aware of, especially from the high-level project management with a focus on proof development team managers. As stated previously, we do not consider the strategies to be optimal, even though we provide the "best" proof path strategy tool for some degree of optimisation and acknowledge the influence of expertise, collaboration and dedicated tools, as part of a "social process". Indeed, accumulated expertise from past projects is not required, but obviously reduces the likelihood of property fluctuations that we try to smooth with the breadth-first strategy. Rather, we provide a systematic procedure from code to Qed independent from the project which sets an upper bound to the proof work and delineates a proof path, from the first proof instants to completion. Reaching the last Qed of the project and continuing to monitor the proof status will consolidate these results.

At last, common metrics fall short of giving a precise overview of the proof development status, but instead, analyse the overall process from a higher point of view as illustrated in [8]. In this paper, we propose dynamic proof dashboard views for comprehensive and adaptative progress reports. This is crucial to communicate outside expert circles, in order to receive external funding and support from the hierarchy line in an industrial environment. Indeed, our project manager successfully extracted information to represent the progress with common project metrics, which is not yet present, to our knowledge, in the state-of-the-art. The ability to explain exactly where the proof stands within the "labyrinth" of the verification process, and makes it clear for non-experts, helps the wide adoption of the formal verification approach in the industry.

8 CONCLUSION

Full formal verification projects are still rare in the academic landscape, and even less in the industry. Indeed, past projects demonstrated heavy resource consumption, in terms of machines, time, and especially human effort and task difficulty guided by intuition and expert know-how. In this paper, we propose the rationalization of the proof

development process, often seen as a labyrinth for each code block to prove. We first discuss the proof-construction process given our proof/soft co-design approach. The process consists of reiterable steps to conduct the proof of a code block and get out of the labyrinth. From there, we identify the sources of important proof effort and we introduce proof management practices based on proof structuring, proof strategies, and proof planning, to reduce wasted proof efforts. Practices have been successfully applied to an operating system kernel with a formal verification approach. We compute metrics and leverage the tools presented in this paper in order to structure, plan, follow up, and construct proofs for projects using deductive verification in an efficient and appreciable way, thereby providing systematic guidance to reduce the reliance on the intuition and expertise of the proof developers. We also create a project management dashboard with a burnup chart which helps project managers to capture the proof status, drive it strategically and communicate about it. As such, proof developers gain in productivity and allow project managers, as well as other non-expert developers, to fully engage in the formal verification project. With systematic understanding and practices of proof development, Proof Engineering is facilitated and can take the turn for industrialisation for broader adoption.

A TYPE AND FIELD IMPACTS ON PIP-MPU'S CONSISTENCY AND SECURITY PROPERTIES

We identify which memory types or fields are used in the properties and bind them together, as illustrated in Tables [7, 8, 9, 10].

Table 7. Type changes impacts in Pip-MPU.

	Properties	PDT type	BE type	SHE type	SCE type	PADDR type
Consistency	nullAddrExists	-	-	-	-	1
	wellFormedFstShadowIfBlockEntry	-	1	1	-	-
	PDTIfPDFlag	1	1	1	-	-
	AccessibleNoPDFlag	-	1	1	-	-
	FirstFreeSlotPointerIsBEAndFreeSlot	1	1	1	1	1
	multiplexerIsPDT	1	-	-	-	-
	currentPartitionInPartitionsList	*	*	*	-	*
	wellFormedShadowCutIfBlockEntry	-	1	-	1	-
	BlocksRangeFromKernelStartIsBE	-	1	-	-	-
	KernelStructureStartFromBlockEntryAddrIsKS	-	1	-	-	-
	sh1InChildLocationIsBE	-	1	1	-	-
	StructurePointerIsKS	1	1	-	-	-
	NextKSIsKS	-	1	-	-	1
	NextKSOffsetIsPADDR	-	1	-	-	1
	NoDupInFreeSlotsList	1	*	-	-	*
	freeSlotsListIsFreeSlot	1	1	1	1	1
	DisjointFreeSlotsLists	1	*	-	-	*
	inclFreeSlotsBlockEntries	1	1	-	-	1
	DisjointKSEntries	1	1	-	-	1
	noDupPartitionTree	*	*	-	-	*
	isParent	1	*	-	-	*
	isChild	1	*	-	-	*
	accessibleChildPaddrIsAccessibleIntoParent	*	*	-	-	*
	noDupKSEntriesList	1	*	-	-	*
	noDupMappedBlocksList	1	*	-	-	*
	noDupUsedPaddrList	1	*	-	-	*
	sharedBlockPointsToChild	*	1	*	-	*
	wellFormedBlock	-	1	-	-	-
	MPUFromAccessibleBlocks	*	-	-	-	-
Security	verticalSharing	*	*	*	-	*
	partitionsIsolation	*	*	*	-	*
	kernelDataIsolation	*	*	*	-	*
	Total score	14+8*	16+13*	6+5*	3	7+14*

