Challenges and Experiences in Managing Large-Scale Proofs

Timothy Bourke[1], Matthias Daum[1,2], Gerwin Klein[1,2], and Rafal Kolanski[1,2]

[1] NICTA, Sydney, Australia[⋆]
[2] The University of NSW, Sydney, Australia

{timothy.bourke,matthias.daum,gerwin.klein,rafal.kolanski}@nicta.com.au

Abstract. Large-scale verification projects pose particular challenges. Issues include proof exploration, efficiency of the edit-check cycle, and proof refactoring for documentation and maintainability. We draw on insights from two large-scale verification projects, L4.verified and Verisoft, that both used the Isabelle/HOL prover. We identify the main challenges in large-scale proofs, propose possible solutions, and discuss the *Levity* tool, which we developed to automatically move lemmas to appropriate theories, as an example of the kind of tool required by such proofs.

Keywords: Large-scale proofs, Isabelle/HOL, Interactive theorem proving

1 Introduction

Scale changes everything. Even simple code becomes hard to manage if there is enough of it. The same holds for mathematical proof—the four-colour theorem famously had a proof too large for human referees to check [3]. The theorem was later formalised and proved in the interactive proof assistant Coq [4] by Gonthier [7,8], removing any doubt about its truth. It took around 60,000 lines of Coq script. While an impressive result, this was not yet a large-scale proof in our sense. Verifications another order of magnitude larger pose new challenges.

To gain a sense of scale for proof developments, we analysed the Archive of Formal Proofs (AFP) [12], a place for authors to submit proof developments in the Isabelle/HOL proof assistant [14]. The vast majority of the over 100 archive entries are proofs that accompany a separate publication. Submissions contain from 3 to 3,938 lemmas per entry, from 145 to 80,917 lines of proof, and from 1 to 151 theory files. The average AFP entry has 340 lemmas shown in 6,000 lines of proof in 10 theory files. Fig. 1 shows the size distribution of entries ordered by submission date. The spikes between the majority of small entries are primarily PhD theses. Beyond the AFP, an average PhD thesis in Isabelle/HOL is about 30,000 lines of proof in our experience.

In recent years, the first proofs on a consistently larger scale have appeared, in particular the verification of an optimising compiler in the CompCert project [13],

[⋆] 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.

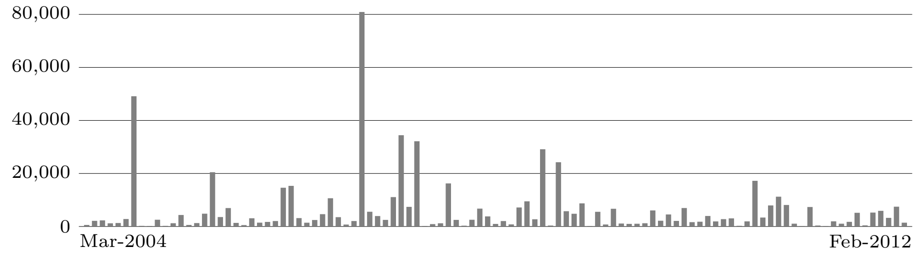

Fig. 1. Size distribution of AFP entries in lines of proof, sorted by submission date.

the pervasive system-level verification of Verisoft [2], and the operating system microkernel verification in the L4.verified project [11]. CompCert 1.9.1 measures 101,608 lines with 3,723 lemmas in 143 theory files, the L4.verified repository currently contains around 390,000 lines with 22,000 lemmas in 500 theories, and the Verisoft project has published about 500,000 lines with 8,800 lemmas in over 700 theories in its three largest releases (duplicates removed). The L4.verified proofs take 8 hours to check, the Verisoft proofs an estimated 12 hours.

When scale increases to this order of magnitude, there is a phase change required in the level of support from the underlying proof assistant. Questions of knowledge management, team interaction, scalability of the user interface, performance, and maintenance become more important. In some cases, they can become more important than the presence of advanced reasoning features.

Large-scale developments concern multiple people over multiple years, with team members joining and leaving the project. The key difference between a large-scale proof and a small one, or even a multi-year PhD, is that, in the former, no single person can understand all details of the proof at any one time.

In this paper, we examine the consequences of this difference against the background of our combined experience from the L4.verified [11] and Verisoft [1,2] projects. We determine which main challenges result from the large-scale nature of these projects for the theorem proving tool, for the proof itself, and for its management. They range from proof exploration for new project members, over proof productivity during development, to proof refactoring for maintenance, and the distribution of a single proof over a whole team in managing the project. We comment on our experience addressing these challenges and suggest some solutions. We describe one of these solution in more technical detail: the proof refactoring tool Levity, developed in L4.verified.

Many of the issues we list also occur for projects of much smaller scale and their solution would benefit most proof developments. The difference in large-scale proofs is that they become prohibitively expensive.

While both projects used Isabelle/HOL, we posit that the challenges we identify are more general. Some of them even are already sufficiently met by Isabelle/HOL. We include them here because the primary aim of this paper is to give theorem prover developers a resource for supporting large proofs that is well founded in practitioner's experience.

