Delft University of Technology
Software Engineering Research Group
Technical Report Series

Verifying FreeRTOS; a feasibility study

C. (Kees) Pronk

Report TUD-SERG-2010-042

TUDelft

SERG

TUD-SERG-2010-042

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note:

© copyright 2010, by the authors of this report. Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the authors.

Verifying FreeRTOS; a feasibility study

C. (Kees) Pronk

Delft University of Technology, The Netherlands
c.pronk@tudelft.nl

Abstract. This paper presents a study on modeling and verifying the kernel of Real-Time Operating Systems (RTOS). The study will show advances in formally verifying such an RTOS both by refinement and by model checking approaches. This work fits in the context of Hoare's verification challenge. Several real-time operating systems will be discussed including some commercial ones. The focus of the latter part of the paper will be on verifying FreeRTOS. The paper investigates a number of ways to verify this operating system. A preliminary set-up of verifying FreeRTOS using model checking is presented.

1 Introduction

A software product should meet all of its requirements and fully conform to its specification. Testing and other quality assessment techniques can only help to approach this goal [17]. Proving that a piece of software will always work according to its specification requires the use of formal methods. Two approaches to using formal methods exist: *correctness by construction* [28] and *post facto*. The first approach can be used to construct an implementation starting from an abstract formal specification. This technique is sometimes referred to as *refinement* and relies upon the use of theorem provers. The second approach, often called *verification*, extracts a model from existing code and uses state based technologies like model checking to determine the correctness of the model.

Tony Hoare has proposed a Grand Challenge project (GC6) [22], whose long-term vision is to develop methodologies and a set of automated tools that can be used to verify whether a piece of software meets its requirements [8]. One of the steps towards the realization of this vision is to build a repository [57] of formalized software designs and verified implementations, which can be used to test and develop the before mentioned tools.

The first case study carried out for the Verification Grand Challenge was Mondex [57], a smart card functioning as an electronic purse. The second case study was the POSIX-like file system suggested by Gerard Holzmann at the VSTTE conference in Zürich in 2005 [58]. The author of the present paper contributed to that goal showing how model checking could be used to prove the correctness of a Flash file system in the presence of power failures [51]. Further case studies in GC6 involve a Cardiac Pacemaker [36, 41] and the verification of the kernel of the FreeRTOS Real-Time Operating System.

Verifying a real-time system (RTOS) is considered difficult for many reasons. An RTOS creates and operates on a number of concurrent user tasks, promises to schedule those tasks within guaranteed time-limits using some priority mechanism, guarantees non-interference between these tasks and between the tasks and the kernel, allows these tasks to communicate using queues, synchronizes these tasks using mutexes or semaphores, and interacts directly with hardware through device drivers, clocks and interrupt mechanisms. An RTOS usually consists of a kernel mostly programmed in the C language and some assembler; the kernel heavily uses dynamic constructs like lists accessible through pointers. An excellent overview of the issues involved can be found in [29]. Each of these topics has already been formally investigated in isolation. Nowadays, full verification of an RTOS is slowly becoming feasible.

This paper does not target pervasive verification of the whole system stack consisting of hardware, compilers and the kernel but concentrates on verification of the kernel itself. This paper also does not target timing overhead verification of particular operating system functions such as context switching time. Such verifications are too much dependent upon the particular hardware used. Interested readers are referred to [13]. Instead, the paper presents an overview of the issues involved in verifying a real-time kernel, discusses the levels of verification technology available and tries to set up a course to do formal verification using one particular technology (model checking).

The focus of the latter part of this paper will be on verifying FreeRTOS. Section 2 discusses the structure of the FreeRTOS source code and run-time model. Section 3 explains which methods can be used and have been used to embark on the verification task. This section also mentions related work in this area. From this section a preferred method will be chosen for further development. Sect. 4 presents the current state of the project and discusses further steps to be taken and Sect. 5 summarizes the paper.

Reservation: In an overview article like this, simplifications and shortcuts are inescapable. The reader is referred to the literature references if in doubt.

2 What is FreeRTOS?

Cited from the web page: "FreeRTOS[1] is a portable, open source, royalty free, mini Real Time Kernel. It is a free to download and free to deploy RTOS that can be used in commercial applications without any requirement to expose your proprietary source code. It has been downloaded more than 77,500 times during 2008. FreeRTOS is the cross platform de facto standard for embedded micro controllers" [5, 6].

The kernel of FreeRTOS has been written in C (2500 lines of .c-files, 800 lines of .h-files and some 100 lines of assembler). These are real lines of code; not comments. FreeRTOS has currently been ported to 24 different microprocessor architectures; ranging from 'minimal' 8-bit architectures to full blown 32-bit

[1] Several (RT)OSes may be protected by trade marks. The reader is referred to the respective web-sites.

architectures. Depending upon the architecture, the kernel is compilable on 14 different compilers. Depending upon the application at hand, the user can include or exclude facilities like mutexes, tracing and scheduling mechanisms.