Table 8. Field change impacts on Partition Descriptor entry fields in Pip-MPU.

	Properties	pdentry.(firstfreeslot)	pdentry.(structure)	pdentry.(nbfreeslots)	pdentry.(parent)	pdentry.(MPU)
Consistency	nullAddrExists	-	-	-	-	-
	wellFormedFstShadowIfBlockEntry	-	-	-	-	-
	PDTIfPDFlag	-	-	-	-	-
	AccessibleNoPDFlag	-	-	-	-	-
	FirstFreeSlotPointerIsBEAndFreeSlot	1	-	-	-	-
	multiplexerIsPDT	-	-	-	-	-
	currentPartitionInPartitionsList	-	*	-	-	-
	wellFormedShadowCutIfBlockEntry	-	-	-	-	-
	BlocksRangeFromKernelStartIsBE	-	-	-	-	-
	KernelStructureStartFromBlockEntryAddrIsKS	-	-	-	-	-
	sh1InChildLocationIsBE	-	-	-	-	-
	StructurePointerIsKS	-	1	-	-	-
	NextKSIsKS	-	-	-	-	-
	NextKSOffsetIsPADDR	-	-	-	-	-
	NoDupInFreeSlotsList	*	-	*	-	-
	freeSlotsListIsFreeSlot	*	-	*	-	-
	DisjointFreeSlotsLists	*	-	*	-	-
	inclFreeSlotsBlockEntries	1	1	1	-	-
	DisjointKSEntries	-	1	-	-	-
	noDupPartitionTree	-	*	-	-	-
	isParent	-	*	-	1	-
	isChild	-	*	-	1	-
	accessibleChildPaddrIsAccessibleIntoParent	-	*	-	-	-
	noDupKSEntriesList	-	*	-	-	-
	noDupMappedBlocksList	-	*	-	-	-
	noDupUsedPaddrList	-	*	-	-	-
	sharedBlockPointsToChild	-	*	-	-	-
	wellFormedBlock	-	-	-	-	-
	MPUFromAccessibleBlocks	-	*	-	-	1
Security	verticalSharing	-	*	-	-	-
	partitionsIsolation	-	*	-	-	-
	kernelDataIsolation	-	*	-	-	-
	Total score	2+3*	3+12*	1+3*	2	1

Table 9. Field change impacts on Block entry fields in Pip-MPU.

