Deductive Verification of Legacy Code

Bernhard Beckert, Thorsten Bormer, and Daniel Grahl

Karlsruhe Institute of Technology

Abstract. Deductive verification is about proving that a piece of code conforms to a given requirement specification. For legacy code, this task is notoriously hard for three reasons: (1) writing specifications post-hoc is much more difficult than producing code and its specification simultaneously, (2) verification does not scale as legacy code is often badly modularized, (3) legacy code may be written in a way such that verification requires frequent user interaction.
We give examples for which characteristics of (imperative) legacy code impede the specification and verification effort. We also discuss how to handle the challenges of legacy code verification and suggest a strategy for post-hoc verification, together with possible improvements to existing verification approaches. We draw our experience from two case studies for verification of imperative implementations (in Java and C) in which we verified legacy software, i.e., code that was provided by a third party and was not designed to be verified.

1 Introduction

Formal software verification is the art of proving that a given implementation conforms to a specification on all possible inputs. Here, we consider deductive program verification at source code level, which is a precise verification technique for properties given in an expressive specification language. High precision means that neither false positive nor false negatives occur; there are no bounds on the domain; and no approximations or abstractions are needed to apply the technique.

Despite the considerable advances of verification technologies and improvements in the automation of theorem proving throughout the past decade, these tools are still highly dependent on user interaction to complete proofs.

One interaction paradigm, used by 'auto-active' [28] tools, is based on the user adding the information needed by the prover to the source code: the implementation is annotated with requirement and auxiliary specifications (e.g., loop invariants). If the auxiliary annotations are sufficient, this allows the tool to find a proof automatically (provided that the program actually meets its requirements).

There is a wide range of software verification tools users may choose from, depending on, e.g., the target programming and specification language, the kind of properties to be verified and, not least, the interaction paradigm that is followed: from (mainly) manual proof interaction in tools like Isabelle [30] or HOL4 [32], to purely auto-active tools such as Dafny [27] or VCC [9]. Taking the middle ground, tools like Why3 [6] or KeY [1] can be both used in an auto-active fashion and by manual interaction during the proof process.

Independently of whether an auto-active or an interactive verification style is used, the need to write formal (auxiliary) specifications has turned out to be the major bottle-neck (cf. [3]). The amount of specification typically outsizes the executable code: the ratio is reported to be 3:1 in [2] (measured in tokens), 4:1 in [7] (measured in lines of code), and almost 11:1 in [23] (measured in lines of proof script). In the light of these case studies, it becomes obvious why post-hoc verification is often unreasonably expensive: Writing specifications is a major part of the verification effort anyways, and that part is made more difficult and laborious if the program was not implemented with verification in mind.

The contribution of this paper is two-fold. Firstly, in Section 2, we discuss which characteristics of legacy code impede the verification effort and give examples. Secondly, we discuss how to handle the challenges of legacy code verification and suggest a strategy for post-hoc verification in Section 3, together with possible improvements to existing verification tools and methodologies.

Our observations are based on software written in imperative/object-oriented programming languages (C and Java). The conclusions drawn regarding the verification of legacy code are thus mainly relevant for other imperative implementations and only partially applicable to other paradigms like declarative or functional programming – e.g., while the difficulty to understand legacy systems applies to both imperative and functional programs, the need to handle shared, mutable state in specification differs between programming paradigms.

We argue that naive specification strategies that work either purely in a bottom-up or purely in a top-down fashion are ineffective: A too weak specification for some module M results in failure to prove properties about client modules invoking M; a too strong specification raises the verification effort beyond necessity.

We claim that, instead, a specification and verification process for legacy systems must be incremental and iterative. Also, using only a single verification tool and its methodology predetermines and often unnecessarily restricts the possible approaches to solve the legacy-verification problem. Instead, a good verification process supports proof exploration and construction with different tools; it lets the user apply different analysis and verification techniques for the various parts of a program and its specification. This choice is particularly important for legacy verification as the code has not been designed with a particular verification method or tool in mind.

We draw our experience from two case studies in which we verified (parts of) legacy software: the *PikeOS* microkernel [22] (using the VCC tool) and the *sElect* voting system [26] (verified with KeY). The PikeOS microkernel is part of a virtualization concept targeted at embedded real-time systems. It acts as a paravirtualizing hypervisor to support safety-critical and security-critical applications and is deployed in industry. The features of PikeOS – being part of highly safety-critical applications and having a manageable code size – made it a good choice for deductive verification.

The sElect e-voting system was developed by the group of Ralf Küsters at the University of Trier [26]. In this distributed system, a remote voter can cast one single vote for some candidate. The vote is sent through a secure channel to a tallying server. The server only publishes a result – the sum of all votes for each candidate – once all voters have cast their vote. The verification goal is to show that no confidential information (i.e., votes) is leaked to the public.

Although both systems feature concurrent computations, here, we restrict our considerations to sequential programs. Concurrency poses particular challenges that are out of the scope of this paper.

