Better Avionics Software Reliability by Code Verification*

A Glance at Code Verification Methodology in the Verisoft XT Project

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

[1] Saarland University, Dept. of Computer Science, Saarbrücken, Germany
[2] University of Koblenz, Dept. of Computer Science, Germany
[3] SYSGO AG, Klein-Winternheim, Germany

Abstract. Software reliability is a core requirement for safety- and security-critical systems. In the area of avionics, for example, the DO-178B standard requires extensive validation, such as software reviews, requirement engineering, coverage analysis, and careful design of test cases. In a broader context, EAL7 (of the Common Criteria framework) also demands "formally verified, designed, and tested" systems. It is part of the BMBF-supported Verisoft XT project (`www.verisoftxt.de`) to explore the freedom of design offered within these regulatory requirements, where code verification is one of the available options.
In recent years, deductive code verification has improved to a degree that makes it feasible for real-world programs. In the Verisoft XT subproject Avionics, the goal is to apply formal methods to a commercial embedded operating system. In particular, the goal is to use deductive techniques to verify functional correctness of the PikeOS microkernel. For verification, we use tools like VCC (the Verifying C Compiler) developed by Microsoft Research, which is a batch-mode verification tool, i.e., when all specifications and other required information have been added as annotations to the source code (which is the actual user effort required), the tool verifies the code automatically. First experiences with this new tool are described in this paper.

1 Introduction

The Verisoft XT Project. The core of the predecessor project Verisoft I (2003–2007) was to demonstrate the feasability of pervasive verification of a complete operating system stack, including user-level programs (such as a secure mail client [2, 3]). This system was purpose-built for the project (i.e., "academic"). It includes the VAMOS microkernel [12], the SOS operating system [7], and runs on the verified VAMP hardware [5, 6, 11, 27].

Now, the experience gained from this project shall be transferred to real-world applications. Therefore, the Verisoft XT project (2007–2010) aims at the verification of industry-use software with several ten thousand lines of code, deploying the methodology developed in Verisoft I. The success of this critical endeavour will emphasize the ongoing progress in the field of formal program verification and open up new perspectives concerning the security of commercial system software. More information on Verisoft XT is available at the web site `www.verisoftxt.de`.

PikeOS and the Verisoft XT Subproject Avionics. PikeOS (see `http://www.pikeos.com/` for further information) is a platform running on x86, PPC, ARM, for developing embedded systems where multiple guest operating systems and applications can run simultaneously in a secure environment (e.g., based on standards like POSIX and ARINC 653). The architecture is based on a design with a small, compact microkernel serving as a paravirtualizing hypervisor that provides a core set of services [18]. It can be augmented by the PikeOS system software (PSSW) and an integrated development environment. In particular, PikeOS offers

* Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07 008. The responsibility for this article lies with the authors.

partitions so that multiple guest operating systems can be virtualized onto the same CPU. If desired, a system integrator can set up appropriate communication channels between partitions, and also impose real-time requirements upon these (time partitioning).

PikeOS has already been developed to meet the DO-178B [24] standard: DO-178B is the leading norm for safety in the aviation industry. However, the DO-178B dates from 1992, when software code verification was yet considered beyond the practical state of the art. In the review process of DO-178 (intended to give rise to DO-178C) it has been discussed to provide a better integration for (e.g., deductive) formal methods – while keeping backwards compatibility with the previously very prominent place of testing.

The Avionics subproject agenda of Verisoft XT comprises the development of a Common Criteria [10] (a security standard that will be discussed below) security target draft, code verification, attestation services and their verification, development of high-level models and their correspondence proofs, as well as general research on the integration of Common Criteria and DO-178. Here we report on code verification.

Formal Code Verification. Formal methods are logic-based approaches to specifying and validating software (and hardware) systems, using a language with precise semantics, and usually including a technique for a mathematical verification of those properties.

Formal methods can be used in various stages of the software development process to improve the quality of software. In our case, in the validation step, we employ deductive program verification to ensure that the source code complies with its formal specification. The question how to come up with a specification that reflects the intended behavior of the program is an important issue and will be discussed in Sect. 4.

Unlike testing, formal software verification covers all possible inputs and every execution path of a program and, thus, is able to expose any error in the program w.r.t. a given specification.

