Lessons Learned From Microkernel Verification
Specification is the New Bottleneck

Christoph Baumann[1] Bernhard Beckert[2] Holger Blasum[3] Thorsten Bormer[2]

[1]Saarland University, Saarbrücken, Germany
[2]Karlsruhe Institute of Technology, Karlsruhe, Germany
[3]SYSGO AG, Klein-Winternheim, Germany

Software verification tools have become a lot more powerful in recent years. Even verification of large, complex systems is feasible, as demonstrated in the L4.verified and Verisoft XT projects. Still, functional verification of large software systems is rare – for reasons beyond the large scale of verification effort needed due to the size alone. In this paper we report on lessons learned for verification of large software systems based on the experience gained in microkernel verification in the Verisoft XT project. We discuss some issues that impede widespread introduction of formal verification in the software lifecycle process.

1 Introduction

In recent years, deductive program verification tools have made significant progress. Full functional verification of individual functions written in real-world programming languages is practicable with reasonable effort. Verifying large and complex software systems is also feasible, as shown in the L4.verified and Verisoft projects using system software as the verification target [21, 6].

Naturally, verifying non-trivial software systems requires substantial effort due to the size of the system alone. In addition, however, even with modern specification methodologies and verification tools, verification of a large software system suffers from scalability issues. Like in the software development process – where the effort of implementing a complete system is more than the sum of implementing its isolated components – the effort of specifying and verifying a real-world system is more than the sum of verifying its components.

In this paper we report on lessons learned from the Verisoft XT project. In the context of this project, core parts of the embedded hypervisor PikeOS (see `http://www.pikeos.com`) have been verified using the VCC [14] verification tool. While the size of this microkernel is several orders of magnitude smaller compared to, e.g., the Linux kernel, for functional verification this can be considered to be a substantial code size.

The issues presented in the paper are drawn from our experience with the PikeOS case study. Some of these issues surprised us while others were expected to occur from the beginning of the project on. But even for the expected problems, their impact on the verification effort often deviated from our anticipations. For example, the difficulty to verify complex algorithms and data structures turned out to be of little concern in the project (as the implementation of PikeOS avoided this complexity for good reason).

The two main conclusions from our work in the Verisoft XT projects are: firstly, verification of complex concurrent software systems can be successfully done and, secondly, given the power of modern verification tools, not verification but *specification* is the real bottleneck for large software systems. We argue that the latter insight applies not only to microkernels but to most non-trivial software system.

The structure of the paper is as follows: In Sect. 2, we present our verification setup in the Verisoft XT avionics subproject, introducing our verification target PikeOS, the used verification tool VCC, as well as our verification approach. Sect. 3 presents the main contribution of this work, namely issues in verifying large software systems, together with approaches to deal with them. This is followed by related work in Sect. 4 and conclusions in Sect. 5.

2 The Verisoft XT Avionics Project

2.1 The PikeOS Microkernel

PikeOS is a virtualization platform deployed and used in industry. It consists of an L4-based microkernel acting as paravirtualizing hypervisor and a system software component. PikeOS is written to run on many platforms, including x86, PowerPC, MIPS, and ARM among others. While PikeOS is able to make use of multi-processor setups, we only considered the single-processor configuration for Verisoft XT – this decision confines concurrency in the system to preemption of user processes and interrupts in the case of the kernel.

The PikeOS kernel is tailored to the context of embedded systems, featuring real-time functionality and resource partitioning. The system software component is responsible for system configuration. Together with the system software, the PikeOS kernel provides partitioning features that allow to virtualize several applications, such as operating systems or run-time environments, on one CPU, where each application runs in a secure environment with configurable access to other partitions if desired. In order to provide real-time functionality, there are many regions within the kernel code where execution may be preempted – we thus have a concurrent kernel. Moreover, the kernel is multi-threaded.

At the kernel level, the mechanisms for communication between threads are IPC, events, and shared memory. High-level communication concepts such as ARINC ports can be mapped onto these kernel-level mechanisms. For a thorough discussion of the evolution of PikeOS, see [19].

Most parts of the PikeOS kernel, especially those that are generic, are written in C, while other parts that are close to the hardware are necessarily implemented in assembly. While the exact amount of assembly depends on the architecture one works on, in our particular case, PowerPC assembly is about one tenth of the codebase.

One consequence of using an existing and widely deployed system as verification target is that we could not modify the code base to make it more amenable to verification. In addition, PikeOS has not been written with deductive verification (as used in VCC) in mind. On the other hand, because of the context of the application domains of PikeOS (especially avionics), the implementation avoids overly complex (and possibly more efficient) implementations in favor of maintainable, adaptable, and robust code. PikeOS has been developed to meet the DO-178B [23] standard, which supports this claim.

2.2 The Verification Tool and Methodology: VCC

VCC is a deductive verification tool for concurrent C programs at the source code level that is used to prove correctness of C implementations against their functional specification. The VCC toolchain allows for modular verification of concurrent C programs using function contracts and invariants over data structures. Function contracts are specified by pre- and postconditions. VCC is an annotation-based system, i.e., contracts and invariants are stored as annotations within the source code in a way that is transparent to the regular, non-verifying compiler.