2 Why is Deductive Verification of Legacy Code Hard?

In the following we will illustrate some characteristics of legacy systems that contribute to the difficulty of post-hoc software verification, adding to the inherent complexity of any deductive verification task. We argue that coming up with the right specification is difficult already at the level of a single module. Specification is even more difficult when the right modularization (including any interface specification) has to be identified based on the legacy implementation. We also claim that current verification tools and methodologies, when each is used on its own, are often insufficient for large post-hoc verification tasks, because legacy code has not been written with a particular verification technique in mind. Thus, legacy code verification requires choosing appropriate tools and techniques on a per module basis and for individual specification parts, depending on the characteristics of the module and the property to be proven.

2.1 Legacy Code is Often Unsuitable for Verification

As part of our verification case studies we identified three causes why legacy code is hard to verify: (1) legacy code is difficult to understand; (2) the existing modularization of legacy systems is inadequate for verification and the right modules are hard to find; (3) implementations of single modules use programming language features or programming styles that are inherently difficult to verify, and the code is not written according to best practice of software development.

Both the structure of legacy systems and the implementation of single modules are often unsuitable for verification. It helps if the system was developed following best practices for software development; but that does not always lead to verifiable code. Rather, verifiability has to be considered explicitly when writing software – if not sufficiently taken into account, the following issues arise.

Legacy code is difficult to understand. As a prerequisite of most of the tasks in the verification process, the verification engineer has to understand the problem that the legacy system is trying to solve – in addition to the requirements to be verified. Understanding the problem at a level of detail needed for producing a *formal* requirement specification is often non-trivial.

For both the requirement specification and the auxiliary specification (respectively the user interaction with the prover in an interactive verification tool), information has to be extracted from (1) existing documentation, which is often only informal, incomplete or imprecise, or (2) the source code.

Although the source code contains the information needed, this knowledge is implicit and has to be made explicit by the verification engineer (e.g., in form of invariants) to be of use for the prover. Determining which information is crucial for the verification task that cannot be deduced automatically by the prover and how to adequately formalize the discovered properties is non-trivial.

An example for implicit knowledge in existing, informal documentation is the description of the effects of a system call in PikeOS that changes the priority of a thread (up to a bound named MCP), taken from the kernel reference manual:

"This function sets the current thread's priority to newprio. Invalid or too high priorities are limited to the caller's task MCP. Upon success, a call to this function returns the current thread's priority before setting it to newprio."

This description keeps effects caused by concurrency implicit: the system call is preemptible and another thread may change the thread's priority before the function's return value is assigned. The "old priority" returned thus might not be what a naive observer might expect who neglects that the system is concurrent.

Establishing the right system modularization is complicated. Ideally, real-world software would consist of modules that each have a single, clearly defined purpose, which would separate different concerns and lead to maintainable code. In reality, maintainability and elegance of an implementation is often not the first priority but also other quality metrics apply (e.g., efficiency). Software that is not developed according to best practices is often hard to maintain, more difficult to understand and analyze, hence also more work for formal verification.

Even if the system was developed according to best practices, the given modularization of the legacy system (given syntactically by the structure of methods, classes, etc., or described in the documentation) does often not coincide with a modularization that is optimal for verification purposes. One of the subtasks when verifying legacy systems is thus to find a better modularization for the implementation – this is a complex, iterative process.

Also, for post-hoc verification, the verification engineer can only change and optimize the modularization within boundaries given by the structure of the legacy system. Solutions of the modularisation problem are hence mostly stuck at a local optimum, which limits the suitability for verification that can be obtained.

When a reasonable division of the system into modules is found, the user still has to come up with the interface specification for each module. As a consequence of the (globally) suboptimal modularization of the system, the resulting interfaces and their specification become unnecessarily complicated in contrast to systems written for verification with a well-chosen structure.

Consider two components that together provide a single functionality to the rest of the system. If those components are treated for verification as separate modules, an additional interface is exposed that may be awkward to specify compared to the specification of a single module containing both components. In the other direction, combining two separate components that have almost no interaction on common state leads to specification overhead for describing the absence of interaction to modularize the verification task (frame problem).

Today, several approaches exist to tackle this problem, among them *dynamic frames*, *ownership* and *separation logic*. However, these approaches produce a considerable overhead in both specification and verification. In contrast, if components can be split into sub-components each operating on disjoint state, separating them already in the design and implementation phase diminishes the issue of framing overhead in the proof process.

The problem of writing interface specifications for legacy code is aggravated by interdependencies between modules. The specification and verification effort does not scale linearly with the number of modules due to interactions via shared data structures and common parts of the program states.

For example, in the PikeOS microkernel, implementations of single C functions are deliberately kept simple to facilitate maintainability and certifiability. The overall functionality of the kernel is implemented by interaction of many of these small functions, operating on common data structures (cf. [24]). More generally, all operating systems have to keep track of the system's overall state, resulting in relatively large and complex data structures on which many of the kernel functions operate conjointly. That this is not an exclusive property of system software but usual in non-trivial software projects in general is demonstrated by empirical studies on software complexity metrics (see, e.g., [5]).

