Formal Verification of a Microkernel Used in Dependable Software Systems*

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. In recent years, deductive program verification has improved to a degree that makes it feasible for real-world programs. Following this observation, the main goal of the Verisoft XT project is (a) the creation of methods and tools which allow for the pervasive formal verification of integrated computer systems, and (b) the prototypical realization of four concrete, industrial application tasks.

In this paper, we report on the Verisoft XT subproject Avionics, where formal verification is applied to a commercial embedded operating system. The goal is to use deductive techniques to verify functional correctness of the PikeOS system, which is a microkernel-based partitioning hypervisor.

We present our approach to verifying the microkernel's system calls, using a system call for changing the priority of threads as an example. In particular, (a) we give an overview of the tool chain and the verification methodology, (b) we explain the hardware model and how assembly semantics is specified so that functions whose implementation contain assembly can be verified, and (c) we describe the verification of the system call itself.

We also explain why this effort matters in regulatory dependability frameworks such as DO-178B and IEC61508 for safety resp. Common Criteria for security.

1 Introduction

Background. In recent years, deductive program verification has improved to a degree that makes it feasible for real-world programs. Following this observation, the main goal of the BMBF-supported Verisoft XT project is (a) the creation of methods and tools which allow the pervasive formal verification of integrated computer systems, and (b) the prototypical realization of four concrete, industrial application tasks.

As correctness of the built-in operating system is a crucial requirement for the reliability of safety- and security-critical systems, the goal of the Verisoft XT Avionics sub-project is to prove functional correctness of the microkernel in the partitioning hypervisor PikeOS, a commercial operating system for embedded systems [3].

For verification, we use tools like VCC (the Verifying C Compiler) developed by Microsoft Research, which follows the verifying compiler paradigm, 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 verification paradigm and the new tool are described in this paper.

* 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.

This Paper. In Section 2, we describe the PikeOS system and motivate why the particular system at hand and hypervisors in general are suitable targets for deductive program verification. We also explain why this effort matters in regulatory dependability frameworks such as DO-178B and IEC61508 for safety resp. Common Criteria for security. Then, in Section 3, we give an overview of the tool chain and the verification methodology used in the Verisoft XT Avionics project.

The goal of the project is the full functional verification of all system calls of PikeOS (i.e., the functionality that the kernel provides to guest systems). In this paper, we present our approach to verifying system calls using a system call for changing the priority of threads as an example. While this particular call has a simple functionality, its execution spans all levels of the PikeOS microkernel, from hardware-related levels to high-level kernel-functionality. Using this example, we first give a detailed account of how we handle (inline) assembly code blocks, which are needed to access hardware functionality that is not visible in plain C. We picture how the PowerPC assembly semantics can be specified, such that assembly functionality and especially the interaction with the C state can be verified (Sect. 4). Then, in Section 5 we use the same example to show how the system call has been specified and proved to be functionally correct using the VCC tool.

The same approach is being applied to verify system calls with more complex functionality that still span the same levels in the kernel as a call with simple functionality.

Related Work. The Avionics subproject of Verisoft XT builds upon previous work in the precursor project Verisoft I, where the pervasive verification of an academic microkernel written in the C0 dialect of C and running on verified DLX hardware was undertaken [14, 30]. Within Verisoft XT, in another subproject, the European Microsoft Innovation Center, DFKI and Saarland Univ. are verifying Microsoft's Hyper-V hypervisor.

Related work in kernel verification was already done in the '70s and '80s in the projects UCLA Secure Unix [36] and KIT [4], and more recently at the Universities of Dresden and Nijmegen (VFiasco project) [13] and in the EROS/Coyotos project [28]. A current project in kernel verification is L4.verified at NICTA (Australia) [19]. An overview and comparison of these and other related projects is given in [18].

2 A Hypervisor Approach to Dependability

Safety by Redundancy and Partitioning. Redundancy is a means for fault tolerance by error compensation: the erroneous state of a system must contain enough redundancy to enable the delivery of an error-free service from the erroneous state (of one component) [20]. In functional safety standards such as IEC61508 [16], to achieve SIL4 (the highest safety integrity level of that standard), each safety-related subsystem must be able to tolerate the occurrence of at least one hardware fault.

In a world where resources were unlimited, it would be best to have physically separated networks for each functionality. However, resources are limited in practice. While there is a need for physical redundancy, one tries to integrate redundancy networks for different functionalities. For example, in avionics (where weight constraints play an important role) this concept is called Integrated Modular Avionics (IMA) and has been specified in standards [1, 26], see also [7] for discussion. In the automotive sector, components have traditionally been more autonomous. While the maximal impact of a single catastrophic system failure is less severe than in avionics and virtualization does not have a long history of standards, virtualization has been characterized as a prevalent safety and security technology for most upcoming automotive IT architectures in the next decade [23].

