TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

Technische Berichte
Technical Reports

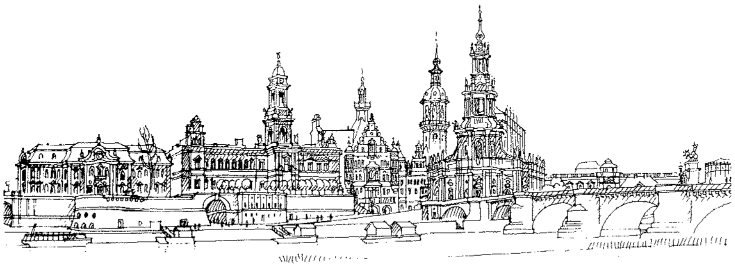
ISSN 1430-211X

TUD–FI01–1 – Januar 2001

Hendrik Tews, Hermann Härtig und Michael Hohmuth

Institut für Theoretische Informatik

VFiasco — Towards a Provably Correct μ–Kernel



Technische Universität Dresden
Fakultät Informatik
D-01062 Dresden
Germany

URL: http://www.inf.tu-dresden.de/

VFiasco — Towards a Provably Correct μ–Kernel

Hendrik Tews
Inst. for Theoretical Computer Science
Dresden University of Technology
tews@tcs.inf.tu–dresden.de

Hermann Härtig Michael Hohmuth
Inst. for System Architecture
Dresden University of Technology
{haertig,hohmuth}@os.inf.tu–dresden.de

Abstract

In this note we suggest to combine the recent developments in two different areas of computer science. First, μ–kernels of the second generation make it possible to structure operating systems into several parts with smaller interfaces. These interfaces can be enforced through the separation of address spaces. Second, powerful tools for theorem proving and new approaches to software verification make it possible to analyze software of considerable size. Through the combination of these two points it seems possible to apply formal method to operating system software. This note presents the VFiasco project that aims at the mechanical verification of security-relevant properties of the Fiasco μ-kernel developed at Dresden University. The project is based on a successfully completed case study.

1 Introduction

For most computer applications absence of errors is desirable, but not strictly necessary. However, there exist application domains where even a single malfunction cannot be tolerated. In the future cryptographic applications will run on general purpose PC's to sign documents, safely transfer money and so on. The correctness of these applications in an environment where possibly malicious attackers are present will be an requirement or at least a strong selling point.

An application can only provide guarantees about its correctness if it can rely on assurances of similar strength of the underlying operating system. For instance for the validity of an electronic signature the operating system must make sure that during every run of the signing program nobody can access the main memory to spy out the secret key that is used for signing.

Nowadays the correctness of operating systems is ensured via careful design and sophisticated testing. However, tests can only reveal errors. The absence of errors can only be proved through the use of formal methods, that is through the investigation of the program and its semantics with mathematical means.

Formal methods are rarely used in operating system programming because they are very labor intensive and existing approaches for software verification do not scale to the size and complexity of todays monolithic operating systems. Therefore, it is unrealistic to expect that traditional monolithic operating systems can provide the security guarantees that are needed in the future.

A possible solution is to partition the operating system into a μ–kernel and a monolithic system (e.g. Linux) that runs on top of the μ–kernel. Security critical applications run directly on the μ–kernel and so the size of the trusted computing base is drastically reduced. Other applications can use the Linux interface. It has been shown that running Linux on top of the μ–kernel L4 involves almost no performance penalty [2]. Fiasco [3] is an implementation of the L4 interface for the x86 architecture in a high–level programming language (namely C++). This fact and the small size of Fiasco (approximately 15.000 lines of code) make Fiasco amenable to formal verification techniques.

In this note we present the VFiasco (for Verified Fiasco) project that aims at the construction of a

This research was supported in part by the Deutsche Forschungsgemeinschaft (DFG) through the Sonderforschungsbereich 358 and by IBM through UPP– and SUR grants.