Figure 1 shows the contract of a function that returns the smallest element of an array. The precondition “requires” parameter `len` to be greater than zero and pointer `a` to point to an array of size `len` in memory, accessible to the currently executing thread (indicated by “thread local”). The postcondition “ensures” that the result is indeed the minimal element of the array, i.e., (a) it is less or equal than any element in the array and (b) actually one of the elements of the array.

As most annotation-based verification systems today, VCC works using an internal two-stage process. The reason for this is a better separation of concerns and easy integration of different tools. As shown in Fig 2, the first stage of the VCC toolchain translates the annotated C code into first-order logic via an intermediate language called BoogiePL [15]. BoogiePL is a simple imperative language with embedded assertions. From this BoogiePL representation, it is easy to generate a set of first-order logic formulas, which state that the program satisfies the assertions. These formulas are called verification conditions and the stage a verification condition generator (VCG).

In the second stage, the resulting formulas are given to an automatic theorem prover (TP) resp. SMT solver (in our case Z3 [22]) together with a background theory capturing the semantics of C’s built-in operators,

```
int min(int *a, unsigned int len)
  _(requires len > 0)
  _(requires \thread_local_array(a, len))
  _(ensures \forall unsigned int i; i<len ==> \result <= a[i])
  _(ensures \exists unsigned int i; i<len && \result == a[i])
{ ... }
```

Figure 1: Example VCC function contract

etc. The prover checks whether the verification conditions are entailed by the background theory. Entailment implies that the original program is correct w.r.t. its specification. Interaction with the VCC tool is only possible (and necessary) before the first stage of the toolchain, by providing annotations. Once sufficient annotations have been provided (assuming the program fulfills its specification), the proof is done automatically, hence the term *auto-active* was coined for systems following this interaction paradigm.

Other tools following the annotation-based paradigm include Spec# [4] or Caduceus [17]. They are all based on powerful fully-automatic provers and decision procedures, and they support real-world programming languages such as C and C#.

Annotation-based Verification. Annotations can serve distinctly different purposes, though sometimes several different ones simultaneously. Besides requirement annotations that assure the behavior of the program towards its environment, auto-active based systems need auxiliary annotations to be able to prove the program correct w.r.t. the requirement specification.

While one class of auxiliary annotation is needed merely for efficiency reasons (e.g., lemmas or intermediate assertions), another class, the *essential* auxiliary annotations [8] are a prerequisite for the existence of a correctness proof (e.g., loop invariants or data-structure invariants and abstractions).

Many annotations are part of the definition of module boundaries (either implicitly or explicitly): pre- and postconditions modularize functions, loop invariants encapsulate functional behaviour of loops, and even assertions within a function body may serve as modularization points (separating the part of the function which has to establish the assertion from the part in which the assertion is assumed to hold). One important task in software verification is to find the right annotations at these module boundaries, especially at the function level in case of large software systems.

In theory, the strategy to come up with contracts for the functions of a large software system is simple: first, the top-most functions in the call graph are annotated with the weakest specification necessary to establish the requirement specification. Then, in a top-down manner, all functions are iteratively annotated with the weakest specification sufficient to establish correctness of their parents. Similarly, this process could also be performed bottom-up, starting with the strongest possible contract for the leaves of the call graph.

Unfortunately, using weakest resp. strongest contracts is not a good idea in practice. They are (a) hard to find and (b) hard to prove (strong contracts) resp. make it hard to prove the parent's correctness (weak contracts). That is, while choosing a strong postcondition for all children of f makes verification of f simple, it complicates verification of the children. So, in practice, the solution to a verification task is a set of auxiliary annotations that are just strong enough resp. just weak enough.

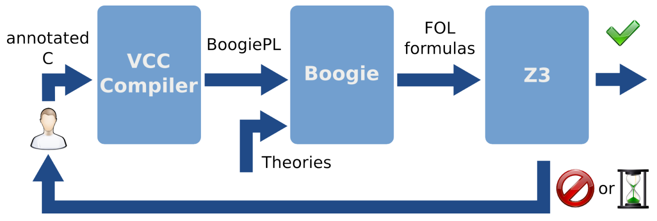

Figure 2: The VCC toolchain

```
int sqrt(int x)
_(requires x ≥ 0)
_(ensures
  \result^2 ≤ x ∧
  (\result+1)^2 > x)
```

```
int sqrt(int x)
_(requires x ≥ 0)
_(ensures
  ∀y; (y ≥ 0 ∧ y ≤ \result ⇒ y^2 ≤ x) ∧
  ∀z; (z ≥ 0 ∧ z > \result ⇒ z^2 > x))
```

Figure 3: Two alternative contracts for the `sqrt` function.

