Large-Scale Formal Verification in Practice: A Process Perspective

June Andronick, Ross Jeffery, Gerwin Klein, Rafal Kolanski, Mark Staples, He Zhang, Liming Zhu
NICTA, Level 5, 13 Garden St, Eveleigh NSW 2015, Australia
School of Computer Science and Engineering, University of New South Wales, NSW 2052, Australia
<firstname>.<lastname>@nicta.com.au

***Abstract*—The L4.verified project was a rare success in large-scale, formal verification: it provided a formal, machine-checked, code-level proof of the full functional correctness of the seL4 microkernel. In this paper we report on the development process and management issues of this project, highlighting key success factors. We formulate a detailed descriptive model of its middle-out development process, and analyze the evolution and dependencies of code and proof artifacts. We compare our key findings on verification and re-verification with insights from other verification efforts in the literature. Our analysis of the project is based on complete access to project logs, meeting notes, and version control data over its entire history, including its long-term, ongoing maintenance phase. The aim of this work is to aid understanding of how to successfully run large-scale formal software verification projects.**

***Keywords*-program verification; microkernel; L4; software process; formal methods**

I. Introduction

Software verification is the discipline of determining whether the software that has been built is consistent with its specification. The most common approaches are testing and code inspection. However, these are limited in rigor and the extent of possible behaviors that can be checked. Formal methods (FM) is the application of mathematical techniques to specify programs (formal specification) and to prove that programs meet those specifications (formal verification). Formal verification can check not just all lines of code or all decisions in a program, but all possible behaviors for all possible inputs. Most previous industrial use of formal methods for software has only performed formal specification, rarely formal verification [1]. While formal verification is mandated in some standards such as the Common Criteria for the highest assurance level (EAL7), even there only design-level verification is required, rather than full code-level verification [2]. When formal verification is performed, often only lightweight properties are proved, rather than proving correspondence to a complete specification of functionality [3].

The L4.verified project has performed not just formal specification, but also formal verification; not just at design level, but down to C source code; and not just for lightweight properties, but for the full functional correctness of a highly complex software system—the seL4 (secure embedded L4) microkernel. seL4 is part of the L4 family of high-performance operating system (OS) microkernels [4]. The seL4 kernel is designed with the explicit goals of high performance, formal verification, and secure access control. The formal verification of seL4 is the most detailed that has been performed on software of this size and complexity [5].

The project ran over 4 years from 2005 to 2009, including the design and implementation of the kernel as well as ongoing maintenance since then. It involved two teams: OS kernel developers and formal methods practitioners. As a formal verification project, it was different in many ways to typical large industrial software engineering projects. This paper describes the experiences of the L4.verified project in terms of the software engineering process and management issues that were encountered. It complements earlier papers that discussed the role of the project's executable specification [6]; described experiences in the use of a Haskell prototype [7]; and described the correctness proof itself [5].

In this paper we aim to contribute to the knowledge about how large formal methods projects can be run successfully. In particular, the contributions of this paper over previous publications are 1) a detailed, descriptive process model supported by project logs and version control data over the entire project history; 2) a comprehensive analysis of this data; and 3) an in-depth discussion of our findings and lessons learned, together with their relation to the perception of formal methods in the literature. We believe that the process model of this project will be more generally applicable.

II. The seL4 Microkernel and Proof

A. The seL4 Microkernel

A kernel is the part of the OS that runs in the privileged mode of the hardware. It has direct access to all hardware resources and provides the basic mechanisms for implementing the rest of the system. A microkernel, as opposed to more common monolithic OS kernels, is reduced to the bare minimum of functionality and code. This radical reduction in size comes with a price in complexity. It results in high coupling and a high degree of interdependency between different parts of the kernel, as becomes apparent in the function call graph of seL4 in Fig. 1.

The seL4 kernel comprises 8,700 lines of C code and 600 lines of assembler, not counting blank lines and comments. The motivation for the radical reduction in size and for formally proving functional correctness, is that the OS kernel

978-1-4673-1067-3/12/$31.00 © 2012 IEEE

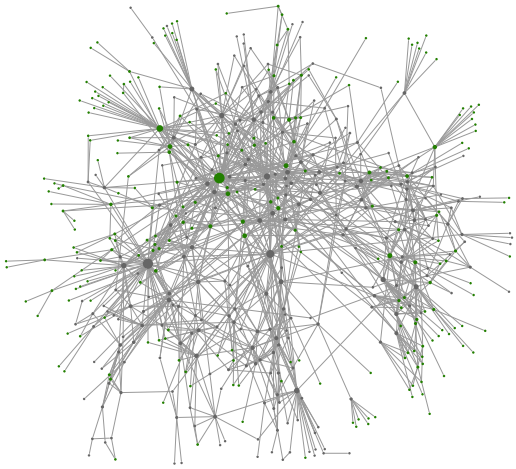


Figure 1. The function call graph of the seL4 kernel.

is one of the most critical components of any system built on top. If one is to provide assurance for safety, security, or correct functionality of a system, one of the first steps must be to provide assurance for the OS kernel. The small size reduces the amount of this critical code. Formal verification gives the highest degree of assurance we can provide [8].

B. Conceptual Process Model and Verification Artifacts

It is a challenge to design a formally verifiable kernel while maintaining high performance. To obtain high performance, kernel developers usually take a *bottom-up* approach to design, focusing on low-level details that allow efficient management of hardware. In contrast, formal methods practitioners often prefer a *top-down* approach based on simple models with a high level of abstraction.