The source code is heavily parameterized upon the above items using the include mechanism of the C-language which makes extracting a proper subsystem to be used for verification a bit cumbersome.

A verified/validated version of FreeRTOS is commercially available upon request. SAFERTOS [46] has been certified for the following standards: IEC 61508 [25], FDA 510K [19] and DO-178B [45].

In order to better understand the complexity of full verification, a number of features of the FreeRTOS kernel is given below:

– Multiple tasks with priorities exist; tasks may be created in run-time. Coroutines with priorities are optionally available too. The scheduler uses prioritized preemptive or cooperative mechanisms to schedule those tasks. Tasks are synchronized by binary and counting semaphores and mutexes (w/w.o. priority inheritance).
– Lists of ready/suspended/blocked tasks involve heap storage, pointers and type casts. Dynamic memory management uses `malloc()` and `free()`. User definable communication queues allow for thread-safe FIFOs.
– The stack-size has been preconfigured for each task.
– The kernel controls hardware registers, timers and listens to interrupts.

The FreeRTOS documentation also defines an API programmers may use to invoke the functionality provided by the kernel. The kernel seems to have been originally developed for a small system not supporting memory protection and is therefore an example of a previous generation of real-time operating systems. Memory protection has been added recently for the new Cortex-M3 processors but will not be discussed here. Modern micro-kernels as discussed in Sect. 3.6 are based upon memory protection schemes but usually have larger code sizes than the FreeRTOS kernel.

3 Verification methods and related work

3.1 What to prove?

Before embarking upon a detailed discussion on methods usable for verification of a real-time operating system, a sketch of some categories of properties to verify and some examples of those are given. The categories indicated below will be used while comparing various methods to verify the kernel. It should be emphasized that checking a real-time operating system (OS) for functional correctness only, is considered insufficient. Any real-time properties of such an OS should be specified and verified as well.

Functional correctness The implementation can be proved correct w.r.t. an abstract specification. Often, the specification only describes the functional properties of the system.

Problems of the implementation language such as:
- Pointer problems like null pointer dereferences and pointer aliasing,
- Memory allocation problems (e.g. memory leaks),
- Use of unspecified language issues such as expression evaluation order,
- Arithmetic problems such as overflow.

Timing properties show that a real-time system indeed schedules tasks 'on-time', e.g.:
- A task scheduled to run at moment t will run at moment $t \pm \Delta t$.
- Interrupt latency time: the time interval from the occurrence of an interrupt to the effective execution of the interrupt handling code should be less than some specified value.

As stated earlier, verifying these properties is outside the scope of this paper.

Safety properties (a.k.a. reachability properties) are used to prove the absence of bad events, e.g.:
- Is deadlock possible in the scheduling mechanism?
- Are queues indeed thread-safe?
- Is mutual exclusion guaranteed?

Liveness properties show that a good event will happen eventually, e.g.:
- A task having been scheduled will eventually be executed.

Fairness properties such as:
- A task scheduled infinitely often will be executed infinitely often.

The latter three categories will, somewhat incorrectly, be referred to as 'temporal specifications'.

In the sequel a discussion of various methods used for formal verification will be given. Section 3.2 discusses verification by means of refinement proofs and Sect. 3.3 will focus on state based verification. As these state based methods lead to so-called state explosion, various ideas to cope with that problem are described in Sect. 3.4. Another approach is translating existing kernel code into a more abstract description. This is the subject of Sect. 3.5. Section 3.6 will discuss some commercial RTOSes which claim to have been verified.

3.2 Refinement methods and proofs

Using the refinement approach, first an abstract specification of an RTOS is created. Then, a refined version (the concrete specification) is created. The two are related by a so-called *abstraction relation*. Proofs are needed to show that the concrete specification correctly represents the abstract one. Multiple refinement steps toward an executable language may be needed. Various tools are available to prove the system correct. Generally speaking, these proofs are performed by automatic theorem provers but steering these proofs requires considerable interactive user expertise. Three entries known to the author are:

The approach by Craig. In [14, 15] Craig describes in an extensive effort the full refinement of an RTOS using the Z notation [49]. He starts off by abstracting the μC/OS of Labrosse [31]. This kernel is similar to the FreeRTOS kernel in the sense that there is no distinction between kernel space and user space. Device drivers are also not discussed. The abstract specification is refined by formal reasoning in one or two steps into Z specifications that can, according to the author, be readily implemented. However, this seems too optimistic: all data types are (still) of type $\mathbb{N}$, there is no heap size checking and the total size of the level 2 spec is approximately one-sixth of the size of the FreeRTOS code. Checking the temporal specifications as in Sect. 3.1 seems infeasible. L. Freitas further detailed and corrected this work in [20].