2 Challenges

We divide the description of challenges into four perspectives: a new proof engineer joining the project, an expert proof engineer interacting with colleagues during main development, the maintenance phase of a project, and the social/management aspects of a development.

2.1 New proof engineer joins the project

Over the course of a large project, team members come and go, and must be brought up to speed as quickly as possible. Learning how to use Isabelle was not an issue, because extensive documentation and local experts in the system were available. Understanding the subject of the verification, e.g. an OS kernel, was harder, but many solutions from traditional software development applied.

However, quickly inducting new proof engineers, even expert ones, into a large-scale verification remains challenging. Consider the theory graph of L4.verified in Fig. 2, which shows roughly 500 theories. As mentioned, these contain 22,000 human-stated lemmas. Adding those generated by tools takes the count to 95,000. In order to perform her first useful proof, a new proof engineer must find a way through this jungle, identify the theorems and definitions needed, and understand where they fit in the bigger picture.

At the start of L4.verified, we developed a *find_theorems* tool which allows, amongst other features, pattern-matching against theorems and their names, filtering rules against the current goal, and ranking by most accurate match. It also includes an *auto-solve* function to warn users if they restate an existing lemma; a common occurrence not only for newcomers.

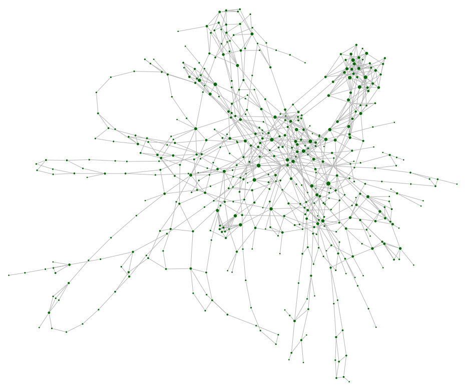
Fig. 2. Theory dependencies in L4.verified

Both tools were incorporated into Isabelle/HOL, but our experience echoed that of Verisoft who found that *Isabelle's built in lemma search assumes that theories are already loaded. This is rarely the case in large developments like ours [...]* [1]. Theorems not already loaded are not visible to the prover. We therefore extended *find_theorems* with a web interface and combined it with our nightly regression test. Any team member can search recent global project builds from their browser. Additionally, we tag Isabelle heaps[3] with revision control information and show this along with the results.

[3] A heap is a file containing a memory dump of prover data structures. It can be loaded to avoid reprocessing theory files from scratch. Isabelle's architecture precludes the separate compilation of theory files possible in some other provers.

In addition to theorems, provers allow custom syntax definitions, which raises questions such as *what does `[x,y.e.z]` mean?* To a newcomer it may not even be clear if the syntax is for a type, an abbreviation, or a constant. Answering such questions requires locating where a symbol is defined. Inspired by Coq, we developed a *locate* command which identifies symbol definitions and syntax. However, the implementation requires explicit knowledge of all definition packages in Isabelle, which are user-extensible. The recent, more deeply integrated approach of the Isabelle prover IDE (PIDE) [20] is promising. In general, better prover IDEs would be beneficial, but designing them for scale is tricky. A short delay in a small test case could translate to a 30 minute wait at scale.

Ultimately, technology can solve only part of this problem. Good high-level documentation of key project areas and ideas like an explicit mentor for each new proof engineer are as important as good prover support.

2.2 Expert proof engineer during main development

For a long-term project member in the middle of proof development, the crucial issue is productivity. In this section, we highlight related challenges.

Automation. A simple way to improve productivity is automation. Domain-specific automation is important in any large-scale verification and the underlying prover should provide a safe extension mechanism. In Isabelle, the LCF approach [9] enabled both projects to provide their own specialised tactics and automated proof search without the risk of unsoundness. An example from Verisoft [2] is a sound, automated verification-condition generator translating Hoare triples about code in a deep embedded language into proof obligations in higher-order logic [17]. L4.verified developed between 10 and 20 smaller automated tactics [6, 22], including a tool to automatically state and prove simple Hoare triples over whole sets of functions.

Non-local change. The usual mode of interactive proof is iterative trial and error: edit the proof, let the prover process it, inspect the resulting proof goals, and either continue or go back and edit further. Verification productivity hinges on this edit-check cycle being short. For local changes, this is usually the case. However, proof development is not always local: an earlier definition may need improvement, or a proof engineer would like to build on a colleague's result. Re-checking everything between a change and the previous point of focus can take several hours if the distance between the points is large enough.

In addition, changes are often speculative; the proof engineer should be able to tell quickly whether a change helps the problem at hand, and how much of the existing proof it breaks. For the former, Isabelle provides support for interactively skipping proof content. For the latter, L4.verified implemented a proof cache that determines if a lemma remains provable, albeit not if the current proof script will still work. The cache works by keeping track of theorem and definition statements and dependencies. A theorem remains provable if none of its dependencies changed, even though global context may have. The cache omits the proof for such theorems using Isabelle's existing skipping mechanism, but replays all others. However, even just scanning definitions and lemma statements without proof can still take tens of minutes. Both questions could be answered concurrently: while the proof engineer turns attention to the previous focus without delay, skipping over intermediate proofs, the prover could automatically replay these proofs and mark broken ones in the background.