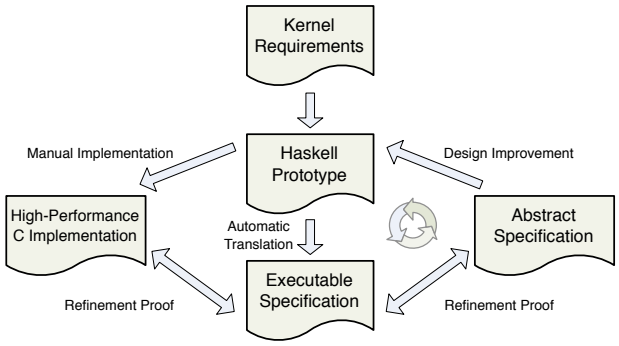


Figure 2. Conceptual process model of middle-out verification

To achieve these two objectives, the L4.verified project bridged the gap between verifiability and performance using an iterative, prototype-based, middle-out process shown in Fig. 2. It is based around an intermediate target that is used and understood by both the kernel developers and the formal methods practitioners, with the aim of rapidly iterating through design, prototype, implementation and formal model until convergence. This intermediate target is a prototype of the kernel written in the functional language Haskell. It is translated automatically into the *executable specification* of the kernel in the theorem prover Isabelle/HOL [9]. The prototype can be used to directly exercise user-level programs that interface with the kernel and thereby validate the API under development. The importance of the use of executable specifications in formal verification in a theorem prover has been recognized previously in the ACL2 community [10], focussing on microprocessor verification in particular.

The *abstract specification* is a formal description of the functionality of the kernel. It specifies the interface and effects of system calls, but not the details of how these effects are implemented. In other words it describes *what* is expected from the kernel, whereas the executable specification describes *how* the kernel will achieve its purpose. In that sense the executable specification is a *design* of the kernel.

The proof that the executable specification refines the abstract specification (on the right in Fig. 2) was carried out first. This proof can be seen as design verification. Then a low-level, *high-performance implementation* of the kernel was manually written in the C programming language. The second proof (on the left in Fig. 2) shows that this C code correctly implements the executable specification, which we will also refer to as code verification. Note that the C code is translated automatically into the theorem prover for verification [11].

III. Modeling the Formal Verification Process

The formal verification of seL4, and the development of the kernel itself, were not part of a conventional software engineering process. Instead they followed the implicit conceptual process described above. In this section, we formulate a detailed model of the process applied in this project, with the aim to identify any reusable process patterns and potential process factors that contributed to the successful scaling of formal verification. We find that the middle-out approach provides advantages over pure top-down and bottom-up processes.

We created the process model using process elicitation interviews with the L4.verified project leaders. The interviews were used by expert process modelers to create initial descriptive process models. These were iteratively corrected and validated by the project leaders, based on their recollections from the project and project meeting notes.

We then performed a more detailed validation of the process model by analyzing and mining data directly from the project repositories: Mercurial source-code version control and an internal wiki, supplemented by meeting notes and email records. The version control repositories provided detailed information about the number and size of changes and artifacts, and the time at which activities took place. The combination of model and data analysis lead to the

identification of phases in the project, and to an appreciation of how activities may overlap in this middle-out approach.

A. Descriptive Process Model

As shown in the conceptual model in Fig. 2, the L4.verified project used a middle-out approach, starting with an executable specification, which was then proved to be consistent with a high-level abstract specification, and finally also proved to be consistent with the low-level source code.

Fig. 3 shows the descriptive process model of L4.verified created from our analysis. Each activity in the graphic model is annotated with the performers (OS or FM team), type (manual or automatic) and its step number (e.g., S1). We use the terms activity and step interchangeably in this paper. To keep the presentation concise, we do not include activities on proof tools and libraries in this version. The formal verification activities are *technical development processes* [12] modelled between the three levels of design abstraction.

The steps S1 to S7 in Fig. 3 correspond roughly to the activities in the conceptual model. The main new activity is performance testing (S7) in the maintenance phase of the project, which itself is also new compared to Fig. 2.

One of the main design goals of the L4.verified project was a high-performance kernel, at least in specific critical areas, to demonstrate that formal verification does not impact performance adversely. Step S7 shows that low-level performance tuning was performed after verification was complete. Performance problems discovered in this step would lead to low-level code changes or in rare cases also to small design-level changes in the Haskell prototype re-starting the process loop in re-verification.

Note that performance testing does not depend on the verification being completed, it just happened to come last in this project because the OS team was confident they could predict performance based on the detailed design. Ideally, it would commence soon after the C code has stabilized earlier in the process. Doing this might save overall time and effort because fewer performance-related changes would have emerged later that required re-verification.

Further differences between the conceptual model and the process description in Fig. 3 are explicit information sources, explicit decision points, and more detail in the project artifacts that are modeled. The information source artifacts from outside the process are denoted by dashed lines. They are the initial kernel requirements on the top left, new feature & change requests, also top left in the diagram, and feature & change requests in maintenance on the bottom right. The decision points in the diagram are mostly explained by their labels. The project artifacts are more detailed than in the conceptual model, and as well mostly explained by their labels in Fig. 3.

An interesting new artifact is the set of invariants of the kernel. Invariants are proved on the level of the executable specification as well as the abstract specification. Both are proved mostly in the design verification step S4, which consumed roughly 60 – 70% of time and effort in the process. The invariants are reused heavily in the code verification step S6. However, step S6 may induce additional invariants to be proved on the executable specification. Invariant proofs are the highest-effort parts of this verification.

The process experienced multiple iterations through steps S1–S6. They were triggered by feature changes in the prototype and by defects discovered during either verification phase. Code-level defects were usually fixed directly in S5 and S6, but in rare case were escalated to the design level. In theory, S4 and S6 could run in parallel. However, significant savings in code-level verification were possible because the invariants from S4 had stabilized. Starting S6 too early may negate this effect.