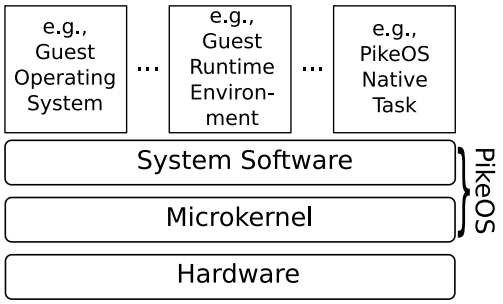

Fig. 1. Exemplary structure of a running PikeOS system

Hypervisors for Shared Use of Hardware. Once we accept the idea that sharing redundant hardware for different systems is a sensible trade-off, the problem is that almost all hardware used in industry (e.g. x86, PowerPC, MIPS, ARM) is designed to run in (at least) two privilege modes: privileged mode and user mode, where the privileged mode has a richer instruction set (on PowerPC, for example, allowing changes to the machine status register including the enabling and disabling of interrupts, memory address translation etc.). A well-known solution to sharing (not necessarily redundant) hardware is to virtualize the hardware using a hypervisor that runs several guest systems. It catches the guest system's use of privileged instructions and communicates with the hardware. Of course, it is an important feature for a hypervisor to guarantee that the guest systems do not and cannot interfere with each other: each guest runs as if it was running alone.

Hypervisors for Dependability. Besides separating the guest systems hypervisors administer hardware resources (this includes CPU time, thus scheduling). Their interface, however, is simpler than that of a typical desktop operating system. Hypervisors do not include higher-level functionality such as networking services and protocols, display drivers, graphical user interfaces, and user management.

This allows to have a small (trusted) hypervisor such as PikeOS running several less trusted larger systems as guests. For example, one can run a Linux system in one partition and (in avionics contexts) an ARINC-653 application or (in traffic control contexts) a POSIX or Java runtime-environment in another [32–35].

With today's state of the in deductive program verification, full functional verification is not feasible for a large guest system (e.g. Linux) but is within reach for the smaller hypervisors. The code size of the hypervisor PikeOS is smaller than that of Linux by several magnitudes.

In the areas of avionics, automotive, and automation, safety concerns (e.g., availability of the system) traditionally carry more weight than security concerns (e.g., confidentiality of the system's data). It has been noted, however, that in complex (virtualized) systems both notions are interconnected [24, 31]. In a Common Criteria context, this mix of safety and security features is labeled robustness [15].

Features of the PikeOS Hypervisor. PikeOS (see `http://www.pikeos.com/`) consists of a microkernel acting as paravirtualizing hypervisor and a system software component. The PikeOS *kernel* is particularly tailored to the context of embedded systems, featuring real-time functionality and orthogonal partitioning of resources such as processor time, user address space memory and kernel resources. The PikeOS *system software* component is responsible for system configuration. Thus the allocation of resources can be bound at compile-time, for

example to conform to partitioning requirements in the (aforementioned) Integrated Modular Avionics. At the kernel level, the mechanisms for communication between threads are IPC, events, and shared memory. High-level communication concepts such as Integrated Modular Avionics ARINC ports can be mapped onto these kernel-level mechanisms. For a thorough discussion of PikeOS and its evolution, see [17]. For an exemplary deployment of a running PikeOS system see Fig. 1. For concrete examples we (again) refer to [32–35].

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. PikeOS runs on many platforms, including x86, PowerPC, MIPS, and ARM among others and the exact amount of assembly depends on the architecture one works on. The verification target we have chosen for our project, is the PikeOS version for the PowerPC processor family, the OEA architecture, and the MPC5200 platform [11, 12]. In this particular case, PowerPC assembly is about one tenth of the codebase.

3 Verification Methodology and Toolchain

The Verifying C Compiler. In the verifying compiler 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 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 [8] (see `http://research.microsoft.com/vcc`). VCC uses a specification language tailored to C, which allows a verification engineer to write the specification in a way close to the syntax and semantics of the programming language.

Given an annotated C program, the VCC tool performs three steps to conduct a correctness proof (if possible). The reason for this breakdown into several steps is a better separation of concerns and easy integration of different tools: (1) The annotated C code is compiled into an intermediate imperative programming language called BoogiePL [10], which includes the specified properties of the C program rendered as assertions. (2) The input for the following translation step consists of two parts: (a) the BoogiePL code that results from compiling the original C source (including annotations) 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 first-order predicate logic formulas (verification conditions), which state that the program satisfies the annotated specification. (3) These verification conditions are given to the automatic theorem prover Z3 [9] to check whether they are valid, which then implies that the original C program is correct w.r.t. the annotated specification.

The possible results Z3 may return are: (1) The formulas are valid (Z3 has found a proof). (2) At least one of the formulas is not valid (Z3 has found a counter-example). (3) Z3 runs out of resources (time or space). In Case (1) above, the program verification was successful. In Cases (2) and (3), the verification engineer has to analyse the problem (using a possible counter-example) and correct the error. In Case (3), he/she may also find that the program indeed satisfies the annotations. Then new annotations (stronger invariants, helpful lemmas, etc.) have to be added. This process is repeated until Z3 finds a 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. Also, VCC itself can be used as a C compiler (compilation mode).

Specification Language. In this section, we give a brief overview of VCC's specification language, focusing on those language features occurring in the examples of PikeOS code used throughout this paper (loop invariants, for example, are part of the language but are not discussed here).

Annotations, Implementation Variables and Ghost Variables. Annotations are written in the form `keyword(block)` where `keyword` gives specifies the specification constraint to be enforced. In a block the verification engineer writes (depending on the `keyword`) an expression or statements (statements again may contain expressions). Expressions are written in a C-like syntax and may use implementation variables. They also may use object variables that are not part of the implementation state (called "ghost variables"). Declarations of ghost variables are guarded by `spec()`, and statements changing values of ghost variables are guarded by `speconly()`. Expressions in annotations also may use implementation variables and ghost variables simultaneously. Note that annotations must not affect the actual behavior of the program (for example, unlike in C, it is not allowed to write `assert(++x==1)` which would change the value of `x`). They merely express what the program is supposed to do. They make explicit those (implicit) restrictions that a program adheres to anyway.