Placing lemmas. As argued, proof development is non-linear. Frequently, general lemmas about higher-level concepts are required in a specialised main proof. Once proven, the question arises where to put such a more general lemma in a collection of hundreds of theory files. The answer is often not obvious, so Verisoft developed the *Gravity* tool [2] that gives advice based on theorem dependencies. But even if proof engineers have a reasonable idea where to put a lemma, long edit-check cycles will inhibit them from placing it at the correct position, leading to a significant risk of duplication and making it hard to build on results from other team members. L4.verified addressed the question of lemma movement with the *Levity* tool that we describe in Sec. 3.

Avoiding duplication. It is already hard to avoid duplication in a small project but it becomes a considerable challenge in large-scale proofs. Clearly, proving the same fact multiple times is wasteful. Despite theorem search and related features, duplication still occurred. Even harder than spotting exact duplication is avoiding similar formulations of the same concept. These will often only be recognised in later proof reviews. Once recognised, eliminating duplication can still cause large overhead. In these cases, it would be beneficial if lemmas or definitions could be marked with a deprecation tag or a comment that becomes visible at the use point, not the definition point. A further aspect is generalisation. Often lemmas are proved with constants where variables could instead have been used, which again leads to duplication. Trivial generalisation could be automated.

Proof and specification patterns. In software development, design patterns have been recognised as a good way to make standard solutions to common problems available to teams in a consistent way. An observation in both projects was that proof engineers tended to reinvent solutions to common problems such as monadic specifications in slightly different ways. Worse, re-invention happened even within the project. It is too simplistic to attribute this to a lack of documentation and communication alone. In software development, sets of standard solutions are expected to be available in the literature and are expected to be used. In proof and specification development, this is not yet the case. A stronger community-wide emphasis on well-documented proof and specification patterns could alleviate this problem.

Name spaces. Names of lemmas, definitions, and types, syntax of constants, and other notational aspects are a precious resource that should be managed carefully in large projects. Many programming languages provide name space facilities, provers like Isabelle provide some mechanisms such as locales, but have mostly not been designed with large-scale developments in mind. The challenge is of the same quality but harder for provers than for programming languages: there exists a larger number of contexts, syntax and notation is usually much more flexible in provers (as it should be), and it is less clear which default policies are good. For instance, names of definitions and their syntax should

be tightly controlled to define interfaces. On the other hand facts and lemmas should be easily searchable and accessible by default. At other times, the interface really is a key set of lemmas whereas definitions and notation matters less. Any such mechanism must come with high flexibility and low overhead. For instance, Mizar's [15] requirement of explicit import lists would make large-scale program verification practically infeasible. On the other hand Isabelle's global name space pollutes very quickly and requires the use of conventions in larger developments.
Proof style. Isabelle supports two styles of reasoning that can be mixed freely: the imperative style, which consists of sequential proof scripts, and the more declarative style of the Isar language [19], which aims at human-readable proofs.

The greater potential for readability ostensibly is an advantage of Isar. However, achieving actual readability comes with the additional cost of restructuring and polishing. Neither of the projects were prepared to invest that effort. Instead, both styles were used in both projects, and we did not observe a clear benefit of either style. During maintenance, Isar proofs tend to be more modular and robust with respect to changes in automation. However, by that same declarative nature they also contain frequent explicit property statements which lead to more updates when definitions change.

The usually beneficial modular nature of Isar had a surprising side-effect: proof engineers inlined specialised facts within large sub-proofs. Though that is often a good idea, it also often turned out that such facts were more generally useful than initially thought, but by inlining, they were hidden from theorem search tools. Since the main mechanism for intermediate facts in the imperative style is a new global lemma statement, this occurred less frequently there.

Our recommendation is to mix pragmatically, and use the style most appropriate for the experience of the proof engineer and sub-proof at hand.

2.3 Proof maintenance

As in software development, proof maintenance does not often get up-front attention. Fig. 3 gives an outline of proof activity over time on one L4.verified module: a main development period, multiple cleanups and bursts of activity, and a long maintenance phase, characterised by low levels of activity, starting well before the project's end. Over time, maintenance becomes the dominant activity.
Refactoring. A large part of maintenance involves refactoring existing proofs: renaming constants, types, and lemmas; reformulating definitions or properties for more consistency; moving lemmas; disentangling dependencies; removing duplication. Such refactorings are in part necessitated by failures to avoid duplication during development. This is inevitable in large multi-year projects.