In the diagram and in the project, the manual implementation of the kernel in C was step S5, i.e. after S4 had gone through its first major iteration. Note that this is not necessary. This step could start earlier in the process, in parallel with other activities, to optimize the overall process performance.

More generally, the descriptive model identifies four phases of the project as shown on the right-hand side of the diagram: 1) the development of the prototype, which clearly appears first, 2) the definition of the specification and validation of the design, as an iterative process on the right of the diagram, 3) the implementation of the kernel together with the code verification, as an iterative process on the left of the diagram, and finally 4) the maintenance phase where change & feature requests are propagated to the top of the process, updating code and models and re-verifying the proofs.

B. Analysis of Project Data

The development and verification of the seL4 kernel from prototyping through implementation, including all formal models and proofs, has been managed using version control. This provides detailed information about the evolution of artifacts over the full lifetime of the project, including its ongoing maintenance phase. We first describe how the data was retrieved from the version control repositories, and then show graphs of artifact size over time extracted from this data. The graphs confirm the identification of phases in the descriptive process model in Fig. 3.

The artifacts of interest resided within three separate project repositories containing thousands of changesets, not all of which concern artifacts discussed in this work. For example, the verification was performed for the ARM11 architecture, while the C repository also contains code for other architectures. To make sense of this data, we needed a higher-level view and developed a tool to iterate over all changesets in the repositories and match file path names against user-supplied path patterns chosen to identify the artifacts related to the formal verification of seL4. If the tool could not classify a file, more patterns were supplied until

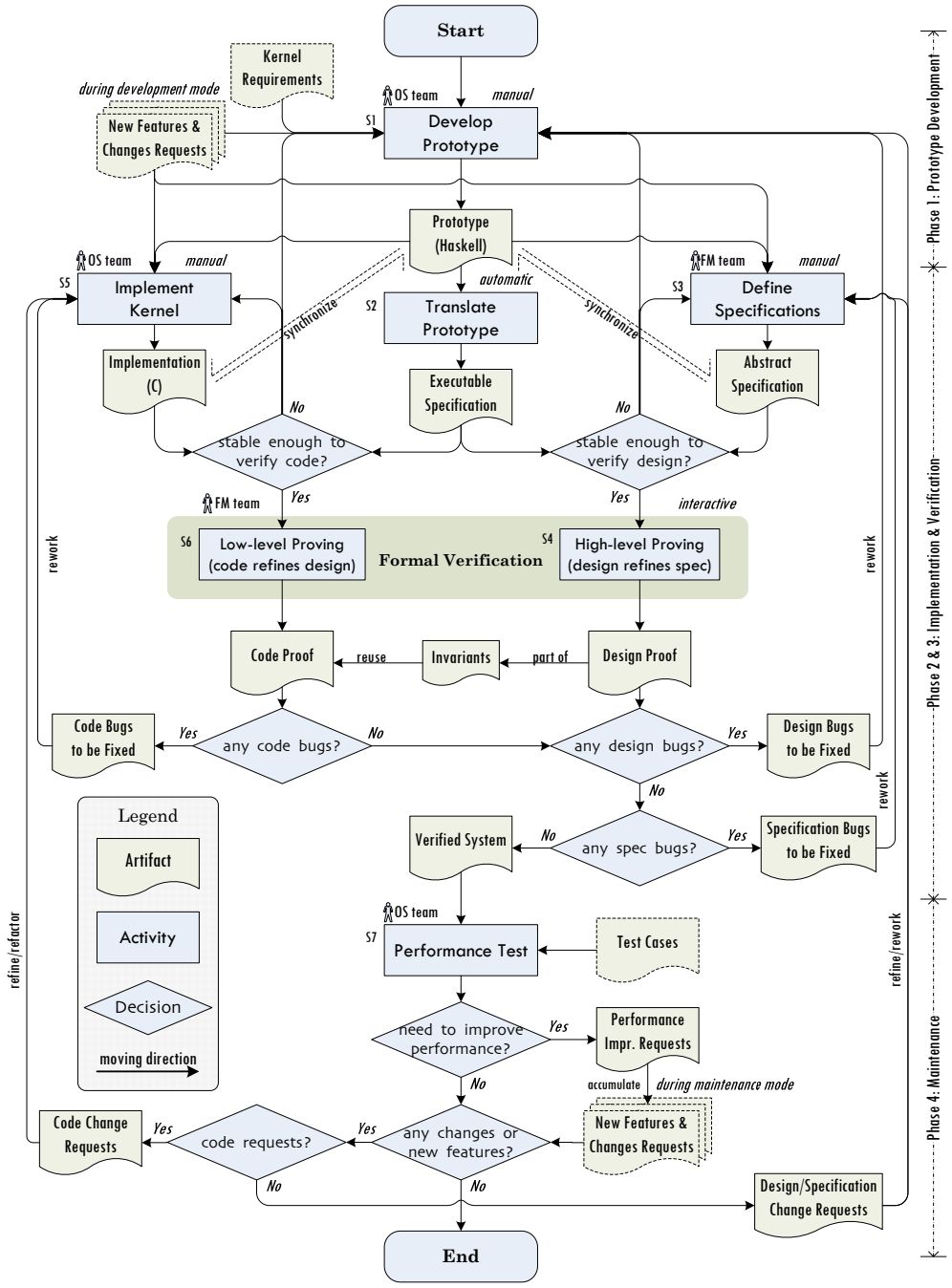
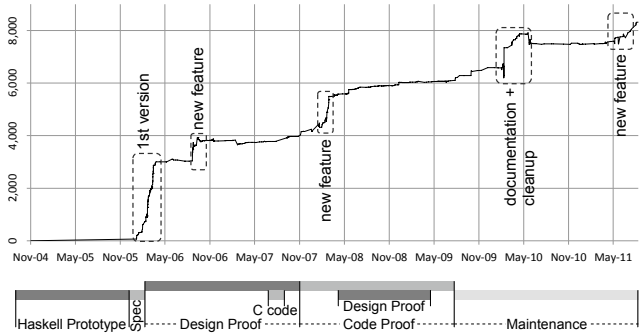