The approach by Klein et al. In [30, 59] many authors describe a monumental effort to develop a round trip engineering project. They start with the C-code of the seL4-kernel, a high performance micro kernel and a further development of the L4 micro kernel [35], and transform the code into an abstract specification in Isabelle/HOL. The specification is then refined into an executable specification in Haskell [52]. In a second, manual, step the Haskell code is refined into a high performance C implementation. The theoretical background of the project is partly based upon separation logic [53, 54]. Particularly impressive is the almost complete handling of the C-language, including the areas where the language standard allows more than one interpretation. The effort is quite large; to verify 8700 lines of C code, more than 150,000 lines of proof script were needed. The whole project took some 30 man years. The authors consider the cost of their approach being less than the cost of an equivalent formal development according to industrial standards. The project concentrated on functional correctness and correct handling of C. Little could be found on proving temporal properties (see Sect. 3.1).

The approach by Déharbe et al. In [16] the authors describe the same subject as this paper: the GC6 FreeRTOS project. They analyze FreeRTOS by hand and construct specifications of the task and queue models. They use the B-method [1, 48] to construct a set-based abstract specification. The approach is similar to the one by Craig described above. Their initial specification leaves task priority undefined; in a later refinement they introduce priority in the system. For the system described, 49 interactive proofs are needed. The paper mentions difficulties foreseen when refining towards pointer constructs but is quiet about further refinements towards control of I/O-devices, interrupts and context switches. The paper clearly describes 'first steps' towards verifying FreeRTOS. It remains unclear from the paper whether the whole range of temporal proofs as described in Sect. 3.1 can be handled.

Refinement technology enables the user to refine operations and data structures into more 'detailed' ones. After a refinement step has produced a stateful (sub)-system, tools checking state properties such as model checkers may be used

to prove relevant temporal behaviour of the (sub)-system. A tool set having an integrated approach in this area is ProB [43].

However, refinements are usually done on indivivual operations and data structures. This requires reiterating the checking of temporal specifications after each refinement step. Little seems to be known about refinements having introduction rules spawning or refining temporally correct (sub)-systems.

3.3 State based methods

Contrasting with refinement based approaches, state based approaches (such as model checking) usually start from an existing implementation. Properties are being proved over the implementation. Such properties can be proved using classical model checkers or so-called SAT-solvers.

The implementation tested can be written in (C) code or in some abstraction of it, written in the modeling language of a model checker. In this paper, the Spin system [23] and the Promela language will be used; many other systems and languages do exist but will not be elaborated upon here due to lack of space.

Languages for model checking allow the programmer to (dynamically) create processes; for each process an automaton is constructed. During model checking, the model checker will probe all possible interleavings between the processes exposing implementation errors that cannot be found using regular testing approaches.

To enable proofs of required properties, such a modeling language is usually quite restricted in its data and control structures. Properties to be proved are given in a (temporal) logic (propositional logic, predicate logic, Linear Temporal Logic (LTL), Computation Tree Logic (CTL), Timed-CTL (TCLT))[2]. The model checker combines the automaton and the property specification and provides either a proof that the system indeed conforms to the specification or a counterexample showing a path leading to the error.

The model checker combines the automatons describing the processes into one automaton; the size of the resulting automaton being the product of the sizes of the constituting automatons; this is referred to as *state explosion*. Due to this problem, checking C code directly used to be next to impossible. Recent developments lessen this problem as indicated in various papers below.

3.4 Abstraction based approaches

Performing model checking directly on C-code looks like a interesting proposal. Unfortunately, the state space resulting from a direct translation of C-code into an automaton is likely to explode already on small examples due to recursion and pointer constructs. Various authors have tried to remedy that situation by restricting the search procedures and the power of the query language. Usually, these systems have a limited search depth and use heuristics to search only in

[2] The reader is assumed to be familiar with these logics. Otherwise, [3] will provide an excellent introduction.

areas where the programmer expects an error. Figure 1 shows how the search depth of a tree can be restricted to reduce the search space. Unfortunately, in software verification, error traces are often quite long and therefore a large bound is needed; otherwise errors may be missed.

One way to reduce the size of the verification model is by applying the technique of *predicate abstraction*. This so-called CEGAR (Counter Example Guided Abstraction Refinement) method was introduced by Clarke et al. [10] and heavily used in SLAM (see below). Predicate abstraction abstracts the data by keeping track of certain predicates on the data, leaving the control flow of the code unchanged. Each predicate in the concrete program is represented by a Boolean variable in the abstract program; the original data are eliminated. The abstract model and the (reachability) property are passed onto the model checker. If the property is proved correct on the abstract model, it is also correct on the concrete model. The abstract model has more behaviours than the concrete model. Therefore, a property shown to be false on the abstract model may be a spurious counterexample. A number of refinement steps may be needed to resolve the issue. In these approaches however, the query language is limited, e.g. only assertions and deadlock detection are supported and the expressivity of the programs is often reduced by disallowing e.g. threading and/or mutexes.