We have already mentioned proof refactoring tools above. Such tools are becoming popular in programming environments, but even there they are often imperfectly implemented. Current theorem proving systems offer no native support for even simple refactorings such as renaming or lemma movement. While in programming languages strict semantics preservation is paramount, it is less of an issue for proof assistants: the theorem prover will complain if the new proof breaks. In practice, time overhead is more important. Refactorings are typically

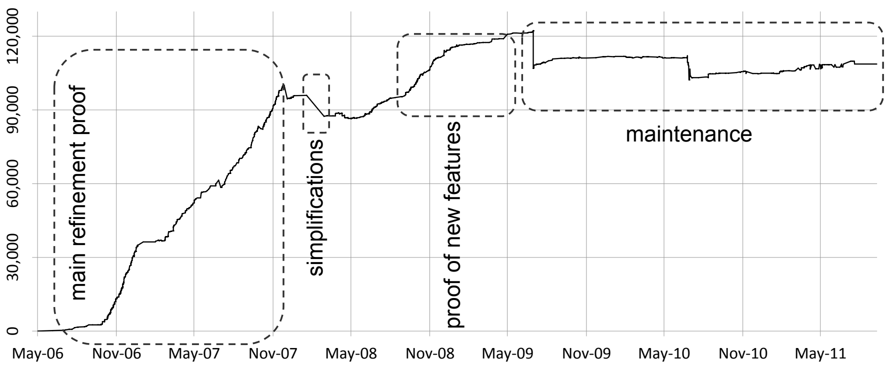

Fig. 3. Lines of proof over time in one L4.verified module.

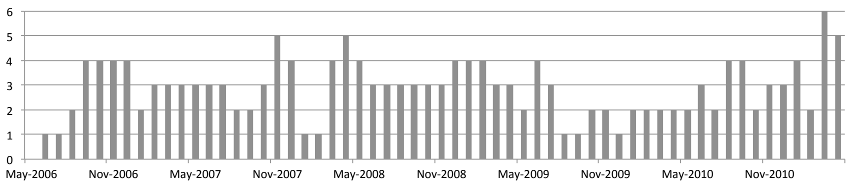

Fig. 4. Number of team members working on one L4.verified module over time.

non-local and can easily lead to edit-check cycles of multiple hours. This can be mitigated to a degree by reliably automating refactorings in offline nightly batch runs. Proof refactoring is an open, technically challenging research area, important for large-scale proof. Even simple renaming requires a deep semantic connection to the prover. Again, approaches like the PIDE [20] are promising, but create their own challenges in scaling to hundreds of large theories.

Debugging and performance. A frequent maintenance issue is proofs breaking after an update elsewhere in the project. In large proofs not written for readability, finding and repairing the cause of the breakage is challenging. Conventions for maintainable use of automated tactics help, but are not sufficient. We have found it useful to provide single-step versions of our automated tactics to help users pinpoint where proof searches go in unexpected directions. Extending this to a general design principle for all tactics would increase maintenance productivity, as well as helping beginners to understand what is happening under the hood. The same single-stepping analysis could be used to improve performance of proofs where automated methods run for too long. The right trade-off is important. For instance, Isabelle provides a tracing facility for term rewriting, but it easily overwhelms the user with information.

Context. Mathematical proof depends on context, i.e. a series of global assumptions and definitions under which one proves statements. Isabelle in particular has fine-grained context management that both projects could benefit from. However, proof context in Isabelle also determines which facts automated reasoners may use. On one hand, the concept of a background context has low mental overhead for the user if set up correctly. On the other hand, it introduces an extra-logical dependency on proof scripts. The same script might not work in a different position where the same facts are known, because the background context for automated tactics may differ. This makes it hard to move lemmas. Provers such as HOL4 [18] name such tactic contexts explicitly. This is not ideal either, since remembering their names within hundreds of theory files can be overwhelming. The Isabelle sledgehammer tool [5] provides an interesting solution: it searches in the global context, but explicitly indicates which facts were used in the proof, so that the proof now only depends on explicitly named entities. In theory this approach could be adopted for all proof search tools, making them context independent while possibly improving replay performance by reducing search. It would also answer a question frequently asked by new users: *which lemmas did automated method x use?*

2.4 Social and management aspects

Since proofs at the L4.verified and Verisoft scale are necessarily a team effort, social and management aspects play an important role. In this section, we concentrate on aspects that are either specific to proofs, or would not commonly be associated with proof development.

The main challenge is managing a team of proof engineers in a way which enables them to concurrently work towards the same overall proof goal with minimal overhead and duplication of work.

Discipline. In our experience, self-discipline alone is not sufficient for enforcing conventions and rules, be it for lemma naming, definitions, commit messages, or documentation. While self-discipline is a passable short-term (months) measure, adherence to conventions will deteriorate without incentive, explicit policing, or mechanical tests, especially as team members join and leave. We found that within the same group, this effect is stronger for proof development than for code. We speculate that two factors play a role: firstly, proof engineers get used to the theorem prover mechanically checking their work with immediate feedback for right and wrong; and secondly, theorem proving involves many concepts which need names. Given many similar concepts, labelling each lemma or definition with a useful name is surprisingly hard: it can take longer to think about a lemma's name than it takes to prove it. An effective renaming tool would allow the rectification of poor naming decisions and retroactive enforcement of conventions.