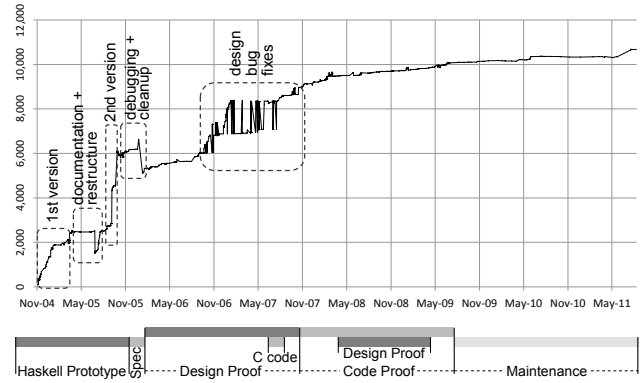
Figure 3. Descriptive process model of L4.verified

all files were classified as either artifact-related or irrelevant. Each changeset identifies who made the change, giving us an estimate of the total number of people working on each artifact in any period. However, this is at best a coarse estimate, because of variation in the frequency and size of commits by project members.

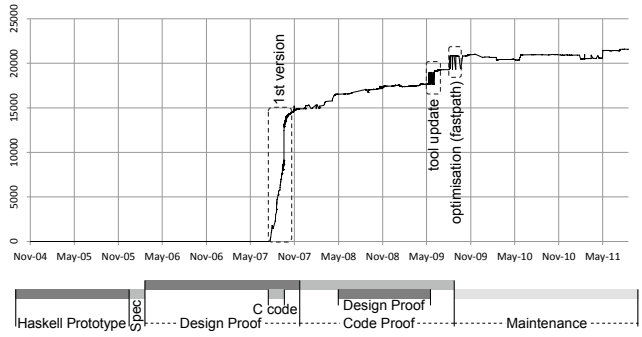
Instead, we chose to analyze the variation in lines of code (LOC) for each artifact on a per-changeset basis. This included the C code, the code for the Haskell prototype, and the Isabelle/HOL proofs. The resulting graphs were reviewed by people with experience working on the artifacts. Our choice of LOC was motivated partly by its ability to act a common denominator across the variety of artifact types and file formats in the project (we discuss the choice of metric further in Sect. IV-C). We counted all lines, including comments and empty lines. This is justified, especially given the way we use LOC to analyze productivity trends, not quality [13]. The raw line counts reported here are different



(a) Abstract Specification

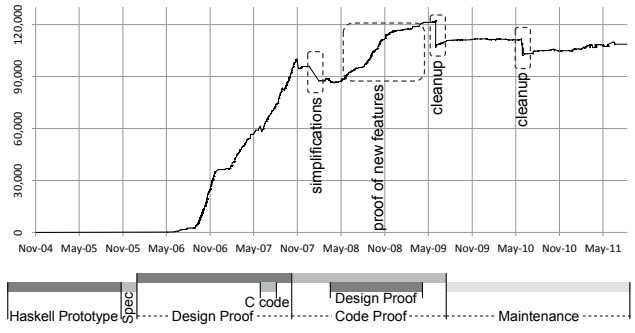


(b) Executable Specification

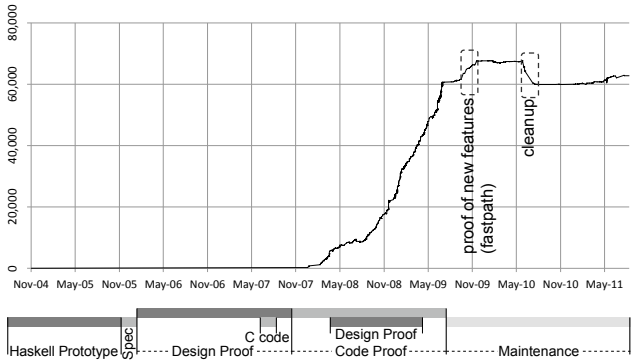


(c) C Implementation

Figure 4. Size of L4.verified code/spec artifacts (X: time; Y: LOC)



(d) Refinement Proof Abstract/Executable



(e) Refinement Proof Executable/C

Figure 5. Size of L4.verified proofs (X: time; Y: LOC)

to the SLOCCount-based numbers [14] for seL4 given in Sect. II-A and previous work, which are normalized and intended for comparison with other projects.

The graphs provide a view of the project comparing the changes made to the multiple kinds of artifacts. Fig. 4 shows the graphs for code and formal specifications, and Fig. 5 for the two refinement proofs. Early rapid development, later feature addition, and maintenance phases are visually evident. Note that some graphs contain rapidly oscillating segments. These result from use of a distributed version control scheme in which development may proceed in parallel on separate branches before merging.

Analysis of the graphs combined with explanatory project logs enabled us to identify the main phases in the project, as shown under each graph. The phases are consistent with the descriptive model in Fig. 3. We now show how the graphs validate the process model and reveal the phases. We have annotated the graphs with explanations of the main events, so we will not describe these individually. Instead we concentrate on the conclusions we can draw from them.

The first activity in the project is the development of the executable specification (Fig. 4 (b)). Following this graph into the verification phases and noting its stability after the design proof shows that design defects were mainly revealed during this design proof, and less during the code proof. This is the desired behavior. It highlights that the middle-out process allows early detection of design problems, and minimizes the impact of code defects on the design.