As a consequence, interface specifications may have strong dependencies on each other – namely the joint data structures and their invariants. Finding the right auxiliary annotations for a single module requires the verification engineer to consider several modules at once, due to these dependencies.

For legacy systems, the (locally) optimal module structure established for post-hoc verification is more complicated and, thus, also the interface specification is more complex compared to systems written for verification. Therefore, each single attempt to find a suitable interface specification for a single module is also more complicated for legacy systems. The iterative nature of the specification process which is needed in case of highly interdependent modules acts as a multiplying factor for the disparity in specification effort at this local module level.

Single modules are badly written for verification. In the worst case, real-world code is produced with little care and low quality, resulting in buggy programs that not only fail to meet their requirements but are also difficult to analyze. Bugs in programs further increase the complexity of program verification as they introduce the uncertainty whether a proof cannot be completed due to a bug in the program or because of missing annotations or ineffective proof search.

Even if a program serves its purpose, the issue that this kind of code is often barely legible and badly maintainable remains. But even if code is well written in the sense that it is maintainable and adaptable, it is not necessarily easy to verify. A typical case is the frequent use of (standard) libraries, because the library functions may be hard to specify (e.g., string operations). Also, certain language constructs are notoriously difficult to specify and verify and should be avoided, e.g., the use of Java's reflection capabilities.

```
 1  private byte[] getResult() {
 2    if (!resultReady()) return null;
 3    int[] _result = new int[numberOfCandidates];
 4    for (int i=0; i<numberOfCandidates; ++i)
 5        _result[i] = votesForCandidates[i];
 6    return formatResult(_result);
 7  }
 8
 9  private static byte[] formatResult(int[] _result) {
10    String s = "Result of the election:\n";
11    for( int i=0; i<_result.length; ++i )
12      s += "Number of votes for candidate " + i + ": " + _result[i] + "\n";
13    return s.getBytes();
14  }
```

Listing 1. Code example from the sElect e-voting system.

Another issue with legacy programs are overly general and flexible implementations which can be used in a broad range of scenarios – which in the real system are then actually only used for a single, well-defined purpose (e.g., software product lines). If the verification engineer is unaware of this, a complex specification for the general functionality has to be provided instead of a more specific variant that clearly communicates the intended purpose of the module.

As an example for code that is written in a way that it is hard to verify, consider the implementation from the original e-voting system [26] that retrieves the election result in the server (Listing 1). Several issues are to be noted: (a) The method `getResult()` returns a null reference in case it is called in an illegal state (Line 2). (b) The array containing the number of votes for each candidate is copied to a fresh instance (Line 5). (c) The result is embedded into a string (Line 12) and encoded into an array of bytes (Line 13). Item (a) represents a common modeling pattern, even though in good object-oriented design, it is preferable to raise an exception. Returning `null` (or any other error element) does not complicate verification, but must be reflected in the specification. Item (b) is just superfluous code – we could pass the original array reference. Both the allocation of a fresh array and copying the values invokes unnecessary complexity in verification – and also in the specification since we need an invariant for the loop. Item (c) is the most serious: The encoding in strings effectively makes verification extremely difficult, even when support for reasoning about strings is available.

2.2 Lack in Tool Support for Post-hoc Verification

We argue that the issues pointed out so far are not sufficiently mitigated by supporting measures in most deductive verification tools and methodologies.

Handling large software systems is supported by modular specification and verification. Modular verification is one of the advantages of deductive verification. But at the same time, modularity is also *essential* for deductive verification tools to scale at all. The need for auxiliary specifications that comes with modularity can be a drawback. For instance, while KeY supports inlining of method calls – which is suitable at least for smaller methods – VCC does not provide that option and the verification engineer has to provide method contracts for all methods.

There is often no good support for inspecting and understanding the interplay between different modules of a given program, as current deductive verification tools and methodologies tend to focus on specifying and verifying a single module at a time. Instead, verification tools should provide a view on dependencies of module specifications and make effects of local specification changes to the rest of the system explicit. Already the task of determining which previously completed proofs have to be re-run after a change to an auxiliary specification that is exported to other modules is often not sufficiently supported by verification tools. To assist the user, the integration of KeY into the Eclipse IDE [20] tracks dependencies between proofs for a system, automatically tries to re-run proofs affected by a change in either the program or specification and notifies the user of the proof result.

To reduce the effort needed to verify interdependent modules, techniques such as abstract operation contracts [8] or lazy behavioral subtyping [12] can be used: The former approach allows to compute and cache parts of the proofs that are independent of a given concrete specification, while the latter approach simplifies verification of object-oriented programs by reducing contracts of overriding methods to those properties actually needed at the call sites of the superclass methods.