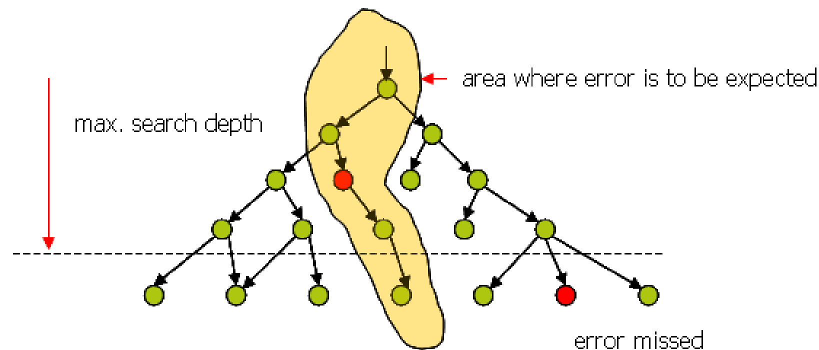

Fig. 1. Partial checking, limited search depth and specific search area

StEAM. The StEAM model checker is an assembly level model checker [34, 37]. C-code is translated into code for the IVM (Internet Virtual Machine) in a manner similar to the approach used by JPF (Java Path Finder) sketched in Sect. 3.5. According to both papers, deadlocks and assertion violations can be detected. A particular problem of the approach is that StEAM uses a proprietary system of threads and locks. The examples given are small sized: the philosophers, leader election and bounded buffer protocol.

CBMC. In Bounded Model Checking (BMC), the transition relation for a state machine and a property specification are jointly unwound to obtain a Boolean formula that is tested for satisfiability by using a SAT-solver. If the formula is satisfiable, a counterexample is extracted from the SAT output.

Clarke et al. [12] report on their development of CBMC (Bounded Model Checker for checking C programs). Their C sub-language supports pointers, dynamic memory allocation, recursion and the `float` and `double` data types. The tool is able to proof safety properties such as correctness of pointer constructs, array bound checking and checking of user provided assertions. There is no support for concurrency and interleaving. The paper is silent about the size of the programs the tool can handle.

In another paper [11], the authors discuss the behavioural consistency of C and Verilog programs. This paper is hardware verification oriented. An interesting feature in this paper is the translation of LTL formulae into C code. The C code is then merged with the program code and fed to the SAT solver.

Rabinovitz and Grumberg [44] extend the previous approach to handling threaded C programs. Their work is based upon CBMC [12] extended with threads and mutexes. The system allows dynamic generation of threads; the number of threads can be interactively increased until a bug is found or until the SAT solver explodes. In order to obtain results, the number of allowed context switches had to bounded. Using this technique, they can detect races and deadlocks invalidating safety properties. The size of the programs tested is quite limited; they report results on testing a concurrent implementation of bubble sort.

Schlich and Kowalewski [47] report on their use of the CMBC system for testing micro controller software. They discover two major problems: (a) Direct access to memory locations, as needed for I/O operations, gave rise to compiler errors. This problem could be solved by some rewrites of the C-code, but not in a generic way for a range of microprocessors. (b) The loop unrolling needed for CBMC-like testing did not handle infinite loops as are prevalent when an I/O device has been signaled and the driver is waiting for the device to respond. This problem is due to the lack of concurrency in CBMC. The authors decided to develop their own model checker [MC]SQUARE. This model checker checks the machine code; uses CTL and user provided asserts.

F-Soft. F-Soft [26] can be seen as a further development of the CBMC ideas. The tool has been used primarily for verification of sequential programs. It verifies reachability properties on labeled statements and programming errors such as array bound violations, null pointer dereferences, memory leaks etc. The tool optionally supports predicate abstraction.

The C program is translated into a finite state representation. Heap memory is modeled as a finite array; `malloc()` returns an index in the array. Two case studies are presented: a verification of a part of the PPP protocol and a verification of a TCAS component. The latter one consisted of the verification of a preprocessed C program of 1652 lines. The number of predicates needed for veri-

fication seems to have stressed the system. The authors remark that the absence of loop unrolling made F-Soft scale better than CBMC on larger programs.

SLAM The SLAM system (see [4] and references therein) builds upon the above mentioned CEGAR technology and Boolean abstraction of programs to deliver a tool for static device driver verification; in particular for Windows operating systems. SLAM focuses on stateful APIs: APIs requiring a certain temporal ordering of API function calls. E.g.: a file must be opened before it can be written into. A device driver to be tested for API conformance needs to be enclosed in an environment. Driver entry points should be called from a test harness mimicking the operating system. As drivers may call other (Windows-) functions, these must be replaced by stubs. SLAM is currently regularly being used on all Windows device drivers and reportedly has greatly contributed to the robustness of Windows.