Object Invariants and Ownership. One way to capture global properties of a software system is to define invariants for data structures (i.e., `structs` in the case of C) used in the program. 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 `structs`), 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 [21]). Every `struct` has exactly one "owner" and can itself own arbitrarily many structures. At the top of the ownership hierarchy, `structs` can be owned by executing threads. The ownership relation is provided explicitly in annotations by the verification engineer, and it reflects his/her abstract knowledge about the data structure and how it is used.

It would not be efficient to always check all invariants on all data structures. Hence, a data object can have two states, `open` and `closed`. The convention is that a thread only may change objects it owns and that it may force a check on all invariants by `wrap`ping an object, that is by moving it from `open` to `closed`. Ignoring volatile variables, a 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. That is relevant in a concurrent setting when a context switch may occur.

Function Pre- and Postconditions. The specification of a C function can be seen as a contract between the caller of the function and the function itself and is given by pre- and postconditions. These are inserted between the function head and the function body, as shown in the following example:

```
1 int max(int a, int b)
2   requires (a >= 0 && b >= 0)
3   ensures  ((result == a || result == b) &&
4             result >= a && result >= b)
5 { if (a > b) return a;
6   return b;
7 }
```

The precondition of a function is labeled with the keyword `requires`, and the postcondition is labeled with `ensures`. The keyword `result` represents the return value of the function. In the examples of the following sections, the reader will also encounter ownership relations asserted at the function level.

Further Specification Constructs. For convenience, `maintains` can be used for a property that a function both `requires` and `ensures`. `writes` can be used to denote the set of memory locations to which a function (at most) writes. `returns` is shorthand for `ensures` on a `result`. Objects that are closed and owned are called `wrapped`, objects that are open, owned and typed are called `mutable`, which means that they may be modified. `keeps` denotes that an object owns a certain set of objects only (and nothing else).

Guidance for the Automatic Proving Engine. The verification engineer may have to give some hints to the prover: this is achieved by `bv_lemma` for bitvector related lemmas, `assert` for intermediate assertions to be enforced and several other annotations (e.g., on the conservation of properties throughout function calls), which are excluded from the listings for clarity.

4 Verification of Assembly Code and Low-Level Functions

A peculiarity that distinguishes real-world microkernels from other C programs is the high percentage of assembly code which is found in separate assembly files (macro assembly) as well as inlined into the C code using the `__asm__` keyword. Moreover this assembly contains privileged mode instructions which are commonly neglected in formal definitions of instruction set semantics for user space programs [2, 5]. Because of the ubiquity of assembly in the near-hardware layers of PikeOS and the inability of VCC to interpret machine instructions, we have to define our own machine and assembly model. In this paper we especially focus on the inline assembly portions, where C and hardware semantics are mixed and data is interchanged between the models. To our best knowledge only the Verisoft I project [14, 30] achieved substantial progress in this specific field. However our approach is different in that we chose an industrial microprocessor (Freescale MPC5200) as the target architecture for the Verisoft XT avionics subproject. In addition the syntax and semantics of inline assembly in compilers like gcc is more complex than in the one used in Verisoft I and we lack formal compiler semantics. Nevertheless in some places (e.g. inline assembly, memory and branch instructions) we have to simulate the behavior of the compiler assuming its correctness.

In the following, we introduce our approach to modeling the hardware and the semantics of assembly instructions. Furthermore we exemplify how this methodology is applied to actual PikeOS functions in the second part of this section.

4.1 Defining the Semantics of Privileged Mode PowerPC assembly

We establish the semantics of assembly instructions as a transition relation over hardware configuration. First we have to make an abstraction and identify the hardware components that must be represented. Then the effect of each PowerPC instruction can be stated writing a specification function which reflects the corresponding impact on the hardware configuration.

Modeling Hardware Components in the Global PowerPC Ghost State. The PowerPC core of the MPC5200 microprocessor comprises a variety of different components, such as general purpose registers (GPRs), special purpose registers and other user and system registers, caches,

translation look-aside buffers (TLBs). Also the physical memory must be taken into consideration. When building a hardware model for a modern processor on the C level, several questions arise.

Where is the model defined and how does it interact with the kernel implementation? In fact there are two options. One could either add the model to the kernel implementation and replace all assembly commands with their representative model functions. However, this would modify the structure of the C state and thus corrupt the verification effort. The alternative is to keep the whole hardware model in the specification-only ghost state. At first sight, with this approach one would face the problem to transfer data between the hardware ghost state and the C state, which VCC forbids. However, one can circumvent this issue via specification functions that indirectly "assign" some value to a C variable by assuming their equality as a postcondition (an example of such a function will be shown later).

Following these thoughts we defined the global hardware configuration `PPC_c` in ghost state, which then can be modified by assembly instructions only:

```
1 spec(struct PPC_config_struct {
2   // exemplary components:
3   PPC_B32_t stackPtr;     // stack pointer
4   PPC_MSR_t msr;          // Machine State Register
5   invariant(keeps(&msr))  // ownership invariant
6 } PPC_c;)
```

As C code runs on the underlying hardware, how can these effects be captured? Basically this is impossible as there is no formal C and compiler semantics available to project the execution of C statements to the machine level. Nevertheless assembly code should not rely on the effect of preceding C statements on the processor state (this would imply the programmer applies knowledge about the C and compiler semantics). Hence we simply divide the hardware components into those that are not changed by C statements (like system and special purpose registers) and those that are affected by the execution of C (basically the user-visible registers). The former ones are comprised in the `PPC_c` structure while the latter ones only become visible as local variables in the context of assembly code. In case compiler properties were assumed about one of these globally invisible components, it can be initialized accordingly at the beginning of the assembly block (cf. Sect. 4.2 for an example).