Abstraction is another important instrument to handle verification of large systems. Good abstraction of the behavior of a system helps to focus on important details of the functionality, and allows for clear and succinct specifications. Poorly chosen abstractions may complicate verification up to making it impossible. Which abstraction is appropriate depends on both the system properties to be verified and on how well the verification tool is able to reason about the abstraction.

To find the right abstraction for a data structure, analyzing its implementation alone is often not sufficient. Rather, one has to find out which properties of the data structure are important for verifying the modules using it. While techniques exist that may help in some cases in finding the right abstractions (such as CEGAR), these methods are not sufficiently supported in current annotation-based systems.

Moreover, specification-language support for abstractions is often not flexible enough. For data abstraction, most verification systems feature some kind of user-defined abstract data types – however, there is a large amount of established formalisms, like CASL, that should be taken into further consideration when extending the specification language. For control abstraction, many established formalisms exist that could be used for one of the abstraction layers on top of the code, e.g., CSP or abstract state machines. Also, a built-in refinement mechanism is needed to connect the different abstraction levels.

3 Ways to Successful Post-Hoc Verification

Given the difficulties one faces when applying deductive verification to legacy code, one may consider a re-implementation of the verification target from scratch with formal verification in mind. In particular, if full functional verification of the whole software system is required, this may well lead to less effort than a legacy code verification. In practice, re-implementation is rarely the best option, as several reasons call for verification of existing code in its original context in a legacy system: (1) To be formally verified is not the only quality the software is measured against; the newly written code has to be, e.g., as efficient or as maintainable as the legacy version. (2) Existing knowledge of the development team about the legacy implementation, and also documentation would be largely rendered worthless in case the software was written from scratch. (3) Often, full functional verification of the whole system is not necessary as either a smaller set of important parts of the system or only specific characteristics of the system is of interest (e.g., security properties such as absence of certain information flow).

Another important point is that the user's trust in a system to perform as expected, which has been developed by extensive testing or long-term use, cannot simply be replaced by the fact that the system has been formally verified.

3.1 A Verification Process Based on Separation of Concerns

To handle the challenge of verifying complex legacy software, we have to split up and simplify the specification and verification task by decomposing the verification problem into parts, which we call *verification concerns*. A concern consists of some part of the code together with part of its specification. The main goal is to arrive at a small set of simple concerns that are easy to specify and verify *in isolation* – as multiple verification attempts are often required before successfully completing a single proof and thus repeated effort in user interaction is the normal case, this isolation prevents propagation of revisions through the rest of the program and specification. As explained below, concerns are related to but not identical to the modules of the program to be verified. Moreover, concerns within one verification project may be formalised using different specification methods. And they may be intended for different validation methods, which – besides verification – may include testing or inspection for some concerns.

There are four main strategies that may be applied to handle a concern $C = (S, P)$, consisting of a (requirement) specification S and an implementation part P: decomposition, abstraction, substitution, and local verification.

Decomposition. To decompose a concern (S, P), the program P is partitioned into modules and corresponding interface specifications are added for each of the resulting modules. For example, if we have a proof sketch for the correctness of P w.r.t. specification S and have identified how different components of P contribute to its correct operation, we can use decomposition to get a new set of concerns with (possibly informal) requirements for smaller parts of P.

A special case is decomposing a program P "in situ" by marking out parts of a method body with specification constructs without affecting the actual code structure – e.g., in KeY, the user can enclose parts of a method body in a Java block and give it a contract; this allows to split large methods into more manageable pieces.

Another possibility to isolate functionality of a system is to extract and aggregate related methods of the implementation in a trait which can then be reasoned about using an incremental deductive verification approach [10].

Besides dividing the implementation, also the specification can be split up into different concerns (e.g., termination, information flow properties, functional properties) or different cases depending on the input. The different execution paths in the implementation that fit the specification parts may then also be isolated by choosing a subset of relevant or interesting statements for further analysis (resulting in a program slice).

That verifying the decomposed concerns implies validity of the original concern is either another explicit proof obligation in the verification process (i.e., is a concern itself) or is entailed by a general argument about the decomposition step.

Abstraction. Simplifying a concern by using control or data abstraction allows for hiding implementation details irrelevant for the underlying reasons of correct operation of the concern – any details removed in such a way then only appear in a separate refinement proof obligation, i.e., as additional concern in the process.

Typical examples include the abstraction of non-trivial implementation of pointer-based data structures by a suitable data type like sequences or sets, or providing an interface specification for more involved operations which is underspecified (e.g., replacing pivot selection in Quicksort by random choice). Also, producing a prototype is a special case of abstraction.

Substitution. Both the program and the specification part of a concern C can be replaced by a version C' that is optimized for further treatment in the verification process. In this case, all completed proofs that depend on C are rendered invalid and have to be reinspected and possibly redone with the new concern C'. In contrast to prototype construction, however, this approach does not need to justify the relation between C and C'. Instead, C' is the result of the process.