Moreover, the logical strength of a contract is not its only relevant property but its syntactic form is just as important: consider the two contracts for the function `sqrt` that computes (an integer approximation of) the square root of an integer shown in Fig. 3. While both contracts are logically equivalent, i.e., specify the same behaviour of `sqrt`, one of the two contracts may be much more useful than the other one, depending on the verification tool used and the properties that are needed in the verification of a caller of `sqrt`.

2.3 Verification Strategy in Verisoft XT

For the first verification targets within PikeOS, we chose functions that were neither dependent on hardware features nor complex features of the programming or specification language (in particular, concurrency was left out). Also, minimal dependencies w.r.t. other parts of PikeOS were of advantage – eligible functions were identified using the call-graph of PikeOS. This allowed for quick progress in verification but also reduced efforts in adapting specifications when changes in the specification language were made as VCC evolved.

Next, we extended verification to sequential execution of a simple system call that already included kernel functionality spanning all levels of the PikeOS microkernel, in particular PowerPC assembly [6]. For this verification task, a first version of an abstract model of PikeOS according to the informal descriptions of the PikeOS kernel (e.g., the kernel's reference manual) was introduced.

Then, using VCC's support for specification and verification of concurrent code, which was added in the course of Verisoft XT, we were able to specify and verify preemptable system calls. This most notably included handling of preemption locks, interruption of threads, and process switch. All sequentially executed parts of the system call between the preemption points are represented by a transition on the abstract kernel state. Coupling invariants between the abstract and concrete kernel state ensure correct functionality of the kernel implementation. The overall correctness of PikeOS is then expressed as a simulation theorem between the abstract model of the kernel with the user machines and their concrete counterparts [7].

3 Lessons Learned From Microkernel Verification

Although the following list of issues might indicate otherwise, our primary conclusion from the Verisoft XT project is that current auto-active verification systems are powerful enough to be successfully applied to microkernels, i.e., complex concurrent systems. While only some core parts of the microkernel could be verified within Verisoft XT in the given time, all relevant specification mechanisms could be established. This would allow to extend verification to the rest of PikeOS, given enough manpower.

Independently of the size and the type of the system to be verified, the verification task using annotation-based verification tools can be divided into the following three phases:

1. Formalization of given (informal) requirement specifications as program annotations.
2. Adding auxiliary annotations to describe the boundaries and interfaces of the different modules of the system (e.g., function contracts or loop invariants).
3. "Local" verification of single modules (functions) in isolation.

In practice, all three steps have to be performed repeatedly during several iterations of changing annotations until a fixed-point is reached, any bugs in the code or the requirement specification are fixed, and verification of the whole system succeeds.

There are several common ways to simplify the verification of large and complex systems and make it feasible in practice: (a) reducing the cost of specifying and verifying a single property of a single module (function), (b) modularisation, i.e., decomposing the verification task by verifying one module of the system at a time, and (c) abstracting from details of the system's implementation and behaviour.

All three of these concepts are addressed to a certain extent by current deductive verification tools: (a) Verification tools have made a leap forward in recent years, enabling users to verify individual functions once considered challenging with ease. (b) Annotation-based verification tools like VCC make use of decomposition of the verification task by verifying individual functions and threads in a modular fashion. Unfortunately, in practice, the 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. (c) Abstraction is possible using a separate specification state and abstract data types. But support for this is limited in the VCC tool and methodology.

In the following, we will illustrate why verification of a system like PikeOS still is a challenging task, despite all support by the verification tool and methodology. Although some of the discussed issues are more prominent for verification targets that have the characteristics of a microkernel, they are in no way exclusive for this type of software but occur to a certain extent with every large software system.

The remainder of this section is structured according to the three specification and verification phases listed above.

3.1 Formalizing Requirements

As already argued in Sect. 2.2, in theory it is not necessary to come up with top-level contracts adapted to the (often informal) requirement specification of the system, as the strongest possible contract must always suffice. However, in practice, the strongest contract not only makes verification more complicated, but it also obscures the intention behind the behaviour of the system. In addition, regardless of the strength of a contract, the user has to find the right kind of abstraction when formalizing informal requirements. For these reasons, formal top-level contracts must take the informal requirements and other system documentation into consideration.

Issue: Imprecise Informal Specifications. There exists rich official material in form of end-user documentation and requirement engineering documentation for PikeOS. These are however focused on the strict separation of platforms, architecture, and kernel (which is justified from a maintenance perspective) and is thus of limited use for finding annotations for functional verification. In addition, user-level documentation keeps some details left implicit – one example for this is the informal specification of a system call that changes the priority of a thread (confined to the user-defined value MCP, the "maximum controlled priority" of a task), 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.

However, this system call is preemptable and thus the term "current priority" is vague; also the postcondition of the system call only holds if no other system call was issued in between that changes the thread's priority.

Approaches to Resolving the Issue. To avoid the issues of extracting formal specifications from existing (informal) knowledge in the first place, the silver bullet is to develop the software system with verification in mind from the start. During the system development, at the same time all required formal specifications would then be already fixed.