How should caches and TLBs be handled? In a single-processor setting caches and TLBs (which are in fact also just caches for page address translations) are invisible to the programmer under certain realistic assumptions. Therefore we do not need to model caches and TLBs. However, separate proofs to validate these claims are in order.

How do we model the physical memory? As we are lacking a formal compiler semantics which would define the memory allocation of C variables and code, modeling the complete data and instruction memory is also impossible. Hence memory does not belong to our global hardware model. For memory and branch instructions special measures are taken locally.

Specifying PowerPC Assembly Instructions. Based hardware model we can specify the effect of the execution of assembly code. For each instruction we define a specification function which is equipped with pre- and postconditions that reflect the functionality of the particular instruction. For plain register transfer operations this is easy. E.g. there are privileged mode instructions `mfmsr` and `mtmsr`, which **m**ove GPR contents **f**rom resp. **t**o the machine status register (MSR). The specification function for MFMSR is given below:

```
1 spec (void PPC_MFMSR(PPC_B32_t *dest)
2   maintains(wrapped(&PPC_c))
3   maintains(mutable(dest))
```

```
4   writes(dest)
5   ensures( *dest == PPC_c.msr.reg );)
```

The first two `maintains`-clauses refer to the memory resp. ownership-model of VCC. The `writes`-clause specifies that the destination register is written by the instruction. The last line states the postcondition of MFMSR. In this way, most of the PowerPC instructions can be handled. However, for memory and branch instructions there may be access to the kernel and user address spaces, which requires additional handling. How to model these instructions is outside the scope of this paper, though.

Assembly Code Translation. After having defined the hardware configuration and instruction semantics, the remaining question is how to integrate the hardware model into the PikeOS code. As VCC does not recognize assembly code, these commands have to be replaced by the corresponding specification function calls, which then simulate the execution on the model. The translation of the assembly instructions to our specification functions can be done automatically using a parser. This has already been proposed for x86 assembly in the Verisoft XT Microsoft hypervisor (Hyper-V) subproject [22].

The methodology pictured above applies to the simulation of macro assembly. Fortunately, the semantics of inline assembly does basically not differ from macro assembly semantics. The main additional effort is to establish an interface between the context of a C function and the general purpose registers in the contained assembly code. To this end one has to examine the PowerPC ABI and compiler specifics concerning the syntax and semantics of an `__asm__` statement. An important feature is the aliasing of registers to which the values of C variables are assigned. The compiler can essentially choose an arbitrary register that is not yet occupied to store the C data for the assembly computation. Thus it is hard to exactly determine the registers that will contain the respective data without looking into the compiled code (one would require a formal compiler specification). However, it is not necessary to know the exact distribution of data over the registers. We can just choose any free registers for that purpose. Then the registers have to be initialized with the respective data and after execution of the inline assembly block the results have to be written back.

4.2 Verifying Low-Level PikeOS Functions.

We now want to demonstrate the application of our PowerPC assembly model to three exemplary functions which are called from `p4syscall_fast_set_prio`, a PikeOS system call function that shall be examined later on. The three helper functions contain inline assembly code, which is translated according to the methodology pointed out above. Afterwards the resulting functions are subject to annotation and automated verification with VCC.

Translation of Inline Assembly Code. Firstly we look at the translation from (inline) assembly language to hardware model functions. To do so we introduce the first auxiliary function called by `p4syscall_fast_set_prio`, namely `p4arch_disable_int`:

```
1 static inline P4_cpureg_t p4arch_disable_int(void)
2 {
3   P4_cpureg_t ret;
4   P4_cpureg_t val;
5   __asm__ volatile("mfmsr %0" : "=r"(ret));
6   val = ret & ~MSR_EE;
7   __asm__ volatile("mtmsr %0" : : "r"(val) : "memory");
8   return ret;
9 }
```

This function disables the signaling of external interrupts by clearing the corresponding EE bit in the CPU's machine status register (MSR). It returns the old value of the MSR.

The translation is done in two steps, where the first one is only needed for inline assembly. Here we insert definitions of the local specification variables that establish the hardware context. Moreover the syntax of the `__asm__` statement is parsed and the corresponding pairs of register aliases and C variables are extracted. A free hardware register is determined and set to the value of the C variable it was allocated to. After the assembly block the results are written back to the variables when necessary. This is achieved by assigning the register values to an intermediate local variable using the specification function `PPC_assign`.