Local Verification. At some point, the resulting concerns cannot further decomposed, abstracted, or substituted. They then have to be verified correct in a *local* verification step with a suitable technique. Verification techniques range from interactive deductive verification, more lightweight automatic static checkers, up to testing and run-time checking – or simply adding the correctness of the concern to the set of assumptions made for correct operation of the whole system.

As a prerequisite for verification, the concern has to be prepared (e.g., by translating the specification S to another specification language – one special case is formalizing an informal requirement specification of the concern). Lastly, auxiliary specifications for the concern have to be added and a verification attempt is made.

3.2 Activities in the Concern-centric Verification Process

The defining concepts for a concern-centric verification process are: (1) different verification and validation methods are used for different types of concerns within one project; and (2) different operations on the set of concerns are applied in an *iterative* and *incremental* fashion. How to find the right concerns and how to handle each concern depends on the program to be verified and the methodology used.

Identify and verify concerns by lightweight techniques. In many cases, program correctness (or incorrectness) can be judged by automated light-weight approaches. These approaches are very efficient (in comparison with deductive verification), but cannot be sound and complete at the same time. Combining deductive verification with one of these in a hybrid approach allows us to cut the cost of verification while maintaining soundness and completeness. Suitable technologies include bounded software verification [13], runtime checking [11] and testing, as well as program slicing [17], or invariant generation.

Not only can we make use of these techniques to *verify* particular concerns, they also allow us to *identify* components resp. concerns in the first place.

In the e-voting case study, the critical code for counting votes is interleaved with calls to a logger. Intuitively, logging does not interfere with computing the result. However, it does change the global state. Deductive verification (with the KeY tool) thus includes the concern of proving that logging does indeed not affect the election result, which is expensive. We successfully used decomposition based on a slicing to compute a (smaller) critical slice within the actual code [25], which does not include the logging concern. This allowed us to verify the original code under the assumption that logging does not change the global state, which is justified at the meta-level (correctness of the slicing technique).

Refactor the implementation to simplify verification. Precise instructions on how to refactor a program to ease the verification task can only be given w.r.t. a particular verification technique and a particular target program. An easy to verify module is simple w.r.t. control flow and data flow. In general, the target modules should be implemented in a way such that they only provide functionality that is necessary for the overall system functionality.

For an example of how to improve existing code that is not written for verification, reconsider the code shown in Listing 1 and its shortcomings described in Section 2.1. In our prototype, we have drastically simplified the functionality:

```
private int[] getResult()
   { return resultReady() ? votesForCandidates : null; }
```

We omit the copying and the encoding in string format, which are separate verification concerns, and return the original array. However, we retain the error reporting through returning `null` in order not to deviate too far from the original design.

Another local optimization is to decouple control flow and data flow, where possible. For example, the program fragments `if (b) a = x; else a = y;` and `a = b? x: y;` are equivalent, yet the first one combines control flow and data flow, whereas in the second one, only data flow occurs. For the latter version, in KeY's symbolic execution engine, the location `a` is assigned a symbolic value that depends on the value of `b` but the proof does not branch.

Produce prototypes to understand a verification concern. Sometimes, the measures presented so far are not enough and we need even more invasive changes to enable verifiability. With the e-voting case study, we pursued an approach in which we produced a series of gradually more complex prototypes, which were verified one after another [16, Chap. 9] – similar to a refinement-based development style to produce verified code. In this way, there is quick feedback on the validity of the more abstract specifications. While the code change between each version was rather small, the specification grew significantly. Still, we have found that it is harder to develop the complete specification for the final prototype in one step (or even the actual implementation) than to refine it on every iteration.

3.3 Where to Start the Process?

Given a software system to be verified, an important question is whether to attack the verification problem in a top-down or in a bottom-up manner.

In a top-down approach, we start with the (usually informal) high-level requirements and see how they distribute to single modules. This approach bears the advantage that we focus on the overall goal. On the other hand, it comes with a danger that the formalization of requirements is not well adapted to the modules. Typically, too weak preconditions are derived where side-conditions – in particular implementation-related – were not considered on the higher abstraction level, e.g., size restrictions of data structures. The unpleasant consequence is that we have to refine many module specifications and to repeat all affected proofs.

In a bottom-up approach, we start by specifying and verifying the most elementary modules (i.e., leaves in the call graph). Then, specifications of larger modules are derived by composing the specification of constituents. An obvious benefit of this approach is that elementary modules are of little complexity, hence it is not too difficult to develop a precise specification. We can make post-conditions strong enough that we can reuse the contracts of these components without the danger of having to repeat its correctness proof. This insight is particularly important for (helper) modules that are called often in the system under investigation. However, a bottom-up approach tends to be expensive. Firstly, there is a high human effort in exhaustively specifying the modules. Secondly, the resulting contracts may not be effectively usable, because a precise specification may consider more cases than are necessary in the given verification context. In particular, all corner case are specified, instead of excluding them from consideration through the pre-condition, e.g., it is easier to require that an array access is within bounds than to specify the effect of an illegal access.