In our approach, to check whether a program to be verified performs according to its specification, a logical formula is automatically generated from the source of the program and the specification. This formula, called *verification condition*, is rendered in predicate logic and has the property that, if it is valid, then the program is correct w.r.t. its specification. Finding a mathematical proof for the validity of this formula, which would serve as a witness for the correctness of the program, is then a task to be solved by a theorem proving system.

In the Verisoft XT subproject Avionics, we use the Verifying C Compiler (VCC) developed by Microsoft Research (see `http://research.microsoft.com/vcc`) to perform the transformation of the program together with its specification into a verification condition (in predicate logic), which is then checked for validity using the Z3 [13] automatic theorem prover.

2 Specifics of OS Verification

Non-Termination and Concurrency. A sorting algorithm has the specification that its input is a list (precondition) and that it returns a sorted list (postcondition), which is a permutation of the input list. The semantics of such a program can be described as relationship between its input and its output. (Note that the precondition in this case is non-trivial even if the input can be an arbitrary unsorted list, because lists have a certain structure in memory, they must not be cyclic etc.)

An operating system, on the other hand, consists of an infinite loop, so while there is a precondition (machine state after boot), it does not have a meaningful postcondition and its behavior is not manifested in an input/output relation. Nevertheless, operating systems and similar nonterminating systems (e.g., databases), which are also called *reactive systems* can be described

by their reaction to input from outside. Another way to describe their behavior is to formulate properties that hold throughout or at certain points of their infinite loop.

Moreover, an operating system also uses the protection mechanism of the underlying hardware (in particular, there usually exist at least two different levels such as a user mode and a privileged mode). The operating system kernel itself runs in privileged mode, user applications run in user mode.

To allow optimal usage of CPU time, an operating system allows several user applications to run at the same time (*concurrency*). Interrupts (e.g., from the timer) allow the operating system to do scheduling and to react to external signals. In real-time settings, to guarantee a quick reaction, even within the operating system kernel code itself there are many regions where execution may be intercepted (*preemption*). Because of that, atomicity of actions has to be ensured – for example, when one user application is interrupted while writing to an operating-system-kernel structure and then another user application is restarted that was reading from a kernel structure. Since there is no C-level semantics for interrupt handling, the interrupt handling part of the kernel is written in assembly.

Rich State, Kernel Boundary, and Invariants. The structure of the state of an operating system kernel is quite rich and complex, because the entire data structures of the kernel enter the state; moreover it includes the register state and the memory state.

For operating systems, interesting places to look at are kernel entry and exit points (because one assumes that while running in user space only very tightly controlled changes can be made, as the hardware does not run in privileged mode). It is a tradition (e.g. [8]) to call the predicates (expressible in first-order or higher-order logic) that hold at interesting locations “invariants”.

VAMOS (the microkernel from Verisoft I) had exactly one entry point and one exit point. For PikeOS the situation is more complex, because we have several kernel entry and exit points. (This is not a real necessity of design, though.)

Hence the main proof goal is to show that invariants at the kernel entry and exit points conform to a formal description of the system. In the next step, one can show that the formal description of the systems conforms to the natural language description. This step cannot be a formal proof, but a good formal description of a system can help to gain assurance for its informal natural language description.

3 Methodology and Toolchain

To prove that the implementation of the PikeOS microkernel is correct with respect to the specification (which in turn is derived from high-level properties), we employ the Verifying C Compiler (VCC) developed by Microsoft Research. This verification tool is also used in the Hypervisor subproject within Verisoft XT.

VCC is an automatic verification tool that features a specification language tailored to C. This allows the developer to write the specification in a way that is close to the syntax and semantics of the programming language (unlike modelling language such as the UML, for example, which uses a completely different representation of program elements and their properties in the specification). This concept of annotating source code with specifications in a manner that resembles the syntax of the programming language has already been successfully used (among others) for the Java Modeling Language (JML) [21] to specify Java programs and the SPEC# language (see `http://research.microsoft.com/specsharp`) to specify C# programs.