The graph for the abstract specification (Fig. 4 (a)) shows that activities there only started once the prototype was stable. The main takeaways from this graph are that new features happen at every stage of the kernel lifetime, from its verification up to its maintenance, and that these are the main big changes in the specification size. We will see that these new features are also the main source of re-verification.

The C code graph (Fig. 4 (c)) reveals important points about the middle-out process. Firstly, the C code was implemented extremely rapidly. This is because all design decisions had already been made, and partly validated by the ongoing design verification; the implementation thus mainly addresses optimization. Secondly, the process achieves a

separation of concerns between design, specification and implementation. A change in *how* the kernel is implemented, such as the major optimization activity in the graph (fastpath), had very little impact on the design or the specification. This is again a desired property of the development process.

The proof graphs in Fig. 5 highlight the different phases and how they may overlap. Both show the effort on the first attempt at the proof, and then a succession of cleanup phases and proof updates due to new features. This is clearly consistent with the descriptive process model given in Sect. III-A. It also emphasizes one benefit of the middle-out process: updates in one proof can be done in parallel with the development of the other proof, as seen in the bar chart under the graphs.

IV. Discussion

A number of earlier authors have discussed overall managerial issues for projects using formal methods [15], myths about the application of formal methods [16], [17], and integration of formal methods into the development process [18], [3], [19]. Authors have drawn lessons from industrial quasi-experiments [20], and from reviews of industry experience [21], [1], [22]. In this section we discuss a number of recurring issues from these papers, and compare those prior observations with our experience in the L4.verified project. Many previous large formal methods projects have only attempted formal specification, so our experiences in code-level formal verification sometimes provide a new perspective on these issues.

A. Cost and Effort

Formal methods is often believed to be highly expensive [16]. Although the cost impact of using formal specification varies [15], some prior industrial projects have experienced overall cost reductions when using formal specifications [16], [21], [1]. Nonetheless, complete formal program verification has rarely been attempted, and is still often regarded to be infeasibly expensive [16], [8].

As previously reported [5], systems similar to seL4 could have been expected to take between 4 and 6 person years to develop. Our experience on the L4.verified project is that the overall cost of the kernel design, development and formal verification (excluding costs of more generally-applicable research and development on theorem prover technologies and frameworks) was less than 14 person-years. For the development and formal verification of a comparably-complex new second system, this might be reduced to 8 person years. [5] Our experiences indicate that although formal verification at this very detailed level does still increase overall cost, it is not infeasibly expensive. For critical systems where high levels of assurance are required, the additional cost seems reasonable.

One key aspect of a proof of functional correctness by refinement is that it can reduce the effort for showing further properties about the kernel. If the property is preserved by refinement, it need only be shown about the massively simpler specification, not about the complex C code. For instance, proving the high-level security properties *integrity* and *authority confinement* of seL4 merely took 10 person months. They were proved about the specification, and per refinement they automatically hold on the implementation level of seL4 [23].

In terms of effort, the maintenance period consisted of one to three team members updating proofs on demand next to other ongoing verification projects. The effort spent on maintenance adds up to roughly the equivalent of one full-time person on average since the end of the proof. For some significant new features, such as improving the API for memory allocation, re-verification took on the order of 6 months. This time could have been reduced by adding more verification team members to the task.

B. Scale and Assurance

Both the scale of the verified code and the level of detail and assurance provided by the verification set this project apart from most other formal verification efforts. We posit that the project succeeded in large part due to the process used in the project, together with other formal and technical innovations. We argue that reaching the implementation level with formal verification brings a significant improvement in assurance and practical maintainability over previous approaches such as merely using formal specification or lightweight properties on higher-level models.

In terms of assurance, the code level is an important barrier. If formal, machine-checked verification is performed to the code level, then no trust in human activity is left in the chain from running system to high-level specification. Even though formal verification, as any other kind of verification, is necessarily based on a set of assumptions about the physical world—in this case, the behavior of the machine, the compiler, and linker—these are general assumptions that can be validated for classes of systems. If there is a gap between formal model and code that needs to be bridged by a human manually checking correspondence between implementation and model, this assumption needs to be re-validated every time for every change to code and system. This kind of meticulous, detailed work is highly error-prone if performed manually—even for expert mathematicians [24]. Verifying the implementation directly avoids some prior problems that have arisen in practice, where the programming language did not correspond closely to the one implemented in commercial compilers [25], or where there were errors in automated model-driven code-generation [26].

C. Estimation and Metrics

Previous authors have said that estimation for formal methods projects is hard [17], [15]. It is not known how to calibrate measures of formal methods work to traditional

metrics approaches [22]. Our experience on the L4.verified project certainly agrees with this.

We do not yet have a good understanding of what to measure in formal verification projects. Just as lines of code is a problematic measure of programs, so too lines of proof seems to be a problematic measure of proofs. Furthermore measures similar to structural complexity for source code cannot be used for machine-checked interactive proofs, in which the proof commands given to an interactive theorem prover are often presented linearly, losing the structure of the proof itself. Similar to programming, different formal verification team members have different proof styles, and the different lengths of their proofs is not necessarily an indicator of differing levels of proof productivity. Shorter proofs are sometimes harder to construct, but may be easier to maintain. Different types and amounts of proof automation can also be used, and time spent to improve proof technology, or to develop system-specific proof tools, can sometimes reduce overall verification costs. At this stage, these observations are qualitative lessons only—we do not yet have appropriate measures to express these cost tradeoffs quantitatively, nor do we have explicit decision-making models to inform project management judgments about these tradeoffs.

