Towards Proving a Real-Time Operating System Kernel Formally Correct

Grzegorz Hannula * Wolfgang A. Halang **

* AGH University of Science and Technology, 30-059 Kraków, Poland
  (e-mail: gh@ia.agh.edu.pl)
** Fernunierversität, 58084 Hagen, Germany
  (e-mail: wolfgang.halang@fernuni-hagen.de)

Abstract: On-going work aiming to prove the correctness of a real-time operating system kernel is described. The kernel runs on a platform providing a separate co-processor for it, and was derived by a simplifying re-design of an earlier developed one. The main design decisions taken are outlined with special regard to the scheduling policy employed. The requirements elicited during design specification were grouped by their origins. Particularly important among them is that the kernel is to support programs written in an extended version of the real-time programming language PEARL. The selection of a formal method suitable for the envisaged purpose is discussed. It is outlined how the kernel is modeled in the specification language of PVS, and how the requirements defining its correctness are formalised.

Keywords: Real-time operating system kernel, formal correctness proof, kernel co-processor, non-pre-emptive earliest deadline first scheduling, PEARL, PVS.

1. INTRODUCTION

The growing complexity of computer-controlled systems, especially of those used in hard real-time environments, does not only cause new classes of problems, but is also challenging scientists and practitioners. Among these challenges, the problem of such systems' correctness appears to be the most important one. The application of mathematical methods, often called formal methods, in the areas of specification and verification of software and hardware systems has reached a level of maturity that it has become common practice in verifying safety-critical systems.

In this paper, we are focusing our attention on formally verifying the correctness of a real-time operating system kernel and its hardware platform, developed on the basis of the one presented in Halang and Stoćenko (1991). Just as the latter, the re-designed kernel is to be used in hard real-time environments. In other words, the kernel belongs to the category of hard-real time systems, and as such it should meet several requirements to be discussed. Correctness of the kernel means that all these requirements are met, which is to be shown using formal methods.

Before introducing the execution model, we sketch the kernel's structure. One of the most important features of its hardware architecture Halang and Stoćenko (1991) is physical separation of application programs from activities performed by the operating system. This separation involves using two hardware units, maintaining a master-slave relationship as shown in Fig. 1. The application programs are executed on a general-purpose Task Processor acting as slaves, whereas the operating system itself uses a dedicated Kernel Co-processor acting as master. As schematically shown in Fig. 2, the architecture is composed of a number of co-operating functional blocks, implemented both in hardware and firmware.
From the programmers' perspective, application programs to be executed under control of this kernel are written in a modified version of the real-time programming language PEARL DIN66253-2 (1998), in which concurrency is modeled using the task concept. Tasks are regarded as mutually independent. All task-related operations are directly expressed by PEARL language statements. Execution of tasks can be started at certain time instants, in reaction to external events, or can be triggered by interrupts. The kernel was designed to support the rather demanding execution of PEARL programs.

Proper scheduling of task executions on the task processor is crucial, especially with respect to timing. Therefore, an optimal scheduling principle was chosen Halang and Stoyenko (1991), viz., Earliest Deadline First (EDF). It can be applied provided that both execution times and deadlines of all tasks are known a priori. Task execution times may be supplied by the programmers as part of the task declarations, or be estimated by a schedulability analyser during compilation. Task deadlines are to be given explicitly in their declaration statements. Since the deadlines of all tasks scheduled for execution are checked upon any context switch, any possible violation of a task deadline can be detected ahead of its planned execution.

In our approach, non-pre-emptive EDF scheduling as outlined in Section 2 is employed for its simplifying effects on task sequencing and synchronisation resulting in easier correctness proofs. The syntax and semantics of the PEARL extensions proposed in Halang and Stoyenko (1991) concerning task synchronisation and communication were also modified to the end of facilitating verification, which is described in Section 3. This gives rise to the re-design of the execution platform's architecture and the kernel to be outlined in Section 4.

2. NON-PRE-EMPTIVE EDF SCHEDULING

The considerations in Halang (1992) reveal that EDF is the best scheduling policy for general multitasking, provided that tasks are pre-emptable at any arbitrary point in time. Unfortunately, this precondition is not very realistic, since it is only fulfilled by pure computing tasks fully independent upon one another. In general, however, tasks have resource requirements and, therefore, execute critical regions to lock peripherals and other resources for exclusive and uninterrupted access. When a task is pre-empted in a critical region, the pre-emption may cause a number of problems and additional overhead, such as the possibility of deadlocks or the necessity for a further context switch when the pre-empting task tries to gain access to a resource being locked. Therefore, here we employ the following modification of the EDF discipline proposed in Halang (1992), which allows tasks always to run to completion. Inherently, the algorithm prevents resource access conflicts and deadlocks, and eases synchronisation.