The pragmatic prover. An interesting challenge in managing a large-scale verification is striking the right balance between doing the work and developing tools to automate it. Duplication is a similar challenge. While duplication and proofs by copy and paste lead to obvious overhead, avoiding it can be arbitrarily hard. Both projects spent resources on increasing proof automation. In our experience, the views of the pragmatic programmer [10] are often directly applicable to proof: avoid duplication by automation wherever useful. Semi-regular reviews looking for automation opportunities can make a team significantly more productive. Semi-regular, informal code and proof reviews in general showed the same advantages as in usual software development of increased quality and cohesion within the team. In L4.verified, they were conducted mostly early in the project and in the later maintenance phase when new members joined the team. We should probably have held them more frequently.

The state of the proof. In a large-scale project it is easy to lose view of which parts of the proof currently work, which are under development, and which are broken. A nightly and/or continuous regression test can generate and display this information easily, as long as the prover provides a scriptable batch mode.

Concurrent proof. A key technical challenge in large-scale formal verification is efficient and useful distribution of proof sub-tasks to team members. Fig. 4 shows the number of L4.verified team members contributing per month to the module mentioned in Fig. 3. Up to six people worked on the same proof during development and maintenance. Ideally this is achieved by a compositional calculus that allows clean, small interfaces between sub-tasks, which are defined once and remain stable. In reality, this is rarely the case. Full compositionality often comes with a high price in other aspects of complexity, while interfaces are rarely small or stable. Nevertheless, work towards a common theorem could be distributed effectively in L4.verified by compositionality under side-conditions. Team members could work towards the same property on different parts of the system, or on different properties of the same part. They did not need detailed knowledge about the other properties or parts of the system [6]. When these side-conditions changed, effectively communicating them entailed an overhead. Such overhead is inevitable when for instance an invariant is discovered incrementally by the entire team. The faster the discovery, the smaller the overhead.

Distributing a proof over a team creates another management issue: avoiding that the separate pieces drift apart. An easy way of addressing this is to state the final top-level theorem first, and to use a continuous regression test to check that the parts still fit. Continuous integration avoids unpleasant surprises at the end of the project, such as discovering that an overly weak statement in one sub-proof requires significant rework in another. Note that proof interfaces do not always have to fit together perfectly. Software patterns like adapters and bridges can be applied to proofs, and, with suitable conversion lemmas, notions in one formalism can often be transferred into another. As always, there is a trade-off: the more bridging, the higher the overhead and the harder the proof becomes to maintain.

Active community. An active developer and user community around the main verification system is crucial. The absence of good documentation, online discussion, and fixes for problems could turn small annoyances into major show-stoppers. Active development also has a price, however. For instance, new proof assistant releases will not be fully backwards compatible—it would be detrimental if they were. Updating the proof base to the next prover release can add significant overhead. With one or two prover releases a year, a four-year project may confront a significant update 4–8 times. The L4.verified project invested in an update roughly once a year. Verisoft decided to stay with the 2005 Isabelle version, but set aside budget for back-porting important features of new releases. The latter approach works well for projects with a definite termination date. The former is more appropriate if the project goes into a longer maintenance period. A similar problem occurs in improving project- and domain-specific automation.

Libraries. Proof libraries play a role in any large-scale verification. The technical side of library development is handled well by mature theorem proving systems. Both projects made use of existing libraries and contributed back to the community. However, internal libraries are less polished and can become hard to manage. This occurs when the library is not explicitly maintained and serves only to accumulate roughly related lemmas. Sometimes this is appropriate, but producing a coherent and re-usable library requires a different approach. Good libraries do not emerge automatically, since library development takes significant effort and is usually not a key goal of the project. Verisoft members introduced the idea of a librarian [2], a person responsible for the consistency and maintenance of a particular library, allowing other team members to contribute lemmas as needed. While this alone is not sufficient to achieve a well-designed library, it does yield a much higher degree of usefulness and re-use. If possible, team leaders should set aside explicit budget for library and tool development. For both projects, the decision to do so resulted in increased productivity.

Intellectual property. Intellectual property (IP) is not a usual aspect of formal proof, be it copyright and licenses, patents, or non-disclosure agreements. Larger projects with an industrial focus are likely to involve such agreements, which may constrain the proof. For instance, an agreement may require that proof scripts related to a particular artefact be a partner's property, but not general libraries and tools. Classifying and separating each lemma according to IP agreements during development would be detrimental to the project. In L4.verified, automated analysis of theorem and definition dependencies identified proof parts specific to certain code artefacts, which were isolated with semi-automated refactoring.

3 Tool support for moving lemmas

In this section, we describe the design of a tool we developed to solve the lemma placement issue introduced in Sec. 2.2. The tool is called *Levity*, it was named for the idea that lemmas should float upward through a theory dependency graph to the position that maximises the potential for their reuse.