However, if an already existing system is to be verified, other approaches have to be taken. What is needed in this case is tool support and methodologies to come up with the formal requirement specifications – similar to established requirements engineering processes.

For such a process, instead of end user documentation, existing system architecture and implementation level documents are essential, as well as involving domain experts (i.e., system architects and developers) –

especially to find the right auxiliary annotations (e.g., object invariants) that capture the key ingredients for the system's correctness.

In this regard, to simplify transition from the more informal existing system documentation, a first step would be a specification mechanism that allows to write down the intention of the programmer respectively system architect explicitly, although without the need of a complex formal specification. Preferably, these documents have to be of value in the software development process besides formal verification. One already existing form of such precise specification of the expected system behaviour are test cases.

Besides existing documentation, code inspection is a practicable way to get a first version of the auxiliary annotations needed for the verification of the requirement specification. However, this involves the risk of repeating mistakes in the specification that were already introduced in the implementation.

Issue: No Syntactic Distinction Between Different Kinds of Annotations. Besides the requirement specification, there are two kinds of auxiliary annotations needed for verification with VCC, as already stated in Sect. 2.2. However, in the case of VCC, there is no clearly visible distinction between auxiliary and requirement specification and between essential and non-essential annotations added for performance reasons.

We argue in [8] that it is extremely important for the user to have knowledge about which kind of annotations are essential for the verification system as without that knowledge they may continue to add the wrong annotations in case of a failed proof attempt.

Moreover, we claim that requirement and auxiliary annotations must be syntactically distinguished. That makes specifications clearer and easier to read and understand. In certification processes it is indispensable to have a very clear understanding of which annotations form the requirement specification that has been verified. Also, managing annotations in case of software evolution is more difficult when no clear distinction is given: while auxiliary annotations may be changed or removed at will, requirements have to stay unchanged. As a consequence, without this distinction, maintainability of the annotations suffers, e.g., by having to separate requirements from auxiliary annotations in such a case.

Approaches to Resolving the Issue. It is preferable to keep requirement and auxiliary annotations separate, e.g., requirement annotations in the header file and auxiliary annotations in the C source file. Where no such separation is possible, keywords (in the style of visibility modifiers `public` and `private`) should be used.

3.2 Adding Auxiliary Annotations

After the requirements have been formalized as annotations, a multitude of essential auxiliary annotations have to be added before even the first verification attempt can begin. Among these auxiliary annotations are function contracts, loop invariants, as well as data structure invariants. In most cases, this initial set of auxiliary annotations is not sufficient to prove the part of the system considered correct and the annotations have to be adapted in several iterations, for reasons further explained below.

3.2.1 Modularization

Modularization helps to reduce verification effort by decomposing the verification task. For the modular verification of sequential programs with VCC, a function is verified using the contracts of all the functions it calls – instead of their implementations. This design choice relieves the load on the automated prover at the expense of having to write (auxiliary) function contracts for all functions.

For this reason, using current auto-active verification tools, the bottleneck of verification is how to find the right auxiliary specifications and specifications of interfaces between modules. In addition, architectures such as microkernels feature some inherent characteristics that restrict the extent to which the specification task can be modularized.

In the case of the PikeOS microkernel, implementations of single C functions are deliberately kept simple to facilitate maintainability and certification measures – the functionality of the whole kernel is rather implemented by interaction of many of these small functions, operating on common data structures. Microkernels, and more generally, all operating systems, have to keep track of the overall system's state, resulting in relatively

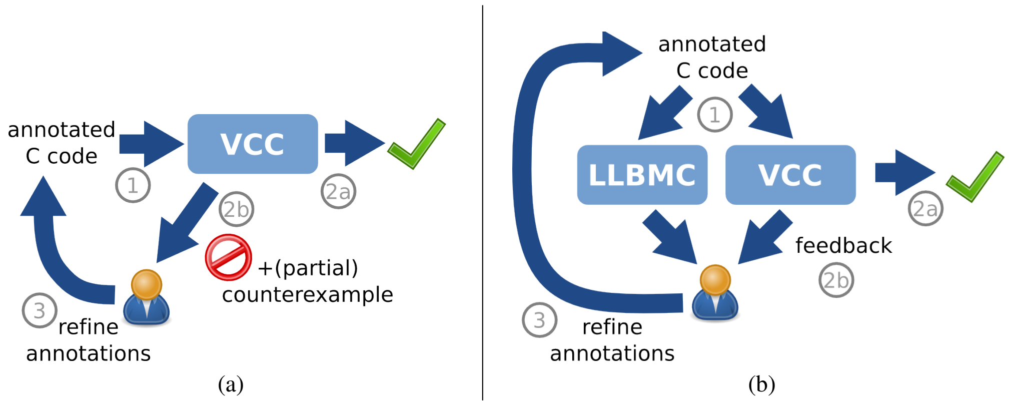

Figure 4: (a) Normal VCC workflow and (b) counterexample guided *manual annotation* refinement (CEGMAR).

large and complex data structures on which many of the kernels functions operate conjointly. This amount of interdependencies has an impact on the following issues.