3.5 Translation based approaches

As checking the C language gives rise to many problems, several authors have proposed translations to various, more abstract, languages.

Translation to Java. As the C-language and the Java-language have much in common, translating the kernel into Java seems a viable option. The resulting Java code can then be tested using the Java Pathfinder system (JPF) [18]. Difficulties regarding the translation of pointers and heap structures seem to disappear except for those places where void pointers are used to simulate genericity. However, mapping the concurrency mechanism of the RTOS to the Java thread model seems less intuitive. Inside JPF, the Java code is translated to code for a Virtual Machine enhanced for model checking. This VM only allows checking for invariants and deadlock to be performed on the Java program. There is no support for temporal specifications as discussed in 3.1.

Translation to Uppaal. The Uppaal model checker [55] has been designed to enable model checking of timed systems. The model checking algorithm is based upon TCTL, a Timed version of CTL (Computation Tree Logic) [3]. Although being one of the few model checkers being designed for timed systems, Uppaal seems to be incapable of checking the code of an RTOS because:

- A Uppaal program consists of state machines and communication channels. The state machines have to be programmed as nodes and transitions. Translating the kernel code to such machines cannot be done in a faithful way.
- The Uppaal system can only handle small programs.
- The query language of Uppaal does not allow checking of fairness properties.

Translation to Promela/Spin. Spin has a restricted facility to include C-code into a Promela program (Feaver, Modex). In [24], Holzmann, Groce and Joshi redesign that mechanism. They distinguish between *matched data* that is stored both on the search stack and in the state space of the model checker and *unmatched data* that is only stored on the stack. A large reduction in the amount of state space needed is obtained. However, the set-up influences the architecture of an application by placing state information in a contiguous area of memory. No examples using pointers are presented. Reportedly, the authors are able to model check programs that are about 10,000 lines of code.

Using this method to check the FreeRTOS kernel looks very promising. However, it is unclear how much *matched data* will be needed in the case at hand having many pointer operations. A distinctive advantage of this method is that all features of the Spin model checker (liveness, fairness, multi core operation and swarm verification) are available. A disadvantage is that the C code inserted in Promela is executed as an atomic block, disallowing interleaving processes. Checking FreeRTOS in this way is problematic but the next paragraph describes a possible solution to that.

In [60], Zaks and Joshi describe another redesign of the same mechanism. Starting at multi-threaded C code, a typed byte code for a virtual machine is generated. The Spin system controls the virtual machine, allowing most proving mechanisms provided by Spin to be used. The paper mainly discusses extending the partial order mechanism needed. The largest example given is a 2800 line C program implementing a multi threaded communication system. By allowing most of the temporal proof mechanisms, this approach is quite promising for verifying the FreeRTOS kernel except for the dependency upon the `pthreads` library which is not appropriate for FreeRTOS.

Gallardo et al. [21] describe an interesting approach using a two dimensional logic that combines time (control flow) and space (dynamic data structures). Control flow is specified using an extension of LTL; data is specified using CTL. Using the extended logic, they use Spin to model check properties of C programs, including dynamic allocation. By extending the checking of invariants in LTL, they are able to detect acceptance cycles. Unfortunately, not all temporal specifications can be tested and the examples given are rather limited in size (e.g.: list reversal).

3.6 Some verified commercial RTOSes

Apart from the safe companion to FreeRTOS, various other systems claim to have been formally verified. Presumably the list below is far from complete, but it is assumed that the systems mentioned will give a good impression of what is available commercially.

Green Hills Integrity RTOS, WindRiver and QNX. These OSes are examples of so-called micro kernels. They have been ported to many architectures and are being used in millions of real-time systems. They have been certified

according to many standards like IEC 61508 [25], FDA 510K [19], ISO/IEC 15408 [27] and DO-178B [45] to various levels. Many of these standards are process level standards however; they prescribe the process a developer should follow to create dependable software. Very few of these standards require the use of formal methods. An exception is ISO/IEC 15408 at level EAL6 and higher.

OpenComRTOS. This system [40] is advertised as a network centric RTOS with support for heterogeneous multiprocessors systems and multi core systems. The system has been ported to many architectures. Most of the C kernel code is generated and checked according to MISRA [38]. The kernel code is reportedly much smaller than the competitors' code. OpenComRTOS is suited for level 3 or 4 SIL applications. Full source code and formal application reports are available on special contract. The system has been partially verified using the TLA+/TLC system [33]. Papers on OpenComRTOS by Verhulst et al. are available in [50, 56]. The authors of those papers report that:

- The mathematical notation of TLA (Temporal Logic of Actions) urged them to work at a higher level of abstraction than e.g. the C-like syntax of Spin [23] would do, which they considered an advantage.
- TLA/TLC was used primarily as an architectural design tool.
- The TLC model checker declares every action as a critical section, disallowing the verification of concurrency present in the system. This resulted in an avoidance of global data structures in the architecture.
- Due to the state explosion problem, the authors were unable to do a complete check of the kernel; modeling was restricted to certain classes of services (Ports, Events, Services).