μ–kernel that provably fulfills basic security criteria. The next section shows recent achievements in formal verification techniques. Section 3 discusses the specific problems in operating system verification. Section 4 outlines the VFIASCO project and Section 5 presents the results of a case study that has been completed in preparation of the project. Section 6 discusses open problems.

2 Verification of Real Software

The use of formal methods, that is the investigation of software with mathematical means, dates back to the beginning of programming. However, software verification is rarely used in practice. There are various reasons for this. First, software verification is very labor intensive (much more than software construction) and thus very costly. It can only be attempted if highly qualified staff is available. Second, the quality requirements can often be met without the use of formal methods. Third, research on software verification often starts from idealized programming languages and uses idealized toy examples. For instance, Hoare logic assumes the absence of abnormal termination (exceptions)[1]. So the original Hoare logic cannot directly be applied to mainstream programming languages. As a last point, software verification involves a great amount of detail: It requires efficient tools that have not been available in the past.

Through continuing work in theoretical computer science, progress has been made such that it seems now possible to apply formal methods to mid-size projects. In this section I review some of these achievements.

The biggest development has been made in the area of theorem proving. A theorem prover is a software tool that implements a logic together with a deduction system. The user can express theorems in the logic and try to prove them with the deduction system.

Theorem provers are mainly divided into automatic and interactive ones. Automatic theorem provers use a brute force method and search the proof space up to a certain depth. Interactive theorem provers require the user to type in proof commands. The interactive theorem prover applies the proof command to the current proof state and computes the resulting proof state. The power of proof commands varies from simple ones to powerful ones that automatically apply induction and search for instantiation of quantifiers.

Even for small programs correctness proofs tend to be large and consist of a lot of simple computations. Therefore only toy examples can be proved without tool support. A good theorem prover relieves the user of the boring tasks of the verification, like simplification of arithmetic expressions, checking that all side conditions are fulfilled, keeping track of dependencies between theorems and so on. It provides visualization tools that guide the user in complex proofs and a database that keeps track of the proof state of all theorems.

Theorem provers vary also in the kind of logic they support. A popular logic is higher-order logic that allows to quantify over functions and arbitrary subsets. Higher-order logic always contains a pure functional programming language[2], so one can program inside the theorem prover. The main approach to software verification is now obvious: One translates the software into (its semantics in) the programming language of the theorem prover, formulates properties of the software (like freeness of deadlocks) in the logic of the theorem prover and finally proves them (hopefully).

A popular theorem prover for higher-order logic is PVS [7] developed at SRI. PVS is comparatively easy to use, its prominent features are a graphical representation of proof trees and powerful decision procedures that, for instance, can solve systems of linear inequations without user interactions. PVS has been used for hardware verification [6]. Similar to programming libraries there exist libraries of theorems for PVS, for instance a library that formalizes the datatype of bit vectors.

A second point of progress in theoretical computer science is the field of coalgebraic specification [10]. A coalgebra is a function with a structured codomain like

$$c \quad : \quad \mathsf{Self} \times A \longrightarrow (\mathsf{Self} \times B) \uplus \{\bot\}$$

Here Self is the state space of the coalgebra and $\uplus$ is the disjoint union. If we apply c to a state and an

[1] See [4] for a Hoare logic adopted to Java that treats abnormal termination.

[2] A functional programming language is pure, if it does not allow side effects.

argument of type A then c either fails with result $\perp$ or produces a successor state and an output of type B. So the c above models partial automata with input A and output B. Coalgebras come equipped with the notion of bisimulation (equality up to observable behavior) and the notion of invariance. Coalgebraic specification is the use of a special logic to describe properties of coalgebras.

There are three important things to note about coalgebras. First, coalgebras naturally support partial functions and it is thus very easy to model programs that fail or terminate abnormally on some inputs. Second, coalgebras fit nicely together with the object-oriented approach of software construction [9]. A finite set of coalgebras (on the same state space) can be regarded as a class and an element $x \in \mathsf{Self}$ as a snapshot of an object. Bisimulation then corresponds to observable equality of objects. Third, with coalgebras one can model possibly infinitely running systems, like operating systems.