In the second step the assembly syntax is parsed and the commands are translated to their hardware model counterparts, which are specification functions operating on the ghost hardware state. The resulting annotated C code for `p4arch_disable_int` looks like this:

```
1  static inline P4_cpureg_t p4arch_disable_int(void)
2  {
3    P4_cpureg_t ret;
4    P4_cpureg_t val;
5    // inline asm variables and initialization
6    spec(PPC_B32_t gpr[32];)              // step 1
7    void * PPC_ret;                       // step 1
8    // start inline asm block
9    PPC_MFMSR(spec(&gpr[3]));             // step 2
10   PPC_assign(spec(&PPC_ret,gpr[3]));    // step 1
11   ret = (P4_cpureg_t)PPC_ret;           // step 1
12   // end inline asm block, assigning return values
13   val = ret & ~MSR_EE;
14   // start inline asm block, passing parameters
15   speconly(gpr[4] = val;)               // step 1
16   PPC_MTMSR(spec(gpr[4]));              // step 2
17   // end inline asm block
18   return ret;
19 }
```

The new lines resulting from the translation are printed in bold face and labeled with the translation step in which they are produced. Ghost code and ghost variable declarations are included using the `speconly` and `spec` keywords (Sect. 3). Functions and data belonging to the hardware model are labeled with the `PPC_` prefix. Registers 3 and 4 are used as they are the first available registers according to the ABI. Note that the translated version of the code does not replace the original functions at run-time. Using `ifdef` case distinctions it is only visible to the compiler when the verification mode is enabled.

Annotation and Verification. Replacing the assembly commands by calls to their representative functions in the hardware model enables us to discuss the functionality of the code. We can now add annotations to the code which specify the behavior that each particular function is supposed to implement. These assertions on the code are then validated by VCC and may be assumed to hold on the caller functions at higher levels of the call graph.

As an example for the verification of function-level annotations and introduce the counterpart to `p4arch_disable_int`, we consider the function `p4arch_restore_int`, which restores the bit MSR.EE from a given value `msr`.

```
1  static inline void p4arch_restore_int(P4_cpureg_t msr)
2  {
3    unsigned ret;
```

```
4   unsigned val;
5   __asm__ volatile("mfmsr %0" : "=r"(ret));
6   val = ret | (msr & MSR_EE);
7   __asm__ volatile("mtmsr %0" : : "r"(val) : "memory");
8 }
```

For simplicity, here we concentrate on two properties of the function: (1) MSR.EE is set to the value of the corresponding bit in parameter `msr`. (2) All other MSR bits (e.g., the current privilege mode MSR.PR) are preserved. The translated code is shown below with all necessary annotations to specify these properties (in bold face).

```
1 static void p4arch_restore_int(P4_cpureg_t msr)
2 writes(&PPC_c)
3 ensures(PPC_c.msr.fld.EE == (old(PPC_c.msr.fld.EE)|GET_BE(msr,16)))
4 ensures(PPC_c.msr.fld.PR == old(PPC_c.msr.fld.PR))
5 {
6   unsigned ret;
7   unsigned val;
8   spec(PPC_B32_t gpr[32];)
9   void * PPC_ret;
10  PPC_MFMSR(spec(&gpr[3]));
11  assert(gpr[3] == (P4_cpureg_t)PPC_c.msr.reg);
12  PPC_assign(spec(&PPC_ret,gpr[3]));
13  ret = (P4_cpureg_t)PPC_ret;
14  val = ret | (msr & MSR_EE);
15  bv_lemma(forall(unsigned int x,y;
16     (GET_BE(x|(y & MSR_EE),16) == (GET_BE(x,16)|GET_BE(y,16)))));
17  bv_lemma(forall(unsigned int x,y;
18     (GET_BE(x|(y & MSR_EE),17) == GET_BE(x,17))));
19  speconly(gpr[4] = val;)
20  PPC_MTMSR(spec(gpr[4]));
21 }
```

As the global hardware model is modified by the assembly portions it has to be included in the `writes` clause. Using the macro `GET_BE` we access certain bits of the `msr` parameter in big endian order. The properties of the function as stated in the `ensures` clauses are specifying a transition relation between the old (marked `old`) and new state of the data structures (the hardware configuration `PPC_c` in this case).

To split the complexity of the problem of proving the postcondition, intermediate asserts like in line 15 are helpful. A problem for automated provers is the handling of non-linear arithmetic, such as bit vector operations. Here VCC offers an extended axiomatization via `bv_lemma`, which can be used to introduce and validate bit-vector-related lemmata where they are necessary to verify corresponding C code.

With the few additional assertions as "step stones" for the verification, VCC is able to prove the two postconditions of the function shown above in about 5 seconds on an AMD Athlon 64 X2 Dual Core 4000+ processor. Adding the postconditions and additional annotations for the other 15 relevant MSR bits increases verification time to roughly 9 seconds.

Reading the Stack-Pointer. A special problem concerning the combination of C and inline assembly arises with the following function.

```
1 static inline void *p4arch_runner(void)
2 {
3   register unsigned long sp __asm__("r1");
```

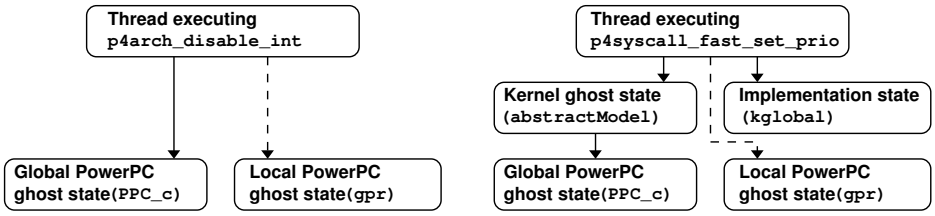

Fig. 2. Verification setups for the exemplarily low-level function `p4arch_disable_int` introduced in Sect. 4 *(left)* and the entire system call introduced in Sect. 5 *(right)*. Straight arrows indicate VCC ownership relations (Sect. 3), dashed arrows indicate implicit dependencies.

```
4   return (void *)(sp & ~(P4_PROCESS_SIZE - 1));
5 }
```