Issue: Entangled Specifications. Function specifications have strong dependencies on each other. Finding the right annotations for a single function requires the verification engineer to consider several functions at once, due to these dependencies. Good feedback of the verification tool in case of a failed verification attempt is essential in this case. Feedback given by annotation-based verification tools so far only focuses on the function currently being verified. This allows the user to pinpoint and fix bugs in the specification or program respectively to change auxiliary annotations, e.g., loop invariants or contracts for called functions. For the verification of single functions, this tool support is sufficient. However, current specification methodologies are lacking w.r.t. providing help to the user in case of analyzing problems with interdependent specifications.

Approaches to Resolving the Issue. One solution to this issue, we considered in [10], is to provide early feedback when starting to specify a software system. Our approach is based on the combination of deductive verification as in VCC with software bounded model checking (SBMC) using the LLBMC tool [24]. For this, annotations written in VCC's specification language are translated into assertions that can be checked by LLBMC (i.e., boolean C expressions extended with some features specific to LLBMC). The SBMC procedure then allows to check these assertions without the need to provide any additional auxiliary annotations as in case with VCC, e.g., functions are inlined and loops unrolled instead of modularized using annotations. However, in contrast to deductive verification, the number of loop iterations resp. the function invocation depth is bounded. If no assertion is violated within the given bounds, this does not imply that the program adheres to its specification, as a bug may still occur, e.g., in a loop iteration outside the given bound. That is, LLBMC does not provide a full proof (that is left to VCC) but a quick check that can point the user to problems in the annotations early on and thus avoid unnecessary VCC proof attempts.

A comparison of the regular VCC workflow with the SBMC-supported variant is shown in Fig. 4. Using VCC alone (Fig. 4a), starting from the sufficiently annotated C code, VCC is invoked (step 1). If VCC verifies the program to fulfill its specification (step 2a), the process ends. More often, though, verification fails (step 2b) and the user has to change annotations using the counterexample provided by VCC (step 3).

Figure 4b shows the same process, this time with guidance by LLBMC: the annotated C code is given to both VCC and LLBMC in step 1. In case LLBMC reports a violated assertion (step 2b), the user is given a concrete trace through the program that leads to this violation, which is very helpful in finding either bugs in the implementation or specification.

Issue: Module Granularity. Dependencies between functions obfuscate module boundaries and thus make finding the right module interfaces difficult. In addition, even if larger modules can be identified, there is no particular support for a hierarchical modularization in annotation-based verification systems and for specifying properties of such a larger module, e.g., on the system architecture level.

Also, by the VCC methodology, module boundaries are fixed by having to provide function contracts for all functions – this implies that every function is also a module. However, some function boundaries of the software system might also be chosen for reasons other than clear separation of disjunct functionality of different modules. Ideally, for those functions of a large software system that are not part of the externally visible part of the system's interface, such function contracts could be omitted – especially for small library functions that occur at the leaves of the system's call graph.

Approaches to Resolving the Issue. Inlining of function calls would help to make function contracts obsolete for those functions of the system that are not a module on its own but only part of some larger functionality. This would already moderate the issue of imposing fixed module boundaries for functions in some cases.

3.2.2 Abstraction

Besides modularization, abstraction is another important instrument to handle verification of large software systems. Good abstraction of the functional 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 to verify the system at all – which abstraction is appropriate not only depends on the system properties to be specified but also on how well the verification tool used is able to reason about it.

Issue: Finding the Right Abstraction. To find the right abstraction for data structures, analyzing the source code alone is often not sufficient in practice. Complexity of the implementation of a single data structure is secondary when considering verification effort. Gathering the important properties of the data structure is crucial in order to find the right abstraction. Which of these properties are needed and relied upon in the software depends on its usage in the functions of the system. While some techniques such as CEGAR exist that may help in some cases in finding the right abstractions, these functions are not sufficiently supported in current deductive annotation-based systems.

Again, multiple dependencies between the functions that all operate on (parts of) the same data structure make it hard to find the right abstraction. Besides better support from verification tools, in order to come up with the right abstractions, information from system developers and architects is vital.

Issue: No Language Support for Abstractions. Abstraction in the VCC tool is mainly achieved by using specification ("ghost") state together with built-in abstract data types like maps or sets, where the abstract state is related to the concrete program state with the help of coupling invariants.

One advantage of having a similar notation for the ghost data structures and statements compared to real C is that programmers are already familiar with it. However, at times, such notations are still too close to the source code and better suited alternatives exist – in general, mechanisms to specify abstractions have to be more flexible in order to be able to choose the right abstraction methodology for the task at hand.

Approaches to Resolving the Issue. Concerning data abstraction, user-defined abstract data types were already introduced into the VCC methodology, 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.