The use of coalgebras is an advantage (in contrast for instance to algebraic methods or process calculi) because they greatly simplify the necessary mathematical framework, thus making it much easier to work with. The combination of modern theorem provers and coalgebras within the LOOP project[3] has lead to a verification environment for Java [5]. Also within the LOOP project the specification language CCSL[4] and a compiler that translates CCSL specifications into higher-order logic have been developed.

3 Specifics of Operating Systems

Software verification is easiest if the software is written in a pure functional programming language, without exceptions, having a simple global control structure (no parallelism), and using only well understood data types like natural numbers or lists. Anything which diverges from that makes the verification more complicated. Operating system software conceals a number of complications: It has to deal with parallelism, it uses low level data structures like bit vectors and it is usually written in an imperative programming language. Each of these complications have been treated alone before. The challenge in verifying an operating system lies in the combination of all these complications. (Note that operating systems do not contain any floating point arithmetic, which would make things even more difficult.) In the remainder of this section I discuss these complications and possible solutions.

Parallelism. Parallelism describes the fact that more than one action occurs at a given time. The difficulty with parallelism lays in the fact that two actions that occur in parallel can interfere with each other such that the overall effect cannot be described as a single combination of the two interfering actions. If every action is a composition of a number of atomic primitive actions (which is the case on a real computer system) it is possible to reduce parallelism to nondeterminism. In this approach one considers all possible serializations of the parallel actions. In order to prove that the parallel execution has a given property one has to prove the property for every possible serialization. Note, that there can exist infinitely many serializations. This is not a problem, because one can also reason over infinitely many serializations at once.

Low-level Data Structures. Operating systems have to use low level data structures such as bits and bit vectors to communicate with the hardware. Further they often explore the fact that integers are represented as bit vectors and that integers are addresses in the memory. In a theorem proving environment these three types (natural numbers, bit vectors, and addresses) are usually not the same. This involves two complications: First one has to deal with data structures that are much more complicated to describe than natural numbers (in the theorem prover PVS a bit vector is a function from a finite, down closed subset of the natural numbers to the booleans). Second one has to insert conversion operations in the theorem prover that compute the bit vector from a natural number and vice versa.

Imperative Programming. In the imperative (and the object-oriented) paradigm any operation is implicitly applied to the state of the system (or the current object) and modifies this state in

[3] Logic for Object-Oriented Programming, see URL `http://www.cs.kun.nl/~bart/LOOP/`.

[4] CCSL stands for Coalgebraic Class Specification Language.

place. One can model an imperative statement in a pure functional environment by a transition function that transforms a state of the system into its successor state. The development of the system under an imperative program is then mirrored by a sequence of states.

4 The VFiasco Project

VFiasco stands for verified Fiasco. Fiasco is an implementation of the L4 interface in C++ that has been developed within the DROPS project [1] at the chair of operating systems in Dresden. Coalgebraic specification [10] is developed at the chair of theoretical computer science. The VFiasco project is a cooperation of both chairs, it is about to start in the next months.[5] The primary goal of the project is to prove some security relevant properties of a reimplementation of Fiasco (hereafter called the VFiasco μ–kernel). The secondary goal is to further develop coalgebraic specification techniques and to apply coalgebraic specification to real software.

The project rests on a successfully completed case study (described in the next section). The planned activities in the project result from an analysis of the case study, the problems it has revealed, and its weak points. The road map of the project is as follows.

1. Identify security relevant properties that are needed by cryptographic applications running directly on the μ–kernel. An obvious example of such a property is protection against unauthorized access of main memory.

2. Identify a subset of C++ (called rC++, restricted C++ in the following) that suffices for operating system programming and for which it is easy to develop a semantics. (So rC++ will not contain a goto statement.) In parallel, extra operations that are needed for operating system programming (like setting the page-directory register for x86 CPU's) are added to rC++. Programs in rC++ will be compiled with a standard C++ compiler (using additional header files and a library).