`p4arch_runner` reads the stack pointer which by default is stored in general purpose register 1 [37]. Then the base address of the current process is computed from the value by removing its offset and returned to the caller as a void pointer. However, the value of the stack pointer is unknown as it is a concept introduced by the compiler and not visible on the C level. To solve this problem, we use the knowledge that the pointer lies in a certain address range, whose base address is determined during process switch, when register 1 is set to point to a stack frame of a specific process. Then, we keep track of these assignments using a global variable `stackPtr` that is part of the hardware model. When necessary, the helper function `set_stackptr` is used to assign the value of `stackPtr` to a general purpose register. Although we only specify a range for the stack pointer we still obtain an exact value (i.e. the base address of the process) by clearing the offset bits. The postcondition of `p4arch_runner` can be written as: `returns((void *)(BASE_ADDR(PPC_c.stackPtr,P4_PROCESS_SIZE)))`, where `BASE_ADDR(`x`,`2^i`)` is a macro computing the bit vector consisting of the high $32 - i$ bits concatenated with i zeroes for unsigned integers x, i with $x < 2^{32}$ and $i < 32$.

5 System Call Verification

In this section, we show how a system call of PikeOS can be verified based on the hardware model and the specification and verification of hardware-related layers of the kernel presented in the previous section.

Verification Setup. Recall that to structure a complex state space, VCC supports the use of an ownership relation between data structures (see Sect. 3). The right part of Fig. 2 shows the situation for a system call in PikeOS.

Because system calls are at the user's interface to the kernel and the PikeOS system is multi-platform, the kernel's specification has to hide any PowerPC implementation details to ensure proper encapsulation. In our modeling this simply means that the abstraction of the kernel's state in ghost state, specified as `abstractModel`, thus *owns* the PowerPC machine model `PPC_c` as formalized by the invariant `keeps(currentThread, &PPC_c)`.

```
1 spec( struct absModel_str {
2     bool interruptsEnabled;
3     invariant(interruptsEnabled == (PPC_c.msr.fld.EE == 1))
4     struct P4k_thrinfo_t *currentThread;
5     invariant(keeps(currentThread, &PPC_c))
```

```
6     invariant(currentThread != NULL)
7 } abstractModel; )
```

The specifications of the C methods on the upper layers of the kernel, like system calls, can now be written in terms of the elements of the abstract model.

An Exemplary System Call. As a first target for verification we have chosen the system call `p4syscall_fast_set_prio`, which changes the priority of a thread. The parameter `newprio` of the system call may not exceed the user-configured *Maximum Controlled Priority* (MCP).

This call has a rather simple functionality, but it serves very well as an example because its execution spans all levels of the PikeOS microkernel, from high-level kernel functionality to hardware-related levels and the user-level interface (system calls are invoked via user interrupts). Systems calls with more complex functionality still span the same levels.

For verifying the `p4syscall_fast_set_prio` system call, two components of the abstract model are needed, namely `interruptsEnabled`, which indicates whether the system currently allows external interrupts to occur, and a pointer to the thread currently running in kernel mode that is given by `currentThread`. These two elements of the abstract model are related to the underlying hardware and hence its representation as the ghost structure `PPC_c`. This relation is explicitly stated as invariants of the `abstractModel` data structure (to save space we only show the invariant for `interruptsEnabled`).

Whether external interrupts are allowed or disallowed in the kernel is indicated by the EE bit in the MSR register of the hardware. In the global ghost state model of the PowerPC hardware as described in Sect. 4.2, this flag is represented by the field `PPC_c.msr.fld.EE`. In `abstractModel`, interrupts are then defined to be enabled, iff this bit in the hardware model is set to 1, as stated by the invariant in line 3 of the specification of `absModel_str`: `interruptsEnabled == (PPC_c.msr.fld.EE == 1)`.

We now consider the actual C and annotation code for the system call under consideration. Setting the new priority values in the data structures of the thread and, for the purpose of faster look-up, in a global info data structure of the kernel, is done by the helper function `p4_runner_changeprio`:

```
1 P4_prio_t p4_runner_changeprio(P4k_thrinfo_t *proc,
2                                P4_prio_t newprio)
3   writes(&abstractModel, &kglobal)
4   requires(proc == abstractModel.currentThread)
5   maintains(wrapped(&abstractModel) && wrapped(&kglobal))
6   ensures(proc->schedprio == newprio &&
7           kglobal.kinfo->currprio == newprio)
8   returns(old(proc->userprio))
9 {
10  P4_prio_t oldprio; P4_cpureg_t oldstat;
11  unwrap(&abstractModel);
12  oldstat = p4arch_disable_int();
13  speconly(abstractModel.interruptsEnabled = 0;)
14  unwrap(proc);
15  oldprio = proc->userprio; proc->userprio = newprio;
16  proc->schedprio = newprio;
17  wrap(proc); wrap(&abstractModel);
18  unwrap(&kglobal); unwrap(kglobal.kinfo);
19  kglobal.kinfo->currprio = newprio;
20  wrap(kglobal.kinfo); wrap(&kglobal);
```

```
21   unwrap(&abstractModel);
22   p4arch_restore_int(oldstat);
23   speconly(abstractModel.interruptsEnabled = PPC_c.msr.fld.EE;)
24   wrap(&abstractModel);
25   return oldprio;
26 }
```

First, this function disables handling of external interrupts by calling `p4arch_disable_int` (line 12), so that from here on concurrency does not need to be considered. Before this, the struct `abstractModel` has to be unwrapped (line 11) because `p4arch_disable_int` writes to the struct `PPC_c`, which is owned by the `abstractModel`.