Efficient reasoning about data structures on the heap requires to abstract from the byte-oriented view of memory in C. For that purpose, the VCC language provides an abstract memory model (see [9]). This includes, in particular, the concept of *ownership*. Using the ownership concept, the developer can declare certain dependencies between data structures to state that properties of one data structure depend on the validity of the properties of other data structures.

Another feature of VCC, which is important for operating system verification, is the support for concurrency, which we however do not discuss in this paper.

The following subsections give a short introduction to the technical aspects of the toolchain, the VCC specification language, and an overview of the ownership-model of VCC.

The VCC Toolchain. Given a C program annotated with the specification as described in the previous section, the VCC tool performs three steps to conduct a correctness proof (if possible).

First, the annotated C code is compiled into an intermediate imperative programming language called BoogiePL [14]. This language offers only relatively simple programming constructs but includes specification constructs that are compiled from the specification annotations of the original C program together with the code itself.

The input for the following translation step consists of two parts: (a) the compiled BoogiePL code of the original C source and (b) axiomatic descriptions (in BoogiePL syntax) of certain aspects of the C programming language, called the BoogiePL prelude. The annotated BoogiePL program together with the prelude is then transformed into a formula in first-order predicate logic (called "verification condition"), which states that the program satisfies the annotated specification. The resulting verification condition is given to the automatic theorem prover Z3 to check whether the formula is valid, which implies that the original C program is correct w.r.t. the annotated specification.

The possible results Z3 may return are:

1. The formula is valid (Z3 has found a proof), i.e., it has been verified that the C program has the specified properties.
2. The formula is not valid (Z3 has found a counter-example). That can mean two things: (a) The program is not correct w.r.t. the specification, i.e., there is an error in either the program code or the annotations. (b) The program is correct, but some loop invariants or other annotations are not strong enough and, as a consequence, the generated verification condition is not a valid formula.
3. Z3 runs out of resources (time or space). This can mean three things: (1) The formula is valid and the program is correct (as in Case 1) but Z3 could not find a proof in the allotted time/space. (2) The formula is invalid (as in Case 2 above) but Z3 could not find a counter-example, and (a) the program is not correct or (b) some annotations are not strong enough.

In Case 1 above, the program verification was successful. In Cases 2 and 3, the developer has to analyse the problem. If he/she finds (using a possible counter-example) that the program does indeed not satisfy the specification, the error has to be corrected. If he/she finds that the program satisfies the annotations, then new annotations (stronger invariants, helpful lemmas, etc.) have to be added. This process is repeated until Z3 finds a proof.

One noteworthy feature of VCC is that proof search is fully automatic. Once the user has provided sufficient annotations (and provided that the proof obligation is valid), VCC constructs a proof without requiring any additional human interaction. Thus, user input is only needed between runs of the prover – it usually takes several iterations to add sufficient annotations to guide the prover. Interactive provers, in contrast, rely in certain cases on user input directly in the process of developing the proof.

The annotation language of VCC is guarded by C macros. When verifying, a flag is set so that these macros evaluate to functions specific to VCC, which in turn generates the corresponding BoogiePL code out of them. If a normal C compiler is used (without this flag), all annotations evaluate to the empty string, so that the annotations are transparent for the compilation process.

Specification Language. In this section, we give an overview of some parts of the specification language of VCC.

Note that the annotations added to a program do not actually change the program's behavior. They may impose restrictions and forbid the program to do certain things. If a programmer writes a program that violates such a restriction (by intention or by accident), then the program is not correct w.r.t. the annotation but it can still be executed. Moreover, well-chosen annotations do not impose useless restrictions, but – to the contrary – they express and make explicit those (implicit) restrictions that a program adheres to anyway.

Pre- and Postconditions. The specification of a function can be seen as a contract between the caller of the function and the function itself. This contract captures the effect the function has on the state of the program (including the value of all global variables and the heap).

In the sequential setting, the specification of a function relates the possible (input) states before the function is executed to the states after execution. The possible states before execution are described by the so called precondition of the function. The postcondition of a function describes the state after execution (depending on the input state).

The meaning of a pre-/postcondition pair attached to a function is that, if the function is started in a state in which the precondition is fulfilled and terminates, then the postcondition holds in the final state. This type of correctness that disregards termination is called *partial* correctness. One proves *total* correctness by showing, in addition, that the function terminates.