Besides the logical strength of a contract, also its syntactic form is important for its utility in conducting a proof. Consider the two alternative postconditions for a function `sqrt` computing the integer square root of `x`:

(a) $\texttt{\textbackslash result}^2 \leq \texttt{x} < (\texttt{\textbackslash result+1})^2$

(b) $(\forall \texttt{y. } 0 \leq \texttt{y} \leq \texttt{\textbackslash result} \Rightarrow \texttt{y}^2 \leq \texttt{x}) \land (\forall \texttt{z. z} > \texttt{\textbackslash result} \Rightarrow \texttt{z}^2 > \texttt{x})$

While both contracts specify the same behavior of `sqrt`, one contract may be much more useful than the other, depending on the verification tool used and the properties that are needed in the verification of a caller of `sqrt`.

For these reasons, we claim that pure top-down nor pure bottom-up approaches are seldom effective. Instead, we have to start at several points simultaneously and have to refine our specification in short iterations. In this way, higher-level requirements and lower-level guarantees can converge. Choosing the concrete approach depends on the program structure. Analyzing the connectivity in the call graph first, gives a good heuristic. Strongly connected components in the call graph are recursion groups. Within them, a bottom-up approach is not possible at all, but a top-down approach can start at the node with the highest incoming connections.

For our considerations, loops behave similarly to internal nodes in the call graph with indegree and outdegree of one: while it is possible to start with the specification of a loop with invariants before writing the contract of the surrounding method, often the loop invariant is not of interest on its own as part of a requirement, but simply an auxiliary specification that enables verification of the method contract. As such, an invariant has to fit to this contract both regarding the logical strength and its syntactical structure, as shown in the `sqrt` example above. Consequently, you do not start with writing down the loop invariant, but derive it from the surrounding method's contract while abstracting.

In the e-voting case study, the modules are arranged hierarchically – without recursive method calls. This allows developing specifications, including class invariants, mostly bottom-up. However, we find it useful to have a good control over when invariants are applied within a proof. Many verification systems offer little user control over how invariants are processed. In contrast, KeY represents class invariants using a symbol in the proof obligation which can be replaced with the actual contents of the invariant only if and when needed.

This allows using the abstraction provided by invariants not only in specification, but also in the proof, since it is often enough to refer to 'the invariant' without knowing its exact contents. A similar concept exists in Dafny with *opaque* functions [18], where the user can decide when to make the body of such a function available to the prover. These mechanisms are helpful in cases where the contents of an invariant or the function body are complex, e.g., if an invariant contains existential quantification.

3.4 How to Improve Tool Support for Post-Hoc Verification?

Effective verification requires good feedback to the user. In a purely interactive proof, the user has full control, but the amount of available information may be too much to handle and is sometimes not given at the right abstraction level (e.g., showing open proof goals instead of notifying the user about which specification is violated in the source code). Several techniques may improve user experience:

High-level user interaction. Constructing a proof interactively without any automation is infeasible for practical verification problems. In the e-voting case study, we encountered single proofs with over 200 000 proof rule applications, where single formulas in the proof goal could fill several screen pages.

Instead of fine-grained manual interaction, user input relevant for proof search and construction should be given in a way that is close to the problem description respectively the implementation. One possibility is to follow the auto-active verification paradigm, giving auxiliary annotations at source-code level.

Another approach is to provide an interaction concept that matches an abstract proof outline of the user more closely, e.g., by using proof scripts – this well-established interaction paradigm is the basis of many interactive provers, e.g., HOL4 [32] or Isabelle [30] with "tactic-based" proof interaction. One example of a more declarative interaction style compared to the procedural style is the Isar formal proof language offered in the Isabelle/Isar system [33].

Similar to these established script-based interaction approaches, proof scripts in KeY serve as a high-level interface to the proof object as the user does not apply concrete single rules, but sketches the proof structure. One use case is a long proof with only a few steps that need interaction. Scripts can often be replayed when a verification concern is modified, as they are robust against smaller changes in the code or its specification.

Better feedback. One of the main issues in verifying large systems are the complex dependencies between verification concerns and the associated correctness proofs. To simplify the task of keeping track of the overall proof state, as well as updating this information in case a verification concern has been changed and proofs have to be redone, the KeY tool has been integrated into the Eclipse IDE [20].

Identifying errors in the specification or implementation during the verification process is another frequent issue, in particular when large proof obligations arise in interactive provers which require manual inspection or user interaction – often, only a small part of the information presented in the proof obligation is relevant for revealing the error. To automatically get quick feedback on the validity of such a proof obligation, a bounded analysis technique has been implemented for the KeY tool, giving concrete counterexamples for single KeY proof obligations [21]. Support for pinpointing the reason for incomplete proof attempts is also provided by another component of KeY's integration into the Eclipse IDE mentioned previously, by giving a view on KeY's proof goals that shows the truth status of subformulas as inferred by KeY in the current proof state [19].