3.3 Local Verification

Deductive software verification tools have improved in recent years to a degree that full functional verification of individual functions written in real-world programming languages is practicable with reasonable effort. This is demonstrated, e.g., by the results of recent verification competitions [20, 13], where selected software verification problems are solved with limited time resources by teams ranging from verification tool developers to regular tool users.

In the following, we consider for small scale verification the verification of a single function against a *given* informal requirement specification. This includes formalizing the requirement specification in a way that is suitable to the verification methodology at hand, as well as coming up with all auxiliary specifications necessary to verify the function correct w.r.t. its contract.

Issue: Support in Finding Auxiliary Annotations. Finding the right auxiliary annotations for local verification is a complicated task. One issue, finding the right essential annotations at module boundaries, has already been mentioned in Sect. 3.2 – besides the obvious function contracts, also loop invariants belong to this group. Even if, after some iterations, these essential annotations are appropriate for verifying the function in question, finding the right auxiliary annotations between modularization points remains to be done.

As functional program verification is undecidable, it is not surprising that coming up with the right annotations is non-trivial. Until now, there is little support by the tools in finding the right auxiliary annotations: in case of performance problems of the prover, the user can inspect the proof process and investigate which annotation takes up most of the time. In addition, statistics of the underlying SMT solver are reported, e.g., the number of quantifier instantiations so far, in order to find bottlenecks. In case of too weak essential annotations, VCC produces a counterexample that helps to find the missing or wrong annotations.

If none of these facilities provide the right clue, the user is left with "debugging" the verification state by inserting additional assertions to further split the proof and isolate those parts of annotations that are difficult for the automatic prover.

At some points the behavior of the verification system is unexpected and experience in using the verification system is crucial: for example, the assertion `\exists int i; i == 1` cannot be verified by VCC without providing another essential annotation in form of a *trigger*. Another stumbling block is that not only is the ordering of assertions of importance in finding a proof, but also is the ordering of loop invariant annotations. Also, for performance reasons, some information about the proof state is discarded when calling other functions and the user has to explicitly state which information to keep after the function call for the verification of the remaining function.

Issue: Amount of Annotations. Although small scale verification is possible using current verification tools and methodologies with reasonable effort, there is still room for improvements. Most notably, the large amount and high verbosity of annotations is one issue when using auto-active verification tools. In case of VCC, the specification language has been optimized several times to reduce annotation overhead.

Still, for example, for the functional verification of parts of the memory manager in PikeOS [5] the overhead of specification compared to the code compared by lines was 5:1, using VCC as of Feb. 11, 2011.

Approaches to Resolving the Issue. An obvious way to reduce the amount of annotations necessary, besides improving the performance of the SMT solver, is to identify common specification patterns and create new specification constructs as abbreviation. Another possibility is to choose defaults for specification constructs that cover the most frequent explicit specifications.

Note that in case of C, function declarations are given in separate header files and VCC takes advantage of this by allowing to annotate these declarations with the function contract. This separates the interface specification from the auxiliary specification in the function body but at the same time impairs usability of the verification tool: either the user does not have the contract and the implementation visible side by side when verifying or inspecting the function, or he/she duplicates resp. moves the contract to the implementation, incurring additional overhead to synchronize the header file with the implementation.

But even with the interface specification taken care of, the problem of a large amount of auxiliary annotations in the function body remains. Again, support of a special verification development environment could help the user to keep track of both implementation and specification. Parts of the annotations irrelevant for the user in understanding the specification could be hidden, given a heuristic that determines the relevancy of an annotation. Information that could be taken into account for such a heuristic is a similarity measure on annotations resp. formulas – this would allow to hide all but one annotation in a group of similar annotations.

Also, adapting the technique of code refactoring to specifications could be used to reduce the amount and improve readability and maintainability of annotations, e.g., by factoring out often used blocks of specifications in a specification function.

One option to reduce amount of required auxiliary annotations, is to shift user interaction towards the proof construction stage, as done in interactive provers. Annotations would only contain the main properties and insights about the implementation (e.g., loop invariants). Proof guidance (e.g., quantifier instantiations) would be done interactively (the information provided by the user can be stored in an explicit proof object for reuse, as in, e.g., the KeY verification tool [1]).

3.4 Handling Software Evolution

While the one-time effort in verifying a large software system is already high, this does not yet include expenditures for re-verification in case of software evolution. In order to integrate functional software verification into the software development process as part of the software quality control, costs for re-verification have to be reduced, e.g., by re-using as many annotations as possible.

Issue: Change Management. Another issue that arises when verifying an implementation in productive use is that it is constantly evolving, for example, to satisfy changing requirements from users of the software. To cope with these changes, when deciding on the frequency of applying code updates, one has to find a balance between costs for re-verification and the benefits of improvements in the system that might simplify specification and verification.

One possibility is to constantly apply the changes that occur in the production code to the source snapshot that is used as the verification target. While some proofs are not affected by such small changes, it is in general still necessary to adapt code annotations and verification proofs. The other possibility is to fix a version of the source code for verification. Then, to get verified properties for the code used in the actual product, the annotations of the verification target have to be adapted in one big leap to the current implementation at the end of the project.