The cost of the formal verification effort is related not just to the number of major sub-systems or functions that must be proved correct, but also to the complexity of the overall system. The complexity of the system is manifest not just during development through coupling between sub-systems, but also in formal verification through the inter-related invariant conditions of these sub-systems. As a microkernel, seL4 may be more complex than might be expected for many other embedded systems.

However, the FM team managed to design refinement calculi for each of the major proof steps that allowed the work to be divided up among members of the team [27], [28]. Especially in the code-level proof, individual lemmas proving the correctness of separate functions in the kernel implementation could be proved independently. In the design-level proof, the same could be achieved once the formulation of invariants had stabilized.

D. Re-verification

In any complex new software development project, the specification, design, and implementation of the system will change during development, to add features, improve non-functional performance, or remove bugs. Instances of all of these types of change have happened in the L4.verified project. A well-recognized challenge for formal verification in dealing with such changes is the need to subsequently re-verify the system [25], [29], [16], [1]. Failing to re-establish correctness will result in the loss of assurance about the system [26]. This sort of problem has arisen in practice, either because a variant of the verified specification was used by developers [29], or because the final code used in the system had received some incorrect manual "tidying-up" [26]. These are not inherent problems with formal verification, but arise from inadequacies in the broader software development process and environment within which formal verification was used. There has been some concern that these inadequacies may not be easily avoidable—prior authors have been skeptical that ongoing re-verification is feasible [25], especially for formal verification using interactive theorem provers [3].

Our experience in the L4.verified project is that re-verification of full functional correctness is feasible using modern interactive theorem provers. As discussed in an earlier paper [5], the cost of re-verification can vary significantly. The cost depends on the nature of the change—changes to the top-level specification, changes to the system design (including major interface specifications), or to system invariant conditions can all entail significant proof re-work. In the worst case, one change to less than 5% of the code base resulted in proof rework equivalent to 17% of the entire original proof effort [5]. However, in the common case, for local changes to the implementation of a specific sub-system, the effort to re-verify the system is low, proportional to the effort for the change to the code, often a factor of roughly 3–5. For instance, a small performance optimization may take 2h to implement and measure and about an additional 6h to update and re-verify the relevant proofs.

Note that after code-level verification finished, there were zero code-level defects, and so no maintenance changes occurred for this otherwise common reason. There may well be specification fixes as described above and there may also be code fixes in assumed, unverified parts of the system, for instance in the 600 lines of seL4 assembly code, but the proof guarantees the absence of disagreements between code and specification. Code fixes in assumed parts require traditional forms of re-verification, e.g. a regression test suite, but do not impact the formal proof unless they change assumptions.

Although code-level formal verification has some additional cost, there are also some significant advantages for system assurance in the face of ongoing system change. In the maintenance period, the team has integrated formal verification into an automated proof checking suite, similar to an automated test suite, but using machine-checked formal proofs instead of executable tests. This provides an automatic check of the state of the code verification, and identifies which specific portions of the proof must be re-established. If the check fails, a visible green light in the office changes to red, and the problem can be quickly addressed. This situation contrasts with previous industrial use of formal specification without verification supported by mechanized proof—in such projects, the ongoing assessment of the consistency of the system must be manually determined, including an analysis of the full extent of the impact of the change.

Another advantage of verification during the maintenance phase was that the detailed knowledge about code behavior, mostly encoded in invariants, could be further exploited. A

proposed performance optimization, for instance, initially concerning small changes in multiple files could be reduced to a single line change, because of already verified invariants. In performing worst-case execution-time analysis using static analysis, the analysis team could rely on invariants to exclude infeasible code paths [30].

E. Restrictions on Design Style

One issue of concern about the use of formal methods is whether it imposes any design constraints on the software being developed. A simple design is usually easier to formally verify than a more complex one. The possibility is that this may lead designers to make a tradeoff in favor of making formal verification easier by create a simplistic system that sacrifices design elements that may have normally been introduced, e.g. to improve non-functional performance. [25] Against this view, earlier literature has said that formal methods does influence design style, but only in "good" ways [18], [3], [25], e.g. by encouraging an increasing effort in the design phase, by better decomposing the system, and by better (and more abstractly) defining internal interfaces.

Our L4.verified experience agrees with the earlier literature. There need be no significant deleterious constraint on design brought about by the use of formal verification. However, formal code-level verification does induce an additional, significant cost factor into design considerations, and some design choices become disproportionately more expensive than others. This constrains design, but need not eliminate every design option that allows non-functional requirements to be achieved.

A key goal for seL4 was to be within 10% of the performance of an earlier (not formally verified) high performance L4 microkernel. This was achieved, and in some cases the performance of seL4 exceeds the earlier kernel [5].

F. Expertise and Collaboration

A common perception about formal methods is that it requires highly trained mathematicians [16], or practitioners with hard-to-acquire expertise [31], [25]. However, much of the reported industrial use of formal methods indicates that, although some training or assistance by consultants may be required [15], [20], [18], formal methods capability is readily acquired by technical experts [19], [20]. In particular, it may be easier to train domain experts in formal methods than formal methods practitioners in the domain [19].

Our experience in the L4.verified project agrees with this. The project began as a joint activity between two teams: OS microkernel developers, and formal methods practitioners. While initially distinct, the capability of the two teams blurred over the course of the project. This enabled a move from a *parallel approach* to more of an *integrated approach* [18] during the project. Unlike [18], we conceptualized this as a change in resource allocation options, rather than as a change to our process model.

The OS team consisted on average of 1 senior expert, 1 PhD student, and 1-2 engineers. The FM team consisted on average of 3-4 senior experts, 1-3 PhD students, and 2-6 engineers who were mostly recent university graduates. The total time commitment was roughly 7 full-time equivalent (FTE) persons on average. Our detailed FTE data per artifact (not presented in this paper) shows that never more than 5 FTE were spent on the artifacts described here, the rest was spent on proof tools and other infrastructure. This tool effort would be greatly reduced for subsequent verifications.