For auto-active verification tools, the situation is often the opposite: too little information is available to pinpoint possible errors or missing specifications. The insight that these tools often do not produce enough feedback for failed verification attempts is not new [28,31]. To improve this situation, tools like VCC, which already show the exact annotation that could not be verified, are usually complemented with tools like the Boogie Verification Debugger (BVD) [15], presenting counterexamples for the proof obligations on the level of the original program.

Still, due to the modular verification methodology of the deductive verification tools, these counterexamples are often not sufficient to retrace the concrete program execution leading to the violated specification (if there is any). This shortcoming has been identified and is addressed by many approaches. To give early and precise feedback beyond the local module currently being verified, we proposed a combination of software bounded model checking and deductive verification [4]. Other techniques allow for generating a program reproducing a concrete trace through the original code from a failed verification attempt [29,31], allowing the user to identify mismatches between program and specification.

Regression verification is an instance of relational program analysis [14]. Instead of asserting that a program conforms to its contract, we prove that *two programs* are functionally equivalent (or more generally, that they expose congruent behavior). While, in general, relational verification is as hard as functional verification, regression verification works well for programs that are of a similar (syntactical) shape with only minor local differences. Proof complexity does not stem from the overall program/specification complexity, but only from the difference between the two versions. This allows reducing the effort of repeating a proof for a module with only minor changes. We can use this property to verify a prototype implementation P first, and then prove that a refined version P' of the code matches the behavior of the prototype, which proves that P' conforms to the specification. This approach is particularly helpful in case where the actual legacy code must not be changed.

4 Conclusion

A pure post-hoc deductive verification approach for full functional requirement specifications is often unreasonably expensive, despite recent advancements of specification methodologies and verification tools.

It is thus crucial to identify and separate different concerns (i.e., parts of the property to be proven and portions of the implementation) to take advantage of different program analysis techniques – in the best case discharging otherwise complex deductive proof obligations automatically, e.g., by syntactic analysis such as program slicing. These analysis techniques not only play a crucial role in proving system properties, but also help with the identification of concerns as part of an iterative specification and verification process.

Any remaining concern that can only be proven correct using deductive verification tools like KeY or VCC, which often requires extensive user interaction, should be preprocessed and rewritten to ease verification: examples are property-preserving program refactorings, writing more abstract variants of the implementation and proving refinement between the different versions – or, if necessary, re-implement relevant parts of the system from scratch.

At times, also implementations that follow best practices of software engineering are needlessly complex from the point of view of software verification. Raising awareness of these issues for program verification would allow improving the state of implementations w.r.t. verifiability when legacy code is changed or new modules are implemented. In addition, verification complexity metrics or implementation and design patterns adapted to program verification could provide guidelines for how to produce code that is easier to analyze and verify using formal methods.

References

1. Ahrendt, W., Beckert, B., Bruns, D., Bubel, R., Gladisch, C., Grebing, S., Hähnle, R., Hentschel, M., Herda, M., Klebanov, V., Mostowski, W., Scheben, C., Schmitt, P.H., Ulbrich, M.: The KeY platform for verification and analysis of Java programs. In: Giannakopoulou, D., Kroening, D. (eds.) Verified Software: Theories, Tools, and Experiments (VSTTE 2014). pp. 1–17. LNCS 8471, Springer (2014)
2. Alkassar, E., Hillebrand, M.A., Paul, W.J., Petrova, E.: Automated verification of a small hypervisor. In: Leavens, G.T., O'Hearn, P.W., Rajamani, S.K. (eds.) 3rd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2010). pp. 40–54. LNCS 6217, Springer (2010)
3. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Lessons learned from microkernel verification: Specification is the new bottleneck. In: Cassez, F., Huuck, R., Klein, G., Schlich, B. (eds.) 7th Conf. on Systems Software Verification. SSV 2012, Sydney, Australia. No. 102 in Electronic Proc. in Theoretical Computer Science (2012)
4. Beckert, B., Bormer, T., Merz, F., Sinz, C.: Integration of bounded model checking and deductive verification. In: Formal Verification of Object-Oriented Software International Conference, FoVeOOS 2011, Turin, Italy, October 5-7, 2011, Revised Selected Papers. pp. 86–104. LNCS 7421, Springer (2012)
5. Bhattacharya, P., Iliofotou, M., Neamtiu, I., Faloutsos, M.: Graph-based analysis and prediction for software evolution. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) 34th Intl. Conf. on Software Engineering (ICSE 2012). pp. 419–429. IEEE (2012)
6. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd Your Herd of Provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64. Wroclaw, Poland (2011)
7. Bruns, D., Mostowski, W., Ulbrich, M.: Implementation-level verification of algorithms with KeY. Software Tools for Technology Transfer 17(6), 729–744 (2015)
8. Bubel, R., Hähnle, R., Pelevina, M.: Fully abstract operation contracts. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications (ISoLA 2014), Imperial, Corfu, Greece, October 8-11. pp. 120–134. LNCS 8803, Springer (2014)
9. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics (TPHOLs 2009). pp. 23–42. LNCS 5674, Springer (2009)
10. Damiani, F., Dovland, J., Johnsen, E.B., Schaefer, I.: Verifying traits: an incremental proof system for fine-grained reuse. Formal Asp. Comput. 26(4), 761–793 (2014)
11. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: Shin, S.Y., Maldonado, J.C. (eds.) Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013. pp. 1230–1235. ACM (2013)
12. Dovland, J., Johnsen, E.B., Owe, O., Steffen, M.: Lazy behavioral subtyping. Journal of Logic and Algebraic Programming 79(7), 578–607 (2010)
13. Falke, S., Merz, F., Sinz, C.: The bounded model checker LLBMC. In: Denney, E., Bultan, T., Zeller, A. (eds.) 28th IEEE/ACM Intl. Conf. on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA. IEEE (2013)
14. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: 29th IEEE/ACM International Conference on Automated Software Engineering (ASE 2014). pp. 349–360. ACM (2014)