Note that, despite the nontermination-characteristics of operating systems (see Sect. 2), termination is an issue and has to be shown for the individual functions of the system. Only the main loop of the system is non-terminating.

In VCC, the pre- and postconditions are given using a syntax similar to C expressions. The specifications are annotated directly to the source code of the program, as shown in the example in Fig. 1. The precondition of a function is labelled with the keyword `requires`, and the postcondition is labelled with `ensures`. As can be seen in this example, the specification uses side-effect-free C expressions with some special language features from VCC, like the keyword `result`, which represents the return value of the function. Other VCC language features (not shown here) include, for example, quantifiers over (among others) integer datatypes.

```
int max(int a, int b)
  requires (a >= 0 && b >= 0)
  ensures  ((result == a || result == b) &&
            result >= a && result >= b)
{ if (a > b) return a;
  return b;
}
```

Fig. 1. Computing the maximum of two integers

Framing. If a function writes to globally accessible locations, either through a pointer passed to it or via global variables, it has to be annotated with so called `writes` clauses that list the

memory locations to which the function is going to write. This information, together with the function's pre- and postcondition, forms its contract.

Loop Invariants. The user has to annotate loops with so called loop invariants. The invariant may be temporarily broken in the loop body, but it has to hold before and after each iteration at the boundary of the loop.

The invariant of a loop has to fulfill several properties:

- it has to be true just before executing the loop,
- after each loop iteration, the invariant has to hold at the end of the loop body,
- the invariant has to convey enough information such that the prover is able to derive the postcondition of the loop from the invariant.

The last of the above conditions is important because the only information that the prover has about what value memory locations that have been changed by the loop have in the state after the loop terminates consist of the invariant and the fact that the loop condition does not hold anymore (otherwise, the loop would not have terminated).

Finding a proper loop invariant that complies with all three properties is a major task for the developer in the verification process and often requires several iterations.

Pointers. Access to memory locations via pointers requires in the VCC methodology that the type of the pointer is known, i.e., the prover can show a well-typedness property. This, for example, avoids errors due to accessing already freed memory regions or accesses past array boundaries.

Object Invariants and Ownership. One possibility to capture global properties of a software system in an abstract way is to define invariants for data structures (i.e., `struct`s in the case of C) used in the program. Like loop invariants, the properties stated in object invariants do not have to hold during all execution steps of the program but rather have to be proven to hold at certain points in the program. They can then be used as useful lemmas in a proof. With VCC, such invariants can be given by annotating a `struct` with (arbitrarily many) `invariant` clauses.

To enable modular reasoning about properties of complex data structures, e.g., pointer structures or nested `struct`s, and to capture relations between data structures, the concept of *ownership* between structured data is used (VCC's ownership model is an extension of the one used in the Spec# methodology [22]). The idea of ownership is that every `struct` has exactly one "owner" and can itself own arbitrarily many structures. The ownership relation is provided explicitly in annotations by the developer, and it reflects his/her abstract knowledge about the data structure and how it is used.

Another part of the framework for modular reasoning about invariants is that `struct`s can be in different states – they are either open or closed. Ignoring volatile variables, one property that VCC enforces in verification is that members of a closed struct cannot be modified by the program. In addition, if an object is closed, its invariant is guaranteed to hold.

Ghost Fields. Variables in the C code can be defined as being only usable in annotations (not in the actual code). These variables are called *ghost variables*. As they must not be accessed in normal code, the value of ghost variables cannot affect the execution of the program. Their use is to refer to them in specifications, providing, for example, an abstract view on the program state or "remembering" intermediate results of computations without interfering with the program.

4 Verification of the PikeOS Microkernel

Verification of an industrial microkernel leads to new issues that need to be addressed – beyond the mere scaling-up of verification technology to a larger and more complex system. Two of these issues are discussed in more detail in this section, namely (1) the problem of finding the underlying structure in an existing system, which then has to be made explicit in the annotations for the verification effort to succeed, and (2) handling hardware-dependent parts of the kernel, including assembly code.

Based on the work described below and using abstract specifications of the hardware-related functions, we were able to verify – as proof of concept – first system calls. Now, verifying all system calls is ongoing work.