	Properties	blockentry(blockindex)	blockentry(blockrange).(startAddr)	blockentry(blockrange).(endAddr)	blockentry(present)	blockentry(accessible)	blockentry(read)	blockentry(write)	blockentry(execute)
Consistency	nullAddrExists	-	-	-	-	-	-	-	-
	wellFormedFstShadowIfBlockEntry	-	-	-	-	-	-	-	-
	PDTIfPDFlag	-	1	-	1	1	-	-	-
	AccessibleNoPDFlag	-	-	-	-	1	-	-	-
	FirstFreeSlotPointerIsBEAndFreeSlot	-	1	-	1	1	1	1	1
	multiplexerIsPDT	-	-	-	-	-	-	-	-
	currentPartitionInPartitionsList	-	*	-	*	-	-	-	-
	wellFormedShadowCutIfBlockEntry	-	-	-	-	-	-	-	-
	BlocksRangeFromKernelStartIsBE	1	-	-	-	-	-	-	-
	KernelStructureStartFromBlockEntryAddrIsKS	1	-	-	-	-	-	-	-
	sh1InChildLocationIsBE	-	-	-	-	-	-	-	-
	StructurePointerIsKS	1	-	-	-	-	-	-	-
	NextKSIsKS	1	-	-	-	-	-	-	-
	NextKSOffsetIsPADDR	1	-	-	-	-	-	-	-
	NoDupInFreeSlotsList	-	-	*	-	-	-	-	-
	freeSlotsListIsFreeSlot	-	1	*	1	1	1	1	1
	DisjointFreeSlotsLists	-	-	*	-	-	-	-	-
	inclFreeSlotsBlockEntries	-	-	1	-	-	-	-	-
	DisjointKSEntries	-	-	-	-	-	-	-	-
	noDupPartitionTree	-	-	-	1	-	-	-	-
	isParent	-	-	-	*	-	-	-	-
	isChild	-	-	-	*	-	-	-	-
	accessibleChildPaddrIsAccessibleIntoParent	-	*	*	*	*	-	-	-
	noDupKSEntriesList	-	-	-	-	-	-	-	-
	noDupMappedBlocksList	-	-	-	1	-	-	-	-
	noDupUsedPaddrList	-	*	*	*	-	-	-	-
	sharedBlockPointsToChild	-	*	*	*	-	-	-	-
	wellFormedBlock	-	1	1	1	-	-	-	-
	MPUFromAccessibleBlocks	-	*	*	*	*	-	-	-
Security	verticalSharing	-	*	*	*	-	-	-	-
	partitionsIsolation	-	*	*	*	-	-	-	-
	kernelDataIsolation	-	*	*	*	*	-	-	-
	Total score	5	4+8*	2+10*	6+10*	4+3*	2	2	2

Table 10. Field change impacts on Shadow 1 and Shadow Cut entry fields in Pip-MPU.

	Properties	sh1entry.(PDflag)	sh1entry.(PDchild)	sh1entry.(inChildLocation)	scentry.(origin)	scentry.(next)
Consistency	nullAddrExists	-	-	-	-	-
	wellFormedFstShadowIfBlockEntry	-	-	-	-	-
	PDTIfPDFlag	1	-	-	-	-
	AccessibleNoPDFlag	1	-	-	-	-
	FirstFreeSlotPointerIsBEAndFreeSlot	1	1	1	1	1
	multiplexerIsPDT	-	-	-	-	-
	currentPartitionInPartitionsList	*	-	-	-	-
	wellFormedShadowCutIfBlockEntry	-	-	-	-	-
	BlocksRangeFromKernelStartIsBE	-	-	-	-	-
	KernelStructureStartFromBlockEntryAddrIsKS	-	-	-	-	-
	sh1InChildLocationIsBE	-	-	1	-	-
	StructurePointerIsKS	-	-	-	-	-
	NextKSIsKS	-	-	-	-	-
	NextKSOffsetIsPADDR	-	-	-	-	-
	NoDupInFreeSlotsList	-	-	-	-	-
	freeSlotsListIsFreeSlot	1	1	1	1	1
	DisjointFreeSlotsLists	-	-	-	-	-
	inclFreeSlotsBlockEntries	-	-	-	-	-
	DisjointKSEntries	-	-	-	-	-
	noDupPartitionTree	-	-	-	-	-
	isParent	-	-	-	-	-
	isChild	-	-	-	-	-
	accessibleChildPaddrIsAccessibleIntoParent	-	-	-	-	-
	noDupKSEntriesList	-	-	-	-	-
	noDupMappedBlocksList	-	-	-	-	-
	noDupUsedPaddrList	-	-	-	-	-
	sharedBlockPointsToChild	1	1	-	-	-
	wellFormedBlock	-	-	-	-	-
	MPUFromAccessibleBlocks	-	-	-	-	-
Security	verticalSharing	*	-	-	-	-
	partitionsIsolation	*	-	-	-	-
	kernelDataIsolation	*	-	-	-	-
	Total score	5 + 4*	3	3	2	2

Table 11. Property complexity matrix *PCM* in Pip-MPU.