During later phases of the project, all members of the OS team volunteered to learn how to use the theorem prover. Almost all of the OS team then directly contributed to the proof in some form. The time to become productive for new verification team members, be it from outside or the OS team, averaged 2-3 months, with some people becoming productive after as little as two weeks. While Isabelle comes with ample, high-quality documentation, a key factor in coming up to speed with the tool and the proof was the availability of an existing team of FM experts willing to offer direct advice and training. The other direction of verification team members contributing to OS design and implementation was less frequent, but did occur as well, in particular with long-standing team members. In the maintenance phase, it was frequently one of these team members that had deep insight into how design and implementation changes would affect the kernel and proof.

G. Tool Support

Some industrial projects that have performed formal specification but not formal verification have either not used supporting tools, or deemed tools not to be critical. [22] However, code-level formal verification demands extreme precision and accuracy for a massive amount of proof detail. The formal verification of large software systems must be supported by appropriate tools [1]. The situation is similar to the use of high-level programming languages. It is theoretically possible (and was historically practiced) to translate source code to machine code manually. But it would be impractical to do this for large programs on modern machines—compilers are required. Similarly, it would be impractical to undertake formal verification of a large software system without mechanized proof support.

The L4.verified project used the Isabelle/HOL theorem prover [9], an interactive LCF-style [32] theorem prover that offers a high degree of proof automation. A significant degree of tool customization was performed within the L4.verified project—within the bounds of safety provided by Isabelle's LCF architecture [32]. As well as theory definition at the various levels of abstraction, some system-specific proof search tools were created. There is a cost to developing such supporting infrastructure, but it can reduce the overall formal verification effort.

As discussed above, machine-checked proofs were vital not just to ensure that mistakes were not made in the initial proofs, but also to quickly and automatically ensure no mistakes were made during subsequent changes to the specification, implementation, or proofs.

Aside from theorem provers, other technologies exist for formal verification, such as model checking. However, for the depth and scale of formal verification performed in the L4.verified project, model-checking is unlikely to be sufficient by itself. Model-checkers can verify some complex properties of simple systems, or some simple properties of complex systems. The proof of correctness for seL4 was a complex property of a complex system, for which the flexibility and power of interactive theorem proving is well-suited.

V. Future Directions

The key needs for future research arising from the work reported here are for metrics, estimation models, and process-management tools that support formal verification projects.

As discussed, lines of proof is not an entirely adequate measure for mechanized interactive proofs to assess productivity. 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, to address what-if questions such as evaluating feature change options, and to manage productivity tradeoffs, such as the balance between testing and formal verification, and between proof development and productivity-improving system-specific proof tool development.

We expect these estimation and decision-making models to be based on, or partially validated by, calibrated process simulation models based on the descriptive middle-out process model reported in this paper. The development and validation of these simulation models will be a significant area of future research. Another interesting avenue would be to repeat the analysis on other code-level verification projects such as the type safety proof for Verve [33], or the pervasive verification approach in Verisoft [34].

Finally, the major thrust of future technical work will be to build very large verified systems on top of seL4, in the order of millions of lines of code. The use of code-level formal verification for a highly critical OS microkernel has been considered entirely appropriate, but doubts remain about whether formal verification should extend further [8]. The decision to use formal verification is a tradeoff between higher cost and higher assurance. We believe that it is appropriate and feasible to use formal verification for very large highly-critical systems. Our working hypotheses are that the costs of engineering a software component are only a constant factor higher when using formal verification, and that it is possible to provide system-level guarantees not by formally verifying everything, but instead by formally verifying key components in a high-assurance system architecture that relies on the formally-verified isolation guarantees provided by seL4. No doubt this will place new demands on seL4, with ongoing system maintenance and proof maintenance.

VI. Conclusion

This paper has described experiences and lessons from a process perspective drawn from the L4.verified project, a successful large-scale software systems formal verification.

The major contribution of this paper is a descriptive middle-out process model that was recovered from the experience and validated with reference to data extracted from source code repositories, project logs, and minutes. The phases in the L4.verified project reflected shifting resource allocation over time. However, there is not necessarily a sharp distinction between work at different levels—the process model allows activities to occur in parallel either side of the middle artifact, here the executable specification. In the L4.verified project, although work in later phases shifted from design-level verification to code-level verification, there was nonetheless ongoing change in the abstract and executable specifications.

In addition to process model and data analysis, we have further discussed a number of lessons learned.

The first is that formal verification of complex software systems on the order of 10,000 LOC is feasible. This includes maintenance: when changes inevitably occur to specifications, implementations, or proofs, re-verification is feasible. The cost of re-verification varies, and seems largely to depend on the impact of the change on system-wide invariants.

The second is that machine-checked formal verification in a modern interactive theorem prover is vital to handle the extreme detail and accuracy required for a large formal verification effort tackling full functional correctness. Code-level machine-checked verification has significant benefits for managing proof maintenance and re-verification, because it can provide an automated assessment of whether proofs, specification, and implementation are still consistent. When previous formal verification projects performed design verification, there remained a manual assessment of the correctness of the implementation. The code-level formal verification in L4.verified in almost all cases fully eliminates the need for such manual impact analysis.

Acknowledgment

Authors are listed in alphabetical order. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Computing Surveys*, vol. 41, pp. 19:1–19:36, Oct. 2009.

[2] "Common Criteria for Information Technology Security Evaluation, Version 3.1, Revision 3," Jul. 2009, http://www.commoncriteriaportal.org/cc/.