After `p4arch_disable_int` has set the EE bit of the MSR variable in `PPC_c` to 1, the invariant of `abstractModel` does not hold anymore. Before `abstractModel` can be wrapped again (line 17), its invariant has to be restored. This is achieved by updating the `interruptsEnabled` flag of `abstractModel` (line 13). After the interrupts are disabled, the different updates on the priority values of the thread and kernel information data structure (`kglobal.kinfo`) can be performed (lines 14–20). Restoring the interrupt-enabled state (lines 21–24) and `return`ing the old priority of the thread complete this method.

Using VCC it is now possible to prove that the function satisfies its specification given in lines 3–8. For this, in fact, several intermediate assertions before and after calls to helper functions are necessary to let the verification system validate that certain properties have been preserved during method calls (we have omitted these "lemmas" for brevity).

Following our bottom-up verification scheme we arrive at `p4syscall_fast_set_prio` which calls `p4_runner_changeprio` (see above) among other functions. The implementation of the system call (not shown here), uses the helper methods `p4_runner`, which returns the thread data structure for the current thread, and `p4_runner_changeprio`, which changes the priority values of the thread. The method `p4_runner` is a wrapper for the function `p4arch_runner` (see Sect. 4.2). The specification of `p4_runner` abstracts from the concrete return value of `p4arch_runner`, i.e., the address of the page to which the stack pointer points, and instead returns the ghost variable `abstractModel.currentThread`. This abstraction is valid as it is a system invariant that the stack pointer for the kernel stack always points to the page corresponding to the current thread. The thread data structure of the current thread is placed at the beginning of this particular page.

This, finally, allows us to verify the following method contract for our exemplary system call `p4syscall_fast_set_prio`:

```
1 P4_uint32_t p4syscall_fast_set_prio(P4_uint32_t prio)
2   writes(&abstractModel, &kglobal)
3   maintains(wrapped(&abstractModel) && wrapped(&kglobal))
4   ensures(prio <= abstractModel.currentThread->mcprio ?
5             abstractModel.currentThread->schedprio == prio
6             && kglobal.kinfo->currprio == prio
7           : abstractModel.currentThread->schedprio ==
8             abstractModel.currentThread->mcprio
9             && kglobal.kinfo->currprio ==
10               abstractModel.currentThread->mcprio)
11 { ... }
```

The postcondition of this method (`ensures` clause in lines 4–10) directly matches the informal specification in 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."

Besides this postcondition, the contract specifies that the method is (only) allowed to write to `abstractModel` and `kglobal` (line 2), and that these two data structures are required to be wrapped according to the ownership methodology of VCC, i.e., the thread that is currently executing the method is in possession of these data structures, all their non-volatile fields remain unchanged and all their invariants hold.

6 Conclusion

Verification Setup for a System Call. We have presented the use of deductive program verification in the Verisoft XT Avionics subproject. The formalization of PowerPC assembly semantics enables us to verify kernel functionality spanning all levels of the PikeOS microkernel. In particular, we have shown how interrupts are disabled and then restored again to ensure that the bulk of the system call is in non-concurrent mode. The same approach can be applied to verify system calls with more complex functionality as these still span the same levels in the kernel as a call with simple functionality (this is ongoing work).

Pervasive Verification in Industrial Dependability. PikeOS is developed to the DO-178B [25] avionics safety standard, which requires structural coverage of both high-level and low-level (function- or module-level) requirements. In the practice stipulated by the current (1992) version of DO-178, this enforces the generation of a very large amount of test cases (e.g., the amount of PikeOS test-case code by far exceeds the amount of PikeOS system code). However, due to the combinatorial explosion of states, for any moderately complex system truly exhaustive testing becomes impossible. For instance, to supplement test cases, for DO-178B level A flight-control software, Airbus has developed the concept of "unit proofs" [29], which can be used to gain assurance for low-level requirements of modules (in analogy to unit tests). A formal methods appendix that is (in particular) compatible with the deductive program verification approach is under development for the upcoming DO-178C [27], again not only targeting high-level but also low-level requirements. Here we expect the output of the deductive program verification approach to be of value. Similarly, for the highest level SIL4 in the functional safety standard IEC61508, formal methods are highly recommended for software design down to the individual modules. In the area of security, it is labor-intensive and expensive to produce evidence for the fulfillment of (structural) security assurance requirements such as FDP_FSP.6, ADV_IMP.2 and ADV_TDS.6, which are required by the highest Common Criteria EAL level. In cooperation with another branch of the Verisoft XT Avionics project, we have come to the conclusion that such evidence could be derived from deductive program verification artefacts.

Future Work. It is current work to apply the verification approach presented in this paper to all the system calls and interrupt handlers of PikeOS to get a full functional verification of the kernel. In a next step, we will then consider the effects of concurrency when parts of the kernel are executed without disabling of interrupts. Support for verifying concurrency has recently been added to VCC by its developers [6].