	Properties	Size of property	Number of lists	Number of variables	Number of final proof goals	Total
Consistency	nullAddrExists	1	0	0	1	2
	wellFormedFstShadowIfBlockEntry	2	0	1	1	4
	PDTIfPDFlag	2	0	2	4	8
	AccessibleNoPDFlag	4	0	2	1	7
	FirstFreeSlotPointerIsBEAndFreeSlot	2	0	2	2	6
	multiplexerIsPDT	1	0	0	1	2
	currentPartitionInPartitionsList	1	1	0	1	3
	wellFormedShadowCutIfBlockEntry	2	0	1	2	5
	BlocksRangeFromKernelStartIsBE	3	0	2	1	6
	KernelStructureStartFromBlockEntryAddrIsKS	3	0	2	1	6
	sh1InChildLocationIsBE	3	0	2	1	6
	StructurePointerIsKS	2	0	2	1	5
	NextKSIsKS	5	0	3	1	9
	NextKSOffsetIsPADDR	3	0	2	1	6
	NoDupInFreeSlotsList	2	1	2	3	8
	freeSlotsListIsFreeSlot	6	1	4	1	12
	DisjointFreeSlotsLists	4	2	2	5	13
	inclFreeSlotsBlockEntries	3	2	1	1	7
	DisjointKSEntries	4	2	2	1	9
	noDupPartitionTree	1	1	1	1	4
	isParent	3	2	2	1	8
	isChild	3	2	2	1	8
	accessibleChildPaddrIsAccessibleIntoParent	4	4	3	1	12
	noDupKSEntriesList	2	1	1	1	5
	noDupMappedBlocksList	2	1	1	1	5
	noDupUsedPaddrList	2	1	1	1	5
	sharedBlockPointsToChild	7	5	5	2	19
	wellFormedBlock	4	0	3	2	9
	MPUFromAccessibleBlocks	3	1	2	1	7
Security	verticalSharing	3	4	2	1	10
	partitionsIsolation	5	5	3	1	14
	kernelDataIsolation	3	4	2	1	10

ACKNOWLEDGMENTS

Funding: The research leading to these results was funded by ANRT Convention Cifre n°2020/0380 and contributed to the TinyPART project funded by the MESRI-BMBF German-French cybersecurity program under grant agreements n°ANR-20-CYAL-0005 and 16KIS1395K. This paper reflects only the authors' views. ANRT, MESRI and BMBF are not responsible for any use that may be made of the information it contains.

REFERENCES

[1] Alloy team members . 2021. Website of: Alloy Analyzer. https://alloytools.org/. [Online; accessed December 7, 2023].

[2] Kent Beck; James Grenning; Robert C. Martin; Mike Beedle; Jim Highsmith; Steve Mellor; Arie van Bennekum; Andrew Hunt; Ken Schwaber; Alistair Cockburn; Ron Jeffries; Jeff Sutherland; Ward Cunningham; Jon Kern; Dave Thomas; Martin Fowler; Brian Marick . 2001. Website of: Manifesto for Agile Software Development. https://agilemanifesto.org/. [Online; accessed December 7, 2023].

[3] Leslie Lamport . 2022. Website of: TLA+ Tools. https://lamport.azurewebsites.net/tla/tools.html. [Online; accessed December 7, 2023].

[4] The Why3 Development Team . 2023. Website of: Why3. https://why3.lri.fr/. [Online; accessed December 7, 2023].

[5] University of Cambridge and Technische Universität München . 2022. Isabelle. https://isabelle.in.tum.de/index.html. [Online; accessed October 10, 2022].

[6] Jim Alves-Foss, Paul W. Oman, Carol Taylor, and W. Scott Harrison. 2006. The MILS architecture for high-assurance embedded systems. *International Journal of Embedded Systems* 2, 3-4 (2006), 239–247. https://doi.org/10.1504/ijes.2006.014859

[7] June Andronick. 2022. Website of: CPP keynote "The seL4 verification: the art and craft of proof and the reality of commercial support". https://popl22.sigplan.org/details/CPP-2022-papers/27/The-seL4-verification-the-art-and-craft-of-proof-and-the-reality-of-commercial-suppo. [Online; accessed June 28, 2023].