Unfortunately, the papers show little detail about which of the temporal specifications mentioned in Sect. 3.1 have indeed been checked.

PikeOS. In [42] the authors describe the verification of the PikeOS [7]; a further development of the L4 micro kernel. The PikeOS web-site states that the kernel is certifiable to safety-critical standards (DO-178B, IEC 61508 and EN 50128 [9]). The authors have used the Microsoft Verifying C Compiler which allows them to formulate contracts and invariants stored as annotations within the source code. So far, they seem to concentrate on the system call level; an example of a system call changing the priority of a thread is given. An interesting feature is the verification of assembly code and hardware semantics.

It remains a bit unclear from the paper how the approach would scale up towards checking the whole of the kernel. Given the tool used, it seems impossible to check for the temporal specifications given in Sect. 3.1.

Design rules Many commercial systems advertise themselves as being secure in some of several ways. Much of this security comes from the proper design of these systems. A number of good design rules as applied in these systems, is given below.

- The kernel certifies that upon initialization, all safety related attributes are set to a ”known good default” reducing chances of tampering with the kernel.
- The kernel guarantees a fixed budget of CPU time to a task. Similar guarantees can be stated for stack space and memory partitioning.
- All accesses to a resource are mediated by the kernel; every access goes through the same mechanism for verification.
- Resources are protected from implicit sharing; upon releasing a resource, a user has the option of explicit clearing of all the information.
- To ensure accountability, all events can be time stamped; recorded events cannot be repudiated with respect to their timing.
- The kernel uses some scheduling mechanism as Rate Monotonic Analysis or Earliest Deadline First Scheduling to guarantee throughput.

Using such good design principles contributes to the perceived safety of the system, but is different from formal proofs of correctness. In fact each of these rules opens up new possibilities for formally proving its implementation correct.

As shown above, many commercially available systems have been (partially) formally verified. Understanding their conformance to the temporal specifications given in Sect. 3.1 is not straightforward. Unfortunately, these operating systems are not open source and few of these verifications are in the public domain.

4 Further steps

Based upon all the material presented herein, it remains to present a suitable proposal for verifying the FreeRTOS-kernel. Basically, a choice can be made between the refinement based approaches and the state based approaches.

Regarding the first category it can be said that, as exemplified in this paper, much successful work has been done already.

In the category of state based systems, the CBMC-like approaches seem unsuited to handle the verification of an OS because these approaches have difficulties handling threads, concurrency and loop unrolling problems. An operating system basically consist of an infinite loop and in the case of FreeRTOS even user tasks must be programmed as infinite loops. In addition, CBMC-like systems are usually limited to proving safety properties.

The SLAM approach [4] requires special mentioning here. SLAM targets correctness of device drivers and in particular the temporal aspects of ordering of API-calls to a device driver. Some of the work of SLAM is valuable to the FreeRTOS-work, but the API-part seems less applicable here because FreeRTOS only has a single API-layer directly accessible from application programs. Checking such APIs involves checking user programs. A test harness containing such programs is currently unavailable but developing the SLAM technology for users of the FreeRTOS would be very fruitful. Such a test harness would however give more credentials to the correctness of the user program than to the correctness of the FreeRTOS itself.

Testing temporal specifications of an RTOS using a model checker as e.g. Spin suggests the following two routes:

- The *matched data/unmatched data* approach by Holzmann et al. [24], extended by the virtual machine of Zaks and Joshi [60].
- A hand translation from C code into Promela code.

The Holzmann/Zaks approach In this approach some problems are to be expected, e.g.:

- The dependency upon the `pthreads` library must be replaced by the more general mechanism of interrupts and context switching.
- Using this method, many probes must be inserted in the code. Finding a systematic method to find and insert probes is much needed. The paper by Andrade et al. [2] will provide some help here.
- Solving the problem of limited concurrency [24].