Further ongoing work is modifying the source code to make it more amenable for verification with VCC, and keeping the verified code in sync with the current production code.

Structuring an Existing System. To be able to handle large (in terms of verification effort) projects, one has to structure the verification task into smaller modules. Discovering structures suitable for verification in an existing project, which was in our case not written with explicit provision for code verification with VCC, is a non-trivial task.

An obvious modularization given within the PikeOS system is the layering into (a) a platform- as well as architecture-related part, written in C and assembly language, and (b) a generic part on top of the hardware-related layer, written entirely in C.

For the generic kernel layer in PikeOS, one structure is given by the implementors of the kernel via translation units, i.e., the C source files. The dependencies between these modules were analyzed with a standard code documentation tool, namely doxygen, by generating a call graph starting from the main C entry point of the kernel. While the modularization given by the translation units alone is not suitable for structuring the verification task (because of the large number of dependencies between the modules), a useful classification of C functions could be derived from this call graph. The following function classes and related features of the VCC tool can be distinguished:

- functions depending on the platform- or architecture-specific implementation layer,
- functions manipulating pointers (e.g., the library for doubly linked lists)
- functions using pointer-manipulation functions (e.g., the memory allocator)
- functions that are neither pointer-structure- nor platform- nor architecture-dependant (e.g., getter- and setter-functions, manipulation functions for simple datastructures).

The class of functions most easily verified is the last one in the above list, namely auxiliary functions with no dependencies. These functions neither depend on specifications of other functions nor are they visible at the boundaries of the kernel (where they would depend on the kernel API specification). Additionally, these functions require only simple features of the verification tool (e.g., no pointer arithmetic, simple heap datastructures). These functions correspond to the leaves in the call graph mentioned above.

Modeling the Layers of PikeOS in VCC. The hardware-independent layer of the PikeOS kernel can be directly annotated and verified, as VCC includes a formal semantics of ANSI C. The hardware-near layer, however, contains portions of assembly code, both in separate assembler source files and inlined into the C code. These code pieces are of course architecture-dependent and VCC does not support the interpretation of assembly code. Therefore a formal model of the assembly language for a certain architecture must be provided. In the Verisoft XT

Avionics subproject, the Freescale MPC5200 board including the PowerPC G2_LE core was chosen as the target system.

In a first step, a concise mathematical model of the core components and of the instructions' effects on them was derived manually from the Freescale hardware manuals [15, 16]. This machine model also contains exact descriptions of the caches, the exception signaling and handling, and it imposes some synchronization conditions on the code needed for consistency. Portions of this model were then translated into C to build a formal hardware model based upon which one can argue about the effects of PowerPC assembly code on the system. In this implementation all the CPU components like general purpose registers, special purpose registers (e.g. for address translation) and parts of the memory system (e.g. translation look-aside buffers) are represented by appropriate C data structures.

Because the impact of regular C code on the hardware registers is to large extents depending on the compiler and there is no formal semantics available for real-world compilers like gcc, it is not feasible to reflect the effect of these code blocks on the hardware model. One must rather determine a subset of the original hardware configuration that is only altered by assembler code. This subset of the model can then be defined as a global C structure. Hardware components which are frequently updated also by C code (like general purpose registers or the program counter) cannot be kept up-to-date in the global hardware state. Hence they are treated as local variables of every C function which contains assembler parts.

For the handling of assembly code one has to distinguish two cases:

1. assembly code blocks contained in separate assembler source files (macro assembler),
2. assembler commands included in C functions using the `__asm__` environment (inline assembler).

Basically, one can define a C function for each instruction of the architecture, simulating the instruction's effect on the hardware model. For macro assembler it is then easy to replace each instruction by a call to the associated C function. In the Verisoft XT Hypervisor project it was shown for x86-64 assembler that this methodology can even be automatized using a parser specially designed for this purpose [23]. For inline assembler, an additional translation step is needed, because C context is mixed with assembler instructions and values are exchanged between registers and variables.