15. Goues, C.L., Leino, K.R.M., Moskal, M.: The Boogie verification debugger (tool paper). In: Barthe, G., Pardo, A., Schneider, G. (eds.) 9th Intl. Conf. on Software Engineering and Formal Methods (SEFM 2011). LNCS 7041, Springer (2011)
16. Grahl, D.: Deductive Verification of Concurrent Programs and its Application to Secure Information Flow for Java. Ph.D. thesis, Karlsruhe Inst. of Techn. (2015)
17. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security 8(6), 399–422 (2009)
18. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B.: Ironclad apps: End-to-end security via automated full-system verification. In: Flinn, J., Levy, H. (eds.) 11th USENIX Symposium on Operating Systems Design and Implementation. pp. 165–181. USENIX Association (2014)
19. Hentschel, M.: Integrating Symbolic Execution, Debugging and Verification. Ph.D. thesis, Technische Universität Darmstadt (Jan 2016)
20. Hentschel, M., Käsdorf, S., Hähnle, R., Bubel, R.: An interactive verification tool meets an IDE. In: Albert, E., Sekerinski, E., Zavattaro, G. (eds.) Proc. of the 11th Intl. Conference on Integrated Formal Methods. pp. 55–70. LNCS, Springer (2014)
21. Herda, M.: Generating Bounded Counterexamples for KeY Proof Obligations. Master's thesis, KIT (2014), `http://dx.doi.org/10.5445/IR/1000055929`
22. Kaiser, R., Wagner, S.: Evolution of the PikeOS microkernel. In: Kuz, I., Petters, S.M. (eds.) 1st International Workshop on Microkernels for Embedded Systems (MIKES 2007). National ICT Australia (2007)
23. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an operating system kernel. Comm. ACM 53(6) (2010)
24. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Transactions on Computer Systems 32(1), 2:1–2:70 (2014)
25. Küsters, R., Truderung, T., Beckert, B., Bruns, D., Kirsten, M., Mohr, M.: A hybrid approach for proving noninterference of Java programs. In: Fournet, C., Hicks, M., Viganò, L. (eds.) 28th IEEE Comp. Security Foundations Symp. (CSF) (2015)
26. Küsters, R., Truderung, T., Vogt, A.: Verifiability, Privacy, and Coercion-Resistance: New Insights from a Case Study. In: Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P). pp. 538–553. IEEE Computer Society (2011)
27. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16). pp. 348–370. LNCS 6355, Springer (2010)
28. Leino, K.R.M., Moskal, M.: Usable auto-active verification. In: Usable Verification workshop (2010), `http://fm.csl.sri.com/UV10`
29. Müller, P., Ruskiewicz, J.N.: Using debuggers to understand failed verification attempts. In: Butler, M., Schulte, W. (eds.) 17th International Symposium on Formal Methods (FM 2011), pp. 73–87. LNCS 6664, Springer (2011)
30. Paulson, L.C.: Isabelle – A Generic Theorem Prover. LNCS 828, Springer (1994)
31. Polikarpova, N., Furia, C.A., West, S.: To run what no one has run before: Executing an intermediate verification language. In: Legay, A., Bensalem, S. (eds.) 4th Intl. Conf. on Runtime Verification (RV 2013). pp. 251–268. LNCS 8174, Springer (2013)
32. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C.A., Tahar, S. (eds.) Theorem Proving in Higher Order Logics (TPHOLs 2008), Montreal, Canada, August 18-21, 2008. Proceedings. pp. 28–32. LNCS 5170, Springer (2008)
33. Wenzel, M.M.: Isabelle/Isar—a versatile environment for human-readable formal proof documents. Ph.D. thesis, Technische Universität München (2002)