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 Alexander Züpke, Jacques Brygier, Knut Degen, Tobias Stumpf, Stephan Wagner (SYSGO AG), the Verisoft XT ES.1 group, Dirk Leinenbach, Mark Hillebrand (DFKI), Marko Wolf (escrypt) and the VCC research team at Microsoft Research (EMIC), in particular Markus Dahlweid, Michał Moskal, Thomas Santen, and Stephan Tobies, and the participants of the RTCA SC-205/EUROCAE WG-71 formal methods (SG06) group meeting (Cologne, Feb 2009).

References

1. Aeronautical Radio, Inc. Avionics application software standard interface. ARINC specification 653.
2. J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *DAMP '09: Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming, Savannah, GA, USA*, pages 13–24. ACM, 2009.
3. C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Better avionics software reliability by code verification. In *Proceedings, embedded world Conference, Nuremberg, Germany*, 2009.
4. W. R. Bevier. KIT: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
5. S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.
6. E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-15, Microsoft Research, 2009. Available at `http://research.microsoft.com/vcc`.
7. P. M. Conmy. *Safety Analysis of Computer Resource Management Software*. PhD thesis, University of York, 2005.
8. M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. Available at `http://research.microsoft.com/vcc`.
9. 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.
10. 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.
11. Freescale Semiconductor. *Programming Environments Manual for 32-Bit Implementations of the PowerPC™ Architecture*, 3rd edition, September 2005. Available at `http://www.freescale.com/files/product/doc/MPCFPE32B.pdf`.
12. Freescale Semiconductor. *MPC5200B User's Manual, Rev. 1.3*, Sep 2006. Available at `http://www.freescale.com/files/32bit/doc/ref_manual/MPC5200BUM.pdf`.
13. 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.
14. T. In der Rieden and A. Tsyban. CVM: A verified framework for microkernel programmers. In G. K. R. Huuck and B. Schlich, editors, *3rd International Workshop on Systems Software Verification (SSV08)*, volume 217 of *ENTCS*, pages 151–168. Elsevier Science B.V., 2008.
15. Information Assurance Directorate. U.S. government protection profile for separation kernels in environments requiring high robustness. Version 1.03, June 2007. Available at `http://www.commoncriteriaportal.org/files/ppfiles/pp_skpp_hr_v1.03.pdf`.
16. International Electrotechnical Commission. *Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 2: Requirements for electrical/electronic/programmable electronic safety systems*. CEI/IEC 61508-2:2000. ISO/IEC, 1st edition, 2000.
17. 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. Available at `http://ertos.nicta.com.au/publications/papers/Kuz_Petters_07.pdf`.
18. G. Klein. Operating system verification: An overview. Technical Report NRL-955, NICTA, Sydney, Australia, June 2008.
19. 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.
20. J. Laprie, editor. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, 1992.
21. 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.
22. 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.

23. J. Pelzl, M. Wolf, and T. Wollinger. Virtualization technologies for cars: Solutions to increase safety and security of vehicular ECUs. In *Proceedings, embedded world Conference, Nuremberg, Germany*, 2009.
24. A. Pfitzmann. Why safety and security should and will merge. In M. Heisel, P. Liggesmeyer, and S. Wittmann, editors, *SAFECOMP*, LNCS 3219, pages 1–2. Springer, 2004.
25. Radio Technical Commission for Aeronautics. *Software Considerations in Airborne Systems and Equipment Certification*. DO-178B/ED-12B. Radio Technical Commission for Aeronautics (RTCA), Inc., 1828 L Street NW, Suite 805, Washington, D.C. 20036, Dec. 1992.
26. Radio Technical Commission for Aeronautics. *Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*. DO-297. Radio Technical Commission for Aeronautics (RTCA), Inc., 1828 L Street NW, Suite 805, Washington, D.C. 20036, Nov. 2005.
27. RTCA SC-205/EUROCAE WG-71. Discussion and development site for Software Considerations in Airborne Systems, 2009. At `http://forum.pr.erau.edu/SCAS/`.
28. 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.
29. J. Souyris and D. Favre-Félix. Proof of properties in avionics. In R. Jacquart, editor, *IFIP Congress Topical Sessions*, pages 527–535. Kluwer, 2004.
30. A. Starostin and A. Tsyban. Correct microkernel primitives. In G. K. R. Huuck and B. Schlich, editors, *3rd International Workshop on Systems Software Verification (SSV08)*, volume 217 of *ENTCS*, pages 169–185. Elsevier Science B. V., 2008.
31. V. Stavridou and B. Dutertre. From security to safety and back. In *Conference on Computer Security, Dependability and Assurance*, pages 182–195, 1998.
32. SYSGO AG. PikeOS selected for traffic control system. Press release on Aug 07, 2008, available at `http://www.sysgo.com`, 2007.
33. SYSGO AG. AIRBUS selects SYSGO's pikeos as DO-178B reference platform for the A350 XWB. Press release on Nov 18, 2008, available at `http://www.sysgo.com`, 2008.
34. SYSGO AG. Flight management system will run on SYSGO's PikeOS in the DIANA project. Press release on Jul 17, 2008, available at `http://www.sysgo.com`, 2008.
35. SYSGO AG. Rheinmetall selects DO178B certifiable PikeOS from SYSGO for A400M project. Press release on Dec 10, 2008, available at `http://www.sysgo.com`, 2008.
36. B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.
37. S. Zucker and K. Karhi. *System V Application Binary Interface: PowerPC Processor Supplement*. SunSoft, Mountain View, CA, USA, 802-3334-10 edition, Sept. 1995. Available at `http://refspecs.freestandards.org/elf/elfspec_ppc.pdf`.