3. For rC++, a formal semantics and a compiler that translates rC++ programs into their semantics in higher-order logic are developed.

4. Develop a coalgebraic specification for the L4 interface in the specification language CCSL. This specification has to entail the properties of Item 1.

5. The μ–kernel Fiasco is rewritten in rC++. The internal interfaces of this new μ–kernel VFiasco will be much stricter than those of Fiasco.

6. The sources of VFiasco are translated into higher-order logic. With the help of a theorem prover it is verified that the translated sources form a model of the specification of Item 4 and thus fulfill the security properties of Item 1.

[5] We acknowledge the approval of the DFG.

5 Showing that the Approach Works — Results of a Case Study

To see if the approach described above works in practice, the first author verified the interface of one class of Fiasco in a case study. The whole case study, including all sources, is available for download at `http://wwwtcs.inf.tu-dresden.de/~tews/vfiasco/`. All relevant details (including a short introduction into the specification language CCSL) are described in the technical report [11] (which is also available at the given URL).

As one of its services Fiasco provides memory management. Internally the class `space_t` represents one page directory. This class contains methods for insertion (`v_insert`) and deletion (`v_delete`) of virtual address space mappings and a method to make a page directory the CPU's current page directory (`switchin_context`). The case study concentrated on a few properties of these methods. The specification states that these methods should terminate normally (there should neither occur any page faults nor invalid assertions). Additionally, the insertion of superpage mappings (pages with a size of 4 MB) should be correct.

The whole specification consists of about 200 lines of CCSL code, divided into three specifications. The first specification describes a plain physical memory with read and write operations (without caching). The second specification describes virtual memory, built on top of physical memory. The read and write operations of the virtual memory fail if the virtual address is not mapped. The third specification describes the class `space_t` that runs in a virtual memory. The specification was quite easy to develop, it took only 4 weeks.

As next the C++ source code of the class `space_t` was translated by hand into the logic of the theorem prover PVS. The main verification task was to show (in the logic of PVS) that the translated methods form a model of the `space_t` specification, that is, that the methods of class `space_t` fulfill the properties that are required in the `space_t` specification. To carry out this proof it was necessary to formalize additional properties and lemmas in the theorem prover PVS. The completed case study consists of about 250 lemmas in 1000 additional lines of PVS source code. All proofs consists of a total of about 5000 proof commands. The verification took three months.

The verification revealed hidden assumptions in the method `v_insert`. Because these assumptions are not part of the specification, the method `v_insert` does *not* fulfill its specification. The problem is that for some possible states of the page directory and certain arguments an assertion in the method `v_insert` fails. In FIASCO the callers of `v_insert` always check for this condition.

The case study was a success because it showed that coalgebraic specification can indeed be applied to operating system software. The case study used the tools that were available at that time, they did not contain any support for imperative programming. One can expect that with such support (which can quite easily be added) one can reduce the complexity of the verification task.

6 Open Problems

The most critical point is the change of sources and module interfaces during software development. Of course, after such a change all theorems must be proved again. Theorem provers store proofs and allow the user to rerun stored proofs. The problem is that even smallest changes can invalidate a stored proof and require user interaction. In the worst case this results in an amount of work that is proportional to the number of proved theorems (in contrast to the relative size of the changes in the sources). In the VFIASCO project it is planned to carefully structure the μ–kernel into modules and to prove properties of the module interfaces. The proofs of these properties refer only to the source code of the own module and to proven theorems of other modules. This way the proofs become independent of changes in the source code of other modules (as long as their interface remains stable). It not clear if this approach will work out, and even if it works out, then the cost of every change in the sources will be magnitudes higher than usual. (To recompile FIASCO takes about a minute, to automatically rerun all the proofs of the case study takes half a hour.) Therefore, an important point for the success of the project will be to change the software construction process to minimize the changes after the first verification attempt.