[8] June Andronick, Ross Jeffery, Gerwin Klein, Rafal Kolanski, Mark Staples, He Zhang, and Liming Zhu. 2012. Large-scale formal verification in practice: A process perspective. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, Zürich, Switzerland, 1002–1011. https://doi.org/10.1109/ICSE.2012.6227120

[9] Andrew W. Appel. 2011. Verified Software Toolchain. In *Programming Languages and Systems*, Gilles Barthe (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17.

[10] Md. Junaid Arafeen and Saugata Bose. 2009. Improving Software Development Using Scrum Model by Analyzing Up and Down Movements On The Sprint Burn Down Chart - Proposition for Better Alternatives. *J. Digit. Content Technol. its Appl.* 3 (2009), 109–115. https://api.semanticscholar.org/CorpusID:32851875

[11] David Aspinall and Cezary Kaliszyk. 2016. Towards formal proof metrics. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9633 (2016), 325–341. https://doi.org/10.1007/978-3-662-49665-7_19

[12] Andreas Bollin and Abdollah Tabareh. 2012. Predictive Software Measures based on Z Specifications - A Case Study. *Electronic Proceedings in Theoretical Computer Science* 86 (jul 2012), 33–40. https://doi.org/10.4204/eptcs.86.5

[13] Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. 2012. Challenges and Experiences in Managing Large-Scale Proofs. In *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7362)*, Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge (Eds.). Springer, Bremen, Germany, 32–48. https://doi.org/10.1007/978-3-642-31374-5_3

[14] Richard A De Millo, Richard J Lipton, and Alan J Perlis. 1979. Social processes and proofs of theorems and programs. *Commun. ACM* 22, 5 (1979), 271–280.

[15] Nicolas Dejon, Chrystel Gaber, and Gilles Grimaud. 2022. From MMU to MPU: adaptation of the Pip kernel to constrained devices. In *3rd International Conference on Internet of Things & Embedded Systems (IoTE 2022)*. AIRCC, Sydney, Australia, 19 pages. https://hal.science/hal-03705114

[16] Nicolas Dejon, Chrystel Gaber, and Gilles Grimaud. 2023. Pip-MPU: Formal Verification of an MPU-Based Separation kernel for Constrained Devices. *International Journal of Embedded Systems and Applications* 13 (06 2023), 1–21. https://doi.org/10.5121/ijesa.2023.13201

[17] Benjamin Delaware, William Cook, and Don Batory. 2011. Product Lines of Theorems. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 595–608. https://doi.org/10.1145/2048066.2048113

[18] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An extensible architecture for building certified concurrent OS kernels. *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016* 4 (2016), 653–669.

[19] C. A.R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[20] G.J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering* 23, 5 (1997), 279–295. https://doi.org/10.1109/32.588521

[21] Alexei Iliasov, Paulius Stankaitis, and Alexander B. Romanovsky. 2016. Proving Event-B Models with Reusable Generic Lemmas. In *Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, November 14-18, 2016,*

Proceedings (Lecture Notes in Computer Science, Vol. 10009), Kazuhiro Ogata, Mark Lawford, and Shaoying Liu (Eds.). Springer, Tokyo, Japan, 210–225. https://doi.org/10.1007/978-3-319-47846-3_14

[22] INRIA. 1984. Website of : Coq. https://coq.inria.fr. [Online; accessed January 17, 2020].

[23] Ross Jeffery, Mark Staples, June Andronick, Gerwin Klein, and Toby Murray. 2015. An empirical research agenda for understanding formal methods productivity. *Information and Software Technology* 60 (2015), 102–112. https://doi.org/10.1016/j.infsof.2014.11.005

[24] Narjes Jomaa. 2018. *Le co-design d'un noyau de système d'exploitation et de sa preuve formelle d'isolation.* Ph. D. Dissertation. Université de Lille, Bâtiment Esprit, Villeneuve d'Ascq.