4 Related Work

Two of the more recent large scale verification projects relevant for our work are the L4.verified project [21] and the Hyper-V sub-project within Verisoft XT. Compared to PikeOS, the Hyper-V source code is roughly on order of magnitude larger with about 100kLOC. Also, Hyper-V makes use of a multi-core hardware setup. Similar to the setup of PikeOS, the code base of the hypervisor was not adapted to simplify verification. Concerning the amount of annotations per line of source code needed, a maximum ratio of three to one is reported.

In another sub-project of Verisoft XT, the functional verification of a simple version of a hypervisor has been accomplished showing feasibility of auto-active verification for system software and which has been used to improve the VCC tool for further use in the Verisoft XT project. Specification effort mentioned in [3] matches the ratio reported for Hyper-V: the implementation consisted about 2.5k C code tokens compared to the 7.7k annotation tokens needed. Based on experience gathered within verification of those hypervisors, the VCC methodology and tool chain has been improved with the goal to increase performance of the automated prover and especially to guarantee fast response times in case of failed verification attempts, as this is the common case in software verification.

In the L4.verified project, an approach was taken that differs in various ways from the one used for the PikeOS and Hyper-V verification in Verisoft XT. Besides using the interactive theorem prover Isabelle/HOL for specification and verification of the seL4 microkernel, the implementation of the kernel was written from scratch for the project and adapted to suit verification, if needed. Also, an abstraction layer in form of a Haskell implementation of the kernel has been reported to be of great help (instead of directly tackling the problem of verifying the equivalent C implementation). Both implementations in Haskell and C together amount to approx. 15kLOC, while the Isabelle script used in verification spans 200kLOC [21].

Improving the Verification Process. To simplify the specification and verification process, being able to specify abstractions of code-level properties is important. There are many formalisms to provide control and data abstraction – of those, we already mentioned the Common Algebraic Specification Language (CASL), as well as Communicating Sequential Processes (CSP) or Abstract State Machines (ASM). Other approaches to improve usability of the specification languages take their cues from software engineering: for example specification patterns [11] or specification refactoring [16, 18] are useful instruments to facilitate writing and maintaining specifications.

A combination of interactive verification and auto-active tools has been used for the verification of checkers for complex algorithmic implementations in [2], taking advantage of VCC for code level verification and Isabelle/HOL to verify mathematical properties. Also, the Isabelle/HOL framework has been used in the HOL-Boogie tool [12], which has been developed to interactively perform complex proofs that VCC cannot handle.

5 Conclusion

Deductive program verification tools have made significant progress in recent years that allows to apply them to complex concurrent software systems. However, to be able to efficiently verify large software, further improvements are necessary. As the modules that can be verified using current tools get larger and more complicated, it becomes apparent that the process of finding the right specification is now the bottleneck.

In this paper, we have described issues in software verification that occurred during the verification of PikeOS and that contribute to the problem of coming up with sufficient code annotations. For most of these issues, we presented ideas on how to resolve the shortcomings.

In general, in order to handle verification of large software systems efficiently, we claim that better support from the verification tool is needed to get from verification of individual functions to verification of whole software systems. This would have to include support in finding the right modularization and abstraction. A first step in this direction is to provide the user with feedback in case of interdependent function specifications, so that mismatching contracts are discovered early in the specification process [10].

Besides annotation-based specifications, which are well suited for describing implementations at source-code level, there is a need for further specification constructs tailored to describing the system properties at higher levels of abstraction. Amongst others, these formalisms should allow to integrate knowledge of system developers even before starting the specification process at code level. At the code level, we propose to make use of established formalisms for abstract data types like CASL, to be able to specify common implementation data structures in a compact manner [9].

References

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager & Peter H. Schmitt (2005): *The KeY Tool*. *Software and System Modeling* 4, pp. 32–54.

[2] Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn & Christine Rizkallah (2011): *Verification of Certifying Computations*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *CAV*, LNCS 6806, Springer, pp. 67–82.

[3] Eyad Alkassar, Mark Hillebrand, Wolfgang Paul & Elena Petrova (2010): *Automated Verification of a Small Hypervisor*. In: *Verified Software: Theories, Tools, Experiments*, *LNCS* 6217, Springer, pp. 40–54.

[4] Mike Barnett, K. Rustan M. Leino & Wolfram Schulte (2005): *The Spec# Programming System: An Overview*. In: *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS), International Workshop, 2004, Marseille, France, Revised Selected Papers*, LNCS 3362, Springer, pp. 49–69.

[5] C. Baumann, H. Blasum, T. Bormer & S. Tverdyshev (2011): *Proving Memory Separation in a Microkernel by Code Level Verification*. In Wilfried Steiner & Roman Obermaisser, editors: *1st International Workshop on Architectures and Applications for Mixed-Criticality Systems (AMICS 2011)*, IEEE Computer Society, Newport Beach, CA, USA.