[3] K. L. McMillan, "Fitting formal methods into the design cycle," in *31st Design Automation Conference*, ser. DAC '94. New York, NY, USA: ACM, 1994, pp. 314–319.

[4] J. Liedtke, "Toward real microkernels," *Communications of the ACM*, vol. 39, pp. 70–77, Sep. 1996.

[5] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *22nd SOSP*. Big Sky, MT, USA: ACM, Oct. 2009, pp. 207–220.

[6] P. Derrin, K. Elphinstone, G. Klein, D. Cock, and M. M. T. Chakravarty, "Running the manual: An approach to high-assurance microkernel development," in *Proc. ACM SIGPLAN Haskell Workshop*, Portland, OR, USA, Sep. 2006.

[7] G. Klein, P. Derrin, and K. Elphinstone, "Experience report: seL4 — formally verifying a high-performance microkernel," in *14th ICFP*. Edinburgh, UK: ACM, Aug. 2009, pp. 91–96.

[8] F. P. Brooks, *The design of design: Essays from a computer scientist*. Addison-Wesley, 2010.

[9] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[10] J. Strother Moore, "Symbolic simulation: An ACL2 approach," in *Formal Methods in Computer-Aided Design*, ser. LNCS. Springer, 1998, vol. 1522, pp. 530–530.

[11] H. Tuch, G. Klein, and M. Norrish, "Types, bytes, and separation logic," in *34th POPL*. Nice, France: Springer, Jan. 2007, pp. 97–108.

[12] L. Zhu, R. Jeffery, M. Staples, M. Huo, and T. T. Tran, "Effects of architecture and technical development process on micro-process," in *Proc. 2007 Int. Conf. on Software Process*. Springer, 2007, pp. 49–60.

[13] J. Rosenberg, "Some misconceptions about lines of code," in *Proceedings of the 4th International Symposium on Software Metrics*. IEEE, 1997, pp. 137–142.

[14] D. A. Wheeler, "SLOCCount," http://www.dwheeler.com/sloccount/, 2001.

[15] D. Stidolph and J. Whitehead, "Managerial issues for the consideration and use of formal methods," in *FME 2003*, ser. LNCS. Springer, 2003, vol. 2805, pp. 170–186.

[16] A. Hall, "Seven myths of formal methods," *IEEE Software*, vol. 7, pp. 11–19, Sep. 1990.

[17] J. P. Bowen and M. G. Hinchey, "Seven more myths of formal methods," *IEEE Software*, vol. 12, pp. 34–41, Jul. 1995.

[18] R. A. Kemmerer, "Integrating formal methods into the development process," *IEEE Software*, vol. 7, pp. 37–50, Sep. 1990.

[19] J. S. Fitzgerald, P. G. Larsen, and P. G. Larsen, "Formal specification techniques in the commercial development process," in *Position Papers from the Workshop on Formal Methods Application in Software Engineering Practice, ICSE-17*, 1995.

[20] P. G. Larsen, J. Fitzgerald, and T. Brookes, "Applying formal specification in industry," *IEEE Software*, vol. 13, pp. 48–56, May 1996.

[21] E. M. Clarke and J. M. Wing, "Formal methods: state of the art and future directions," *ACM Computing Surveys*, vol. 28, pp. 626–643, Dec. 1996.

[22] S. Gerhart, D. Craigen, and T. Ralston, "Observations on industrial practice using formal methods," in *15th ICSE*. Los Alamitos, CA, USA: IEEE, 1993, pp. 24–33.

[23] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein, "seL4 enforces integrity," in *2nd ITP*, ser. LNCS, vol. 6898. Springer, Aug. 2011, pp. 325–340.

[24] J. Harrison, "Formal proof—theory and practice," in *Notices of the AMS*, vol. 55, no. 11, Dec. 2008, pp. 1395–1406.

[25] *Peer Review of a Formal Verification/Design Proof Methodology*. NASA Conference Publication 2377, Jul. 1983.

[26] J. Gibbons, "Formal methods: Why should I care? - the development of the T800 transputer floating-point unit," in *Proceedings 13th New Zealand Computer Society Conference*, 1993, pp. 207–217.

[27] D. Cock, G. Klein, and T. Sewell, "Secure microkernels, state monads and scalable refinement," in *21st TPHOLs*, ser. LNCS, vol. 5170. Springer, Aug. 2008, pp. 167–182.

[28] S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish, "Mind the gap: A verification framework for low-level C," in *22nd TPHOLs*, ser. LNCS, vol. 5674. Munich, Germany: Springer, Aug. 2009, pp. 500–515.

[29] A. Cohn, "A proof of correctness of the Viper microprocessor: the first level," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-104, Jan. 1987.

[30] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing analysis of a protected operating system kernel," in *32nd RTSS*. Vienna, Austria: IEEE, Nov. 2011.

[31] M. Whalen, D. Cofer, S. Miller, B. H. Krogh, and W. Storm, "Integration of formal analysis into a model-based software development process," in *Proc. 12th Int. Conf. on Formal Methods for industrial critical systems*, ser. LNCS, vol. 4916. Springer, 2008, pp. 68–84.

[32] R. Milner, "Logic for computable functions: description of a machine implementation," Stanford University, Stanford, CA, USA, Tech. Rep., 1972.

[33] J. Yang and C. Hawblitzel, "Safe to the last instruction: automated verification of a type-safe operating system," in *Proc. PLDI'10*. Toronto, Canada: ACM, 2010, pp. 99–110.

[34] E. Alkassar, M. Hillebrand, D. Leinenbach, N. Schirmer, A. Starostin, and A. Tsyban, "Balancing the load — leveraging a semantics stack for systems verification," *JAR: Special Issue Operat. Syst. Verification"*, vol. 42, Numbers 2–4, pp. 389–454, 2009.