A second problem is the quality of the existing theorem provers. The case study of Section 5 has been carried out with the theorem prover PVS. Up to now, the development of the case study and its migration to newer versions of PVS has lead to about 20 bug reports. There is an alternative to PVS, namely the theorem prover ISABELLE [8], but there is almost no experience with ISABELLE in Dresden and ISABELLE lacks some of the nice features of PVS.

The approach to deal with parallelism described in Section 3 has not been tested. However, if reasoning about parallel execution path in the μ–kernel poses to much problems it is always possible to reduce the parallelism, for instance through the use of global locks.

If the VFIASCO project succeeds then we know that using a given theorem prover one can prove some properties about the rC++ sources of VFIASCO. However, this does not prove anything about a compiled VFIASCO image. The compiler that is used to compile VFIASCO or the processor on which the image runs might have bugs that invalidate properties of the sources. The theorem prover might have a soundness problem that allows to prove false theorems. The semantics of rC++ might be incorrect and so on. Ideally one would have to use verified hardware and a verified compiler, where both things were verified with a verified theorem prover, and where all these verification took place on verified hardware ...

The aim of the VFIASCO project is to increase the trust into the correctness of the μ–kernel. The weakest point of an operating system are its sources. Problems with the compiler, the hardware, or the theorem prover are theoretically possible, but do not play any role.

Ideally one would like to prove a theorem like "There are no hidden channels on a system running FIASCO". But it is unclear how to formalize or even prove such a property. The project will therefore concentrate on simpler, more concrete properties. For instance: "No unauthorized access to memory pages", or "No user thread can achieve kernel privileges" and so on.

7 Conclusion

Special applications (for instance the generation of electronic signatures) place extraordinary requirements on the correctness of operating system services. They require provably correct behavior or (equivalently) guarantees about the absence of a certain class of errors. This level of correctness cannot be achieved through testing, it can only be achieved by the application of formal methods, for instance the verification of the source code of the operating system.

With the combination of modern specification techniques and modern theorem provers it seems possible to verify some properties of a complete μ–kernel. This note presents the VFIASCO project that aims at the verification of security relevant properties of the FIASCO μ–kernel, a L4 implementation in C++. A nontrivial case study has been completed; it proved that the approach described in this note is feasible.

References

[1] R. Baumgartl, M. Borriss, H. Härtig, C.-J. Hamann, M. Hohmuth, L. Reuther, S. Schönberg, and J. Wolter. Dresden Realtime Operating System. In *Proceedings of the First Workshop on System Design Automation (SDA '98*, pages 205–212, Dresden, March 1998.

[2] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-Kernel-based systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 66–77, New York, October 5–8 1997. ACM Press.

[3] M. Hohmuth. The Fiasco kernel: Requirements definition. Technical Report TUD–FI–12, TU Dresden, December 1998. Available at URL: http://os.inf.tu-dresden.de/fiasco/doc.html.

[4] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS, pages 284–303. Springer, Berlin, 2000.

[5] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object–Oriented Programming, Systems, Languages and Applications*, pages 329–340. ACM Press, 1998.

[6] M.K. Srivas and S.P. Miller. Formal verification of the AAMP5 microprocessor. In M. G. Hinchey and J. P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, 1995.

[7] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, number 1102 in LNCS, pages 411–414. Springer, Berlin, 1996.

[8] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in LNCS. Springer, Berlin, 1994.

[9] H. Reichel. An approach to object semantics based on terminal co–algebras. *Mathematical Structure in Computer Science*, 5:129–152, 1995.

[10] J. Rothe, H. Tews, and B. Jacobs. The coalgebraic class specification language CCSL. To appear in the Journal of Universal Computer Science, 2001.

[11] H. Tews. A case study in coalgebraic specification: Memory management in the FIASCO microkernel. Technical Report TPG2/1/2000, SFB 358, April 2000. Available at URL http://wwwtcs.inf.tu-dresden.de/~tews/vfiasco/.