[6] Christoph Baumann, Bernhard Beckert, Holger Blasum & Thorsten Bormer (2009): *Formal Verification of a Microkernel Used in Dependable Software Systems*. In Bettina Buth, Gerd Rabe & Till Seyfarth, editors: *SAFECOMP'09*, LNCS 5775, Springer, pp. 187–200.

[7] Christoph Baumann, Bernhard Beckert, Holger Blasum & Thorsten Bormer (2010): *Ingredients of Operating System Correctness*. In: *Proceedings, embedded world 2010 Conference, Nuremberg, Germany*. Available at http://formal.iti.kit.edu/beckert/pub/ki-journal2009b.pdf.

[8] Bernhard Beckert, Thorsten Bormer & Vladimir Klebanov (2011): *Improving the Usability of Specification Languages and Methods for Annotation-based Verification*. In Bernhard Aichernig, Frank S. de Boer & Marcello Bonsangue, editors: *Formal Methods for Components and Objects, 9th International Symposium FMCO 2010. State-of-the-Art Survey*, *LNCS* 6957, Springer.

[9] Bernhard Beckert, Thorsten Bormer & Vladimir Klebanov (2012): *Improving the Usability of Specification Languages and Methods for Annotation-Based Verification*. In Bernhard Aichernig, Frank de Boer & Marcello Bonsangue, editors: *Formal Methods for Components and Objects*, *LNCS* 6957, pp. 61–79.

[10] Bernhard Beckert, Thorsten Bormer, Florian Merz & Carsten Sinz (2011): *Integration of Bounded Model Checking and Deductive Verification*. In: *FoVeOOS'11*, pp. 86–104.

[11] Domenico Bianculli, Carlo Ghezzi, Cesare Pautasso & Patrick Senti (2012): *Specification patterns from research to industry: a case study in service-based applications*. In: *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, IEEE Press, Piscataway, NJ, USA, pp. 968–976.

[12] Sascha Böhme, K. Rustan Leino & Burkhart Wolff (2008): *HOL-Boogie – An Interactive Prover for the Boogie Program-Verifier*. In: *Proc., 21st Int. Conf. on Theorem Proving in Higher Order Logics*, Springer, pp. 150–166.

[13] Thorsten Bormer, Marc Brockschmidt, Dino Distefano, Gidon Ernst, Jean-Christophe Filliâtre, Radu Grigore, Marieke Huisman, Vladimir Klebanov, Claude Marché, Rosemary Monahan, Wojciech Mostowski, Nadia Polikarpova, Christoph Scheben, Gerhard Schellhorn, Bogdan Tofan, Julian Tschannen & Mattias Ulbrich (2011): *The COST IC0701 Verification Competition 2011*. In: *FoVeOOS'11*, pp. 3–21.

[14] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte & Stephan Tobies (2009): *VCC: A Practical System for Verifying Concurrent C*. In: *Theorem Proving in Higher Order Logics (TPHOLs)*, *Lecture Notes in Computer Science* 5674, Springer, pp. 23–42.

[15] Rob DeLine & K. Rustan M. Leino (2005): *BoogiePL: A Typed Procedural Language for Checking Object-oriented Programs*. Technical Report MSR-TR-2005-70, Microsoft Research. Available at `ftp://ftp.research.microsoft.com/pub/tr/TR-2005-70.pdf`.

[16] Yishai A. Feldman, Maayan Goldstein & Shmuel S. Tyszberowicz (2007): *Refactoring with Contracts*. In Danny Dig, editor: *WRT*, pp. 13–14.

[17] Jean-Christophe Filliâtre & Claude Marché (2004): *Multi-prover Verification of C Programs*. In: *Formal Methods and Software Engineering*, LNCS 3308, Springer, pp. 15–29.

[18] Ian Hull (2010): *Automated Refactoring of Java Contracts*. Master's thesis, University College Dublin.

[19] Robert Kaiser & Stephan Wagner (2007): *Evolution of the PikeOS Microkernel*. In Ihor Kuz & Stefan M Petters, editors: *MIKES: 1st International Workshop on Microkernels for Embedded Systems*.

[20] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich & Benjamin Weiß (2011): *The 1st Verified Software Competition: Experience Report*. In Michael Butler & Wolfram Schulte, editors: *FM'11*, *LNCS* 6664, Springer.

[21] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch & Simon Winwood (2010): *seL4: Formal Verification of an Operating System Kernel*. *Communications of the ACM* 53(6), pp. 107–115.

[22] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *Proc., 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Budapest, Hungary*, LNCS 4963, Springer, pp. 337–340.

[23] RTCA SC-167 / EUROCAE WG-12 (1992): *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*. Radio Technical Commission for Aeronautics (RTCA), Inc., 1828 L St. NW., Suite 805, Washington, D.C. 20036.

[24] Carsten Sinz, Stephan Falke & Florian Merz (2010): *A Precise Memory Model for Low-Level Bounded Model Checking*. In: *SSV'10*.