Translation by hand When C-code is hand-translated into Promela, several abstraction issues need resolution:

- When verifying FreeRTOS there seems no need to solve the general C to Promela translation problem as e.g. is done in [60] and there is no need to solve the general pointer problem. In the FreeRTOS kernel, pointers are mainly used to construct (doubly) linked lists of a few data types only. The storage model for the heap may be simulated with an array construct; pointers are replaced by indexes as in [26]. This will lead to a faithful transformation; C-code can be translated (by hand) in a one-to-one way into Promela. Preliminary experiments show a large state vector however. This is to be solved by using a special compression mechanism as in [51] in addition to the mechanisms built into Spin already. If state compression proves insufficiently effective, a less faithful model without indexes can be constructed.
- In fact, the kernel and all application programs form a sequential piece of code. Context switching only provides an illusion of parallelism. Real concurrency stems from I/O devices and interrupts. I/O and interrupts can be implemented using a very small state space, thus minimizing state explosion. As a further reduction, limiting the amount of context switches as in [44] can be tried.
- One particular problem is the difference in atomicity between Spin and machine instructions. In Spin, every statement is, by definition, atomic (non-interruptible). With machine code, interrupts can occur on every machine instruction; sometimes even inside a machine instruction. Because the Promela code can be interrupted before and after every instruction, sufficient interruptible locations are present and no difficulties are foreseen.
- In FreeRTOS, a context switch will save all registers on the stack and later restore the previous context. As registers are unavailable in a Spin simulation, it seems sufficient to reduce stack handling to handling return addresses.

– The number of Spin data types is quite restricted compared to the number of C data types. E.g.: C knows about the data types `char`, `signed char` and `unsigned char`. Promela only has a `byte` data type which is equivalent with `unsigned char`. Promela has no `long` data type either.
– As the code of the kernel has been programmed quite neatly, few problems are foreseen. Any problems have to be detected 'by hand', however. The rules given in [39] will be most helpful here.

The above approach has been used on a small scale as shown in a web report on the formalization of the Minix kernel [32].

Currently, both methods mentioned above seem to be the most fruitful ways forward and both implementations are at this moment being worked on.

5 Summary and further work

This paper has provided an overview on various methods to increase the confidence in the functional and temporal correctness of a real-time operating system. In particular, proving safety, liveness and timing properties of such an operating system is considered very important. Both refinement based and state based approaches have been discussed. In the latter area, two promising techniques, the Holzmann/Zaks approach and the direct translation of C to Spin have been selected for further investigation.

References

1. J. R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge Univ. Press, 1996.
2. W. L. Andrade, P. D. L. Machado, E. L. G. Alves, and D. R. Almeida. Test case generation of embedded real-time systems with interruptions for FreeRTOS. In *SBMF 2009*, volume 5902 of *LNCS*, pages 54–69. Springer-Verlag Berlin Heidelberg, 2009.
3. C. Baier and J.P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.
4. T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *CACM*, Dec 2009.
5. R. Barry. FreeRTOS reference manual. `http://www.freertos.org`.
6. R. Barry. Using the FreeRTOS real-time kernel, a practical guide. `http://www.freertos.org`.
7. C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Formal verification of a microkernel used in dependable software systems. In *SAFECOMP 2009*, volume 5775 of *LNCS*, pages 187–200. Springer-Verlag Berlin/Heidelberg, 2009.
8. J. C. Bicarregui, C. A. R. Hoare, and J. C. P. Woodcock. The verified software repository: a step towards the verifying compiler. *Formal Aspects of Computing*, 18(2):143–151, 2006.
9. CENELEC. A European standard for railway traffic. `http://de.wikipedia.org/wiki/EN_50128`.

10. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5), Sept. 2003.
11. E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 368–371, NY, USA, 2003. ACM.
12. E. M. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *LNCS, Volume 2988*, pages 168–176, 2004.
13. D. Cofer and M. Rangarajan. Formal verification of overhead accounting in an avionics RTOS. In *Proceedings of the 23rd IEEE Real-Time Symposium (RTSS'02)*. IEEE, 2002.
14. I. D. Craig. *Formal Models of Operating System Kernels*. Springer London, 2007.
15. I. D. Craig. *Formal Refinement for Operating System Kernels*. Springer London, 2007.
16. D. Déharbe, S. Galvão, and A.M. Moreira. Formalizing FreeRTOS: First steps. In *SMBF 2009*, volume 5902 of *LNCS*. Springer-Verlag Berlin/Heidelberg, 2009.
17. E. W. Dijkstra. Notes on structured programming. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, chapter 1, pages 1–82. Academic Press, 1972.
18. Java Path Finder. JPF. `http://babelfish.arc.nasa.gov/trac/jpf`.
19. U.S. Food and Drug Administration. Medical devices. `http://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/DeviceApprovalsandClearances/510kClearances/default.htm`.
20. L. Freitas. Mechanizing data-types for kernel design in Z. In *SBMF 2009*, volume 5902 of *LNCS*, page 186..203. Springer-Verlag Berlin/Heidelberg, 2009.
21. M. M. Gallardo, P. Merino, and D. Sanán. Model checking dynamic memory allocation in operating systems. *J. Autom. Reasoning*, 42:229–264, 2009.
22. T. Hoare and J. Misra. Verified software: theories, tools, experiments, July 2005. `http://vstte.ethz.ch`.
23. G. J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003.
24. G. J. Holzmann, R. Joshi, and A. Groce. Model driven code checking. *Autom. Softw. Eng*, pages 283–297, 2008.
25. IEC61508, Functional safety of electrical/electronic/programmable electronic safety-related systems. `www.iec.ch/61508`.
26. F. Invančić, I. Shlyakhter, M. K. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking C programs using F-Soft. In *Proceedings of the 2005 International Conference on Computer Design (ICCD'05)*. IEEE, 2005.
27. ISO15408. `http://en.wikipedia.org/wiki/Common_Criteria`.
28. C. Jones, P. O'Hearn, and J. Woodcock. Verified software: A grand challenge. *IEEE Computer: Software Technologies*, 39(4):93–95, April 2006.
29. G. Klein. Operating system verification - an overview. *Sādhanā*, 34(1):27–69, 2009.
30. G. Klein and et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symp/ on OS Principles*, pages 207–220. ACM, Oct 2009.
31. J.J. Labrosse. *MicroC/OS-II, The Real Time Kernel*. Miller Freeman Inc., 1999.
32. Lacuso. 2009casestudy1. `http://lab.cs.ru.nl/laquso/2009CaseStudy1`.
33. L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson Education Inc., 2002.
34. P. Leven, T. Mehler, and S. Edelkamp. Directed error detection in C++ with the assembly-level model checker StEAM. In *SPIN 2004*, volume 2989 of *LNCS*, pages 39–56. Springer-Verlag Berlin/Heidelberg, 2004.