[25] Narjes Jomaa, David Nowak, and Paolo Torrini. 2018. Formal Development of the Pip Protokernel. https://entropy2018.sciencesconf.org/data/pip.pdf. [Online; accessed October 10, 2022].

[26] Narjes Jomaa, Paolo Torrini, David Nowak, Gilles Grimaud, and Samuel Hym. 2019. Proof-Oriented Design of a Separation Kernel with Minimal Trusted Computing Base Proof-Oriented Design of a Separation Kernel with Minimal Trusted Computing Base. *AVoCS 2018* 076 (2019), 21 pages.

[27] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems* 32, 1 (2014), 1–70.

[28] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[29] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–370.

[30] Stephane Lescuyer. 2015. ProvenCore : Towards a Verified Isolation Micro-Kernel. *International Workshop on MILS: Architecture and Assurance for Secure Systems* (20 1 2015), 8 pages.

[31] Daniel Matichuk, Toby Murray, June Andronick, Ross Jeffery, Gerwin Klein, and Mark Staples. 2015. Empirical study towards a leading indicator for cost of formal software verification. *Proceedings - International Conference on Software Engineering* 1 (2015), 722–732. https://doi.org/10.1109/ICSE.2015.85

[32] Peter O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95. https://doi.org/10.1145/3211968

[33] Proofcraft. 2023. Website of: Proofcraft. https://proofcraft.systems/. [Online; accessed June 28, 2023].

[34] ProvenRun. 2020. Website of: ProvenRun. https://www.provenrun.com/. [Online; accessed June 28, 2023].

[35] Ratish J. Punnoose, Robert C. Armstrong, Matthew H. Wong, and Mayo Jackson. 2014. Survey of Existing Tools for Formal Verification. *Sandia Report SAND2014-20533, Sandia National Laboratories, Albuquerque* (12 2014), 1–42. https://doi.org/10.2172/1166644

[36] Talia Ringer. 2021. *Proof Repair.* Ph. D. Dissertation. University of Washington.

[37] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. QED at large: A survey of engineering of formally verified software. *Foundations and Trends in Programming Languages* 5, 2-3 (2019), 102–281. https://doi.org/10.1561/2500000045 arXiv:2003.06458

[38] J. M. Rushby. 1981. Design and verification of secure systems. *Proceedings of the 8th ACM Symposium on Operating Systems Principles, SOSP 1981* 15, 5 (1981), 12–21. https://doi.org/10.1145/800216.806586

[39] Helgi Sigurbjarnarson, James Bornholt, Nicolas Christin, and Lorrie Faith Cranor. 2017. Push-Button Verification of File Systems via Crash Refinement. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 1–16. https://www.usenix.org/conference/atc17/technical-sessions/presentation/sigurbjarnarson

[40] Mark Staples, Ross Jeffery, June Andronick, Toby Murray, Gerwin Klein, and Rafal Kolanski. 2014. Productivity for Proof Engineering. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM/IEEE, Torino, Italy, 1–4.

[41] Claudia Tona, Reyes Juárez-Ramírez, Samantha Jiménez, Ángeles Quezada, César Guerra-García, and Rafael González Pacheco López. 2021. Scrumlity: An Agile Framework Based on Quality Assurance. In *2021 9th International Conference in Software Engineering Research and Innovation (CONISOFT)*. 88–96. https://doi.org/10.1109/CONISOFT52520.2021.00023

[42] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for Change in a Formal Verification of the Raft Consensus Protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (St. Petersburg, FL, USA) *(CPP 2016)*. Association for Computing Machinery, New York, NY, USA, 154–165. https://doi.org/10.1145/2854065.2854081

[43] He Zhang, Gerwin Klein, Mark Staples, June Andronick, Liming Zhu, and Rafal Kolanski. 2012. Simulation modeling of a large-scale formal verification process. In *2012 International Conference on Software and System Process (ICSSP)*. IEEE, Zürich, Switzerland, 3–12. https://doi.org/10.1109/ICSSP.2012.6225979

Received 10 July 2023; revised 14 December 2023; revised 19 March 2024; accepted 3 May 2024