To avoid problems with the peculiarities of assembler-related compiler specifics and the ABI, one can extract each inline assembler portion into an auxiliary C function. Its parameters are the C variables that were transferred to the `__asm__` block. A representation of this function in macro assembler can then be retrieved by compiling the kernel and performing an object dump. This macro assembler code can be conveniently handled as described above. That is, one leaves the effort of interpreting inline assembler to the compiler. The alternative would be to formally define the translation of inline assembler. Of course, for both approaches, compiler correctness must be assumed.

Abstracting from the Code. Having established a method for handling assembly code in a C model, one can start verifying kernel functions which contain or call assembler functionality. Naturally, at the beginning, the aforementioned C functions, which simulate the instructions, must be annotated and equipped with pre- and postconditions thereby creating an interface to the next layers of code. There the information about the effect of single instructions can be applied to prove properties of sequences of assembler instructions. Following this modular principle it is possible to wrap up the semantics of the entire function containing the assembly code portions. Accordingly the hardware becomes virtually invisible to the developer at this point and one can go on verifying the functions on the generic kernel layer. However, one can

even abstract further from the code and build a more general and simplified model of the kernel. This is achieved by defining abstract data structures in the ghost state that are coupled with the corresponding structures in the real code by invariants. These abstractions result in a high-level model of the kernel using which one can prove properties for the security target.

5 The Common Criteria and Future Versions of DO-178

PikeOS has a strong certification background in avionics. But it has already been mentioned, that in non-avionics contexts, the Common Criteria are a norm of high interest (DO-178B requirements have been compared to Common Criteria EAL4/5 [1]). In the Common Criteria, at least for levels up to EAL4, there is mutual international recognition of national certifications. Conversely, there is still some room for the exact interpretation of the more demanding higher levels [28].

Strictly speaking, one could claim that code verification is beyond what the Common Criteria demand. Even EAL6/7-specific requirements close to code verification either do not require any formal argumentation below the function level (that is, functions are treated as black boxes such as in ADV_FSP.6) or where going below the function level (such as in ADV_IMP.2) they do not demand a formal proof. However, the proofs generated by code verification actually are an efficient tool for proving to an evaluator that the implementation honors the specification. In particular, code annotations can be kept within the code, thus ensuring that code and specification are kept in sync. This also may be helpful when different platforms have to be maintained simultaneously.

Moreover, strong deductive methods such as code verification are also encouraged in discussions of DO-178 [25] (the DO-178 places less emphasis on function-level properties and more emphasis on structural properties which fits well with code verification).

6 Related Work

We have already mentioned that the Avionics subproject of Verisoft XT builds upon previous work in Verisoft I. Related work in kernel verification was already done in the '70s and '80s in the projects UCLA Secure Unix [29] and KIT [4], and more recently at the Universities of Dresden and Nijmegen (VFiasco project) [17] and in the EROS/Coyotos project [26]. A current project in kernel verification is L4.verified at NICTA (Australia) [20]. An overview and comparison of these and other related projects is given in [19].

Acknowledgments. We are very grateful to Matthias Daum (Saarland Univ.) for his help and many fruitful discussions, and to Markus Wagner (Univ. of Koblenz) for his work in Verisoft XT Avionics. We also thank Marc Bommert, Knut Degen, Oliver Kühlert, Alexander Züpke (SYSGO AG), Ross Hannan (EUROCAE WG-71), Mark Hillebrand, Bruno Langenstein, Dirk Leinenbach, Andreas Nonnengart (DFKI GmbH), Christian Stüble (Sirrix AG), Colin White (ESG GmbH), and the VCC research team at Microsoft Research (EMIC), in particular Markus Dahlweid, Michał Moskal, Thomas Santen and Stephan Tobies.

References

1. J. Alves-Foss, B. Rinker, and C. Taylor. Towards Common Criteria certification for DO-178B compliant airborne software systems. Technical report, Center for Secure and Dependable Systems, University of Idaho, 2002. Available at `http://www.csds.uidaho.edu/papers/`.
2. G. Beuster. *A Methodology for Secure Interactive Systems*. PhD thesis, University of Koblenz, 2008.