The fact that we developed a tool for cutting-and-pasting lemmas attests to one of the practical differences between small verification projects and larger ones. In a large development, reprocessing intervening files after moving a lemma to a new theory file may take tens of minutes or even hours; a loss of time and a distraction. For this reason, proof engineers in the L4.verified project resorted to adding comments before certain lemmas, like `(* FIXME: move *)` or `(* FIXME: move to TheoryLib_H *)`, with the hope of those lemmas being moved afterward. It turns out that moving lemmas involves more than just placing them after the other lemmas that they themselves require. A broader notion of theory context is required. For instance, lemmas can be marked with `[simp]`, at or after their declaration, to add them to the global set of lemmas that are automatically applied by the simplification method.

We think that a description of the design of Levity and the problems we encountered will be instructive for developers of other tools that manipulate interactive proof texts.

3.1 Design and implementation choices

Our first idea for implementing Levity was to use a scripting language and lots of regular expressions, but we finally decided to use Standard ML and to exploit APIs within Isabelle. This allowed us (a) to rely on the existing parsing routines, which is especially important since Isabelle has an extensible syntax, (b) to easily access the prover's state accumulated as theories are processed, and (c) to readily 'replay' lemmas so as to validate moves. The main disadvantage of this approach is that the APIs within Isabelle evolve rapidly. This not only necessitates regular maintenance, but poses the continual risk of obsolescence: a key feature used today may not be available tomorrow!

The second major design decision was to incorporate Levity into the nightly build process. Large proof developments are similar to large software developments. Proof engineers check-out the source tree to their terminals, state lemmas, prove, and commit changes back to the repository. Nightly regression tests check the entire proof development for errors. Levity is run directly after (successful) regression tests because an up-to-date heap is required for the extraction of lemma dependencies and the validation of lemma moves. We found that a second build is required after Levity runs and before committing any changes to the repository to ensure that the modified development builds without error. Having to run two full and lengthy builds is problematic because, as mentioned in Sec. 2, several hours may elapse from check-out to commit, and, especially in an active development, there may be intervening commits which necessitate merging, and, at least in principle, further test builds.

In the original design, we intended that Levity run regularly and with minimal manual intervention, automatically shifting lemmas to the most appropriate theory file during the night in readiness for the next day's proving, but it may in fact have been better to introduce some executive supervision. The destination theories chosen by Levity are optimal in terms of potential reuse, that is Levity moves lemmas upward as far as possible in the theory dependency graph, but they are not always the most natural; for instance, it is usually better to group related lemmas regardless of their dependencies. While the possibility of stating an explicit destination, or even a list of explicit destinations, in 'fix me' comments helps, it may have been even better to provide a summary of planned moves and to allow certain of them to be rejected or modified.

In the remainder of this section, we discuss the four main technical aspects of the Levity implementation: working with the theorem prover parser, calculating where lemmas can be moved, replaying proofs, and working with theory contexts. Although the technical details are specific to the Isabelle theorem prover and the task of moving lemmas, other tools for manipulating proof developments likely also need to manipulate the text of theories, interact with a theorem prover, and handle theory contexts.

Parsing. Levity processes theories one-by-one from their source files. In processing a file, it first calls the lexer routines within Isabelle to produce a list of tokens. Working with lists of tokens is much less error prone than working directly with text and avoids many of the complications of working with the sophisticated and extensible Isabelle/Isar syntax (any syntactically correct theory file can be processed). This is only possible because rather than filter out comments and whitespace, as is standard practice, the lexer returns them as tokens.

Levity processes each file sequentially, maintaining a list of tokens to remain in that file, namely those not marked for movement and those that could not be moved, and a list of tokens yet to be processed. It shifts tokens from the latter onto the former until it finds a 'fix me' comment followed by a lemma. As far as concerns parsing, two further details are important. First, a lemma's name and attributes must be extracted. There are already functions within Isabelle to do this, but we found that they were not general enough to be called directly and we thus had to duplicate and modify them within our tool. Such duplications make a tool harder to maintain as the underlying theorem prover evolves and may engender errors that type checking cannot detect. Second, the last token in the move must be identified. This is slightly more difficult than it may seem as the keywords that terminate a proof (like `done`, `qed`, and `sorry`) may also terminate sub proofs. And, furthermore, the extent of a definition depends on the syntax defined by a particular package which may be defined externally to the main theorem prover. Since, in any case, we simultaneously replay potential moves to check for problems, we use feedback from Isabelle to find the token that ends a proof or definition.

The tokens comprising a successful move are appended to a list maintained for the destination theory. Levity always appends lemmas to the end of a theory file. Inserting them between other lemmas would effectively require having to replay all theory files, both to detect the gaps between declarations and also to test moves in context. This would require more bookkeeping and take longer to run, but it would have the advantage of testing an entire build in a single pass. The tokens comprising failed moves are appended to the list of unmoved tokens.

Finally, theory files are remade from the lists of associated tokens, which works marvellously as Isabelle's command lexing and printing routines are mutually inverse.

Calculating lemma destinations. Before trying to move a lemma, Levity calculates the lemmas, and thereby theories, on which the proof depends. They are used to validate explicit destinations, and, when none are given, to choose the destination.