35. J. Liedtke. On $\mu-$kernel construction. *SIGOPS, O.S. Review*, pages 237–250, Dec. 1995.
36. H. D. Macedo, P.G. Larsen, and J. Fitzgerald. Incremental development of a distributed real-time model of a cardiac pacing system using VDM. In *Proc. 15th Intl. Symposium on Formal Methods*, volume 5014 of *LNCS*. Springer-Verlag, 2008.
37. T. Mehler and P. Leven. Introduction to StEAM. http://steam.cs.uni-dortmund.de.
38. The MISRA Guidelines. http://www.misra.org.
39. M. Norrish. C formalised in HOL. Technical Report 453, University of Cambridge, Computer Laboratory, 1998.
40. OpenComRTOS. A scalable and network-centric RTOS for embedded systems. http://www.altreonic.com.
41. Pacemaker formal methods challenge. www.cas.mcmaster.ca/sqrl/.
42. PikeOS. http://www.pikeos.com/.
43. proB. http://www.stups.uni-duesseldorf.de/proB.
44. I. Rabinovitz and O. Grunberg. Bounded model checking of concurrent programs. In *CAV 2005*, volume 3576 of *LNCS*, pages 82–97. Springer-Verlag Berlin/Heidelberg, 2005.
45. RTCA. DO-178B, Software considerations in airborne systems and equipment certification. http://www.rtca.org/.
46. SafeRTOS. A safe RTOS. http://www.highintegritysystems.com.
47. B. Schlich and S. Kowalewski. Model checking C source code for embedded systems. *International Journal on Software Technology Transfer*, 11:187–202, 2009.
48. S. Schneider. *The B-Method: an introduction*. Palgrave, 2001.
49. J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
50. B.H.C. Sputh, O. Faust, E. Verhulst, V. Mezhuyev, and T. Tierens. Reliable performance for heterogeneous real-time systems with a small code size. http://www.altreonic.com/sites/default/files/Whitepaper_OpenComRTOS.pdf.
51. P. Taverne and C. (Kees) Pronk. RAFFS: Model checking a robust abstract flash file store. In *Formal Methods and Software Engineering, Proceedings of the 11th International Conference on Formal Engineering Methods ICFEM 2009*, volume 5885 of *LNCS*. Springer-Verlag Berlin/Heidelberg, 2010.
52. S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999.
53. H. Tuch. Formal verification of C systems code; structured types, separation logic and theorem proving. *Journal of Automated Reasoning*, 2-4:125–187, April 2009.
54. H. Tuch, G. Klein, and M. Norrish. Types, bytes and separation logic. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–108, 2007.
55. Uppaal. The Uppaal model checker. http://www.uppaal.org.
56. E. Verhulst and G. de Jong. OpenComRTOS: An ultra-small network centric embedded RTOS designed using formal modeling. In *SDL 2007*, volume 4745 of *LNCS*, pages 258–271. Springer-Verlag Berlin/Heidelberg, 2007.
57. Verified software repository. http://vsr.sourceforge.net.
58. Verified Software: Theories, Tools, Experiments. http://vstte.inf.ethz.ch/.
59. S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish. Mind the gap; a verification framework for low-level C. In *Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*, pages 500–515, 2009.
60. A. Zaks and R. Joshi. Verifying multi-threaded C programs with SPIN. In *Proc. 15th Spin workshop*, 2009.

TUD-SERG-2010-042
ISSN 1872-5392