3. G. Beuster, N. Henrich, and M. Wagner. Real world verification: Experiences from the verisoft email client. In *Proc., Workshop on Empirical Succesfully Computerized Reasoning (ESCoR)*, 2006.
4. W. R. Bevier. KIT: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
5. S. Beyer. *Putting It All Together: Formal Verication of the VAMP*. PhD thesis, Saarland Univ., 2005.
6. S. Beyer, C. Jacobi, D. Koning, D. C. Leinenbach, and W. J. Paul. Putting it all together: Formal verication of the VAMP. *International Journal on Software Tools for Technology Transfer*, 2005.
7. S. Bogan. *Formal Specification of a Simple Operating System*. PhD thesis, Saarland University, Computer Science Department, Aug. 2008.
8. P. Brinch Hansen. Concurrent programming concepts. *ACM Comput. Surv.*, 5(4):223–245, 1973.
9. E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A precise yet efficient memory model for C. `http://research.microsoft.com/apps/pubs/default.aspx?id=77174`, 2008.
10. Common Criteria Maintenance Board. *Common Criteria for Information Technology Security Evaluation. Version 3.1, Revision 1*, volume 1–3. CCRA, Sept. 2006.
11. I. Dalinger, M. Hillebrand, and W. Paul. On the verification of memory management mechanisms. In D. Borrione and W. Paul, editors, *Proceedings, 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, LNCS 3725. Springer, 2005.
12. M. Daum, J. Dörrenbächer, and S. Bogan. Model stack for the pervasive verification of a microkernel-based operating system. In B. Beckert and G. Klein, editors, *5th International Verification Workshop (VERIFY'08)*, volume 372 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.
13. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of the 14th International Conference, Budapest, Hungary*, LNCS 4963, pages 337–340. Springer, 2008.
14. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
15. Freescale Semiconductor. *G2 PowerPC^{TM} Core Reference Manual*, 1st edition, June 2003.
16. Freescale Semiconductor. *Programming Environments Manual for 32-Bit Implementations of the PowerPC^{TM} Architecture*, 3rd edition, September 2005.
17. M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel: The VFiasco project. In *Proceedings, 10th ACMM SIGOPS European Workshop*. ACM, 2002.
18. R. Kaiser and S. Wagner. Evolution of the PikeOS microkernel. In I. Kuz and S. M. Petters, editors, *MIKES: 1st International Workshop on Microkernels for Embedded Systems*, 2007.
19. G. Klein. Operating system verification: An overview. Technical Report NRL-955, NICTA, Sydney, Australia, June 2008.
20. G. Klein, M. Norrish, K. Elphinstone, and G. Heiser. Verifying a high-performance micro-kernel. In *Proceedings, 7th Annual High-Confidence Software and Systems Conf., Baltimore, USA*, 2007.
21. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual, Draft revision 1.193*, May 2006.
22. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *Proceedings, Object-Oriented Programming, 18th European Conference, Oslo, Norway*, LNCS 3086. Springer, 2004.
23. S. Maus, M. Moskal, and W. Schulte. Vx86: x86 assembler simulated in C powered by automated theorem proving. In *Proceedings, 12th International Conference on Algebraic Methodology and Software Technology (AMAST), Urbana, USA*, LNCS 5140. Springer, 2008.
24. RTCA SC-167 / EUROCAE WG-12. *Software Considerations in Airborne Systems and Equipment Certification*. Radio Technical Commission for Aeronautics (RTCA), Inc., 1140 Connecticut Avenue, N. W., Suite 1020, Washington, D.C. 20036, Dec. 1992.
25. RTCA SC-205/EUROCAE WG-71. Discussion and development site for Software Considerations in Airborne Systems, 2009. At `http://forum.pr.erau.edu/SCAS/`.
26. J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *Proceedings, IEEE Symposium on Security and Privacy*, pages 166–176. IEEE Computer Society, 2000.
27. S. Tverdyshev and A. Shadrin. Formal verification of gate-level computer systems. In K. Y. Rozier, editor, *LFM 2008: Sixth NASA Langley Formal Methods Workshop*, NASA Scientific and Technical Information (STI), pages 56–58. NASA, 2008.
28. M. Volkamer and H. Hauff. Zum Nutzen hoher Zertifizierungsstufen nach den Common Criteria. *IT Sicherheit & Datenschutz*, pages 692–695, 766–768, 2007.
29. B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verifification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.