For calculating lists of lemma dependencies, we modified a tool developed within the Verisoft project, called *Gravity* [2], to handle sets of lemmas, to account for earlier moves, and to handle lemmas accessed by index. This tool works through a proof term—the low-level steps that construct a new lemma from existing lemmas, axioms and rules—to construct a dependency graph.

The calculation of destination theories in Gravity takes dependencies on lemmas into account, but not dependencies on constant definitions. Although not usually a problem in practice, as most proofs will invoke theorems about constants in the lemma statement, detecting such problems is difficult because constants moved before their definitions are interpreted as free variables.

Controlling Isabelle. When moving a lemma, respecting its dependencies is not enough. Besides the set of theorems, proofs also depend on several other elements of the theory context; for example, syntax declarations, abbreviations, and the sets of theorems applied by proof methods like simplification. Rather than try to determine all such dependencies in advance, we decided to simply try replaying lemmas in new contexts. The tool effectively replicates the actions of a proof engineer who, after copying-and-pasting a proof, must test it interactively.

Levity replays lemmas by parsing tokens, using Isabelle library routines, into commands which are applied through Isar, the Isabelle theory language. Isar operates as a state machine with three main modes: *toplevel*, *theory*, and *proof* [19, Fig. 3.1]. For each move, Levity initially executes a command to put Isar into *theory* mode, the first command drawn from the list of tokens being replayed should then cause a transition into the *proof* mode; if it does not, the move fails. Subsequent commands are then passed one-by-one to Isar, while monitoring the state for errors and the end of a proof. We use a time-out mechanism to interrupt long running proof steps to recover from divergences in automatic tools due to differences in the original and proposed theory contexts.

Lemmas are replayed at the end of destination theories, but theories cannot be extended after they have been closed, so Levity creates a new 'testing' theory for each replay. Each testing theory only depends on either the proposed destination theory, or on the last successful testing theory for that destination (to account for previous moves). In general, it is necessary to import, directly or transitively, the testing theories of all required lemmas that have also been moved.

Theory context. Interactive theorem proving inevitably involves the notion of context or state. At any point in a theory, there are the set of existing lemmas and definitions, syntax definitions, abbreviations, and sets of lemmas used by proof methods. Furthermore, Isabelle also has *locales* for fine-grained context management; lemmas stated within a locale may use its constants, assumptions, definitions, and syntax. Levity does not resolve all problems related to theory context, but it does address some aspects of locales and proof method sets.

To handle locales in Levity, the tokens being processed from a source file are fed through a filter that tracks the commands that open and close contexts[4] and maintains a stack of active declarations. An alternative would be to replay all commands through Isabelle and then to query the context directly. This would be more reliable but also slower. In any case, given this contextual information,

[4] Namely, `context`, `locale`, `class`, `instantiation`, `overloading`, and `end`.

Levity knows when a lemma is being moved from within a locale and it inserts tokens into the lemma declaration to re-establish the target context. The context stack also enables Levity to form fully-qualified lemma names.

As mentioned earlier, lemmas may be marked with *attributes*, like `[simp]`, to add them to proof method sets. Moving attributes with a lemma may interfere with subsequent applications of methods in other proofs. To avoid this, Levity parses attributes, strips certain of them, and inserts them as declarations at the original location. In general, such declarations, whether inserted by Levity or manually, should be tracked and considered when moving other lemmas. Consider, for example, lemmas l_1 and l_2, both marked for movement, l_1 having the `simp` attribute, and l_2 involving a simplification that implicitly uses l_1. Moving l_1 leaves the `simp` attribute in place. But now, moving l_2 may fail as the simplification step no longer implicitly uses l_1.

3.2 Experience and related work

We ran Levity several times against the main L4.verified project development during its final months. The results were encouraging, but several problems and limitations inhibited its permanent introduction into our build process. Besides the challenges of long build times, our biggest problems were unexpected dependencies and changes to libraries within Isabelle.

By unexpected dependencies, we mean that some lemmas became 'stuck' at seemingly inappropriate theories, and that other dependent lemmas then became queued after them.[5] Levity logs lemma dependencies before moving lemmas, which helps to understand why destinations were chosen, but then it is too late to do anything. When a lemma is not moved as far as expected due to dependencies on related lemmas, it is not uncommon that those related lemma should also have been marked for movement. Ideally then, some kind of manual review should be incorporated into the process; perhaps gathering information at night while performing approved moves, then requesting new approvals during the day.

Levity relies on internal Isabelle routines for parsing, analysing lemma dependencies, and interacting with Isar. When we went to prepare a public release for the latest version of Isabelle, we found that the interfaces to these routines had changed considerably from the version used in the L4.verified project, and, in fact, several essential features were no longer available. Careful compromise is needed between the evolution of a theorem prover's design and the availability of up-to-date third-party tools (and books and tutorials); both being important factors in the success of large verification projects.

Our inability to upgrade the ML version of Levity led to a rewrite [16] using the new PIDE interface to Isabelle [20]. This approach should make the tool more robust to changes within Isabelle. But we cannot yet comment on the efficacy of this tool against a large proof development like the L4.verified project: in particular, on its integration into nightly regression tests, and the effect of

[5] When a lemma move fails, other dependent lemmas are not moved either.

asynchronous recalculations while making automatic changes to large proofs. Development continues on this new version.

Whiteside et al. [21] address the subject of proof refactoring formally and in some generality. In particular, they define a minimal proof language, its formal semantics, and a notion of statement preservation. They then propose several types of refactoring, including renaming and moving lemmas (the latter defined as a sequence of 'swap' operations), define some of them and their preconditions in detail, and reason about their correctness. They explicitly do not *cover all aspects of a practical implementation*, and, in contrast to Isar, their proof language does not include proof method sets and attributes, definitions, locales, or imported theory dependencies. Nevertheless, we find such a formal approach a promising way to understand and validate refactoring tools like Levity; not just to show that they preserve correctness but also that they do not introduce build failures.

4 Summary

We have described challenges that are new or amplified when formal verification reaches the scale of multi-person, multi-year projects. We draw on the experience from two of the largest such projects: Verisoft and L4.verified. Many of these challenges arise from the inability of any single human to fully understand all proof aspects. Without a mechanical proof checker, such proofs would be infeasible and meaningless. For some challenges, we have sketched solutions, and for one, we have shown in more detail how it can be addressed by tool support.

The three most important lessons learnt from our verification experience are: First, proof automation is crucial because it decreases cognitive load, allowing humans to focus on conceptually hard problems. It also decreases the length of proof scripts, reducing maintenance costs. To achieve this, prover extensibility is critical and needs to allow for custom automation while maintaining correctness. Second, introspective tools such as *find_theorems* gain importance for productivity because effective information retrieval is necessary in an otherwise overwhelming fact base. Third, proof production at large scale hinges on an acceptably short edit-check cycle; any tool or technique that shortens this cycle increases productivity, even if temporarily sacrificing soundness.

Acknowledgements Thomas Sewell and Simon Winwood contributed to the requirements for the Levity tool of Sec. 3. We also thank June Andronick for commenting on a draft of this paper.

References

1. E. Alkassar, M. Hillebrand, D. Leinenbach, N. Schirmer, and A. Starostin. The Verisoft approach to systems verification. In N. Shankar and J. Woodcock, editors, *VSTTE 2008*, volume 5295 of *LNCS*, pages 209–224. Springer, 2008.
2. 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*, 42, Numbers 2–4:389–454, 2009.

3. K. Appel and W. Haken. Every map is four colourable. *Bulletin of the American Mathematical Society*, pages 711–712, 1976.
4. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
5. S. Böhme and T. Nipkow. Sledgehammer: Judgement day. In J. Giesl and R. Hähnle, editors, *Automated Reasoning (IJCAR 2010)*, volume 6173 of *LNCS*, pages 107–121. Springer, 2010.
6. D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *21st TPHOLs*, volume 5170 of *LNCS*, pages 167–182, Montreal, Canada, Aug 2008. Springer.
7. G. Gonthier. A computer-checked proof of the four colour theorem. `http://research.microsoft.com/en-us/people/gonthier/4colproof.pdf`, 2005.
8. G. Gonthier. Formal proof — the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.
9. M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *LNCS*. Springer, 1979.
10. A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Reading, MA, USA, 2000.
11. 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*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.
12. G. Klein, T. Nipkow, and L. Paulson. The archive of formal proofs. `http://afp.sf.net`, 2003.
13. X. Leroy. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In J. G. Morrisett and S. L. P. Jones, editors, *33rd POPL*, pages 42–54, Charleston, SC, USA, 2006. ACM.
14. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
15. P. Rudnicki. An overview of the MIZAR project. In *Workshop on Types for Proofs and Programs*, pages 311–332. Chalmers University of Technology, Bastad, 1992.
16. M. Ruegenberg. Semi-automatic proof refactoring for Isabelle. Bacherlorarbeit in Informatik, Technische Universität München, 2011. Undergraduate thesis.
17. N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.
18. K. Slind and M. Norrish. A brief overview of HOL4. In *21st TPHOLs*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008.
19. M. Wenzel. *Isabelle/Isar—a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München, 2002.
20. M. Wenzel. Isabelle as document-oriented proof assistant. In J. H. Davenport, W. M. Farmer, J. Urban, and F. Rabe, editors, *Calculemus/MKM*, volume 6824 of *LNCS*, pages 244–259. Springer, 2011.
21. I. Whiteside, D. Aspinall, L. Dixon, and G. Grov. Towards formal proof script refactoring. In J. H. Davenport, W. M. Farmer, J. Urban, and F. Rabe, editors, *Calculemus/MKM*, volume 6824 of *LNCS*, pages 260–275. Springer, Feb 2011.
22. S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish. Mind the gap: A verification framework for low-level C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *22nd TPHOLs*, volume 5674 of *LNCS*, pages 500–515, Munich, Germany, Aug 2009. Springer.