Algorithm NP-EDF: Schedule tasks earliest deadline first, unless this calls for pre-emption of a running task.

Upon a dynamic arrival of a ready task, with the following schedulability condition the task's response time can be guaranteed — or a future overload can be detected and handled by graceful degradation. The cost of checking a task set's feasible schedulability is linear in the number of ready tasks, and the check itself is trivial, i.e., the operating system is enabled to supervise the observance of the timeliness condition fundamental for real-time systems.

Schedulability condition: For any time $t$, $0 \leq t < \infty$, and any task $T$ with deadline $t_z > t$, let

$$a(t) = t_z - t \text{ is its response time, and}$$

$l(t)$ its execution time required before completion.

If at time $t$ a task joins the ready task set $R(t)$, which is scheduled according to Algorithm NP-EDF, and if the task's deadline is earlier than the one of the executing task $T_j$, $j \geq 2$, with the ready task set being indexed according to increasing deadlines, then all tasks in $R(t)$ will meet their deadlines if the inequalities

$$a_i(t) = \sum_{k=1}^{j-1} \delta_{ik} + \sum_{k=1}^{i} l_k(t), \quad i = 1, ..., n = |R(t)|,$$

hold. When the newly arrived task's deadline is not earlier than that of the executing task, then the necessary and sufficient condition for feasible schedulability is

$$a_i \geq \sum_{k=1}^{i} l_k, \quad i = 1, ..., n.$$

3. TASK ADMINISTRATION IN EXTENDED PEARL

The state diagram of a task shown in Fig. 4 illustrates the state transitions either caused by execution of PEARL statements, operating system actions, or external events. Initially, all tasks are not scheduled. Upon execution of an ACTIVATE statement, a task changes its state to scheduled. When its optional start condition is fulfilled,
the task joins the queue of ready tasks. The start condition specifies how the task is to be started, either at a certain instant of time, or after some delay, or possibly delayed after occurrence of an interrupt. Variants of start conditions allow to request repeated task executions with constant periods during specified time frames. A task can temporarily leave the state running after execution of LOCK or AWAKE statements. It can also be explicitly suspended and later continued using the SUSPEND and CONTINUE statements. To be applied on the invoking task itself, the latter operations are combined in the RESUME statement. The resumption and continuation statements contain simplified start condition not allowing for repetitions. A start condition can be invalidated by executing a PREVENT statement. This operation does not affect a task currently running, but cancels all its further activations or continuations. A task which finished its execution either by reaching the END statement or by being explicitly terminated using the TERMINATE statement enters the state terminated, causing implicit transition to the state not scheduled.

To replace PEARL's unstructured and highly unsafe constructs semaphores and bolts for task synchronisation and communication, in Halang and Stoelenko (1991) the structured LOCK statement was introduced to express the functionality of critical regions. Task communication was based on a mechanism of supervised shared variables and the AWAKE statement using the latter. Both LOCK and AWAKE statements may not be nested, and both comprise timeout clauses to bound execution time.

![Control flow of LOCK statement](image)

**Fig. 5. Control flow of LOCK statement**

Variables used for synchronisation are to be declared with the CONTROLLED_ACCESS attribute. They can be declared as simple variables of basic types, as arrays, or as structures, and are accessible within LOCK statements, only. The operating system stores the status of each controlled variable, which can be obtained at the program level using the operator CONTROLLED_VAR_STATE. The value returned by this operator distinguishes between three possible situations, viz., not occupied, exclusive access, or shared read-only access. The control flow of the LOCK statement is shown in Fig. 5. It can be observed that either a task gains access to a resource and executes its critical region, or the timeout expires. In the latter case, the timeout reaction statements are executed. The amount of time a task can execute its critical region, in other words the blocking time, is limited. Violation of this time limit by a task results in leaving the critical region, and in raising an exception signal by the operating system.

![Supervised shared variable table](image)

**Fig. 6. Supervised shared variables**

Supervised shared variables are shared memory objects, identified by indices and represented schematically as in Fig. 6. It is possible to provide a function, which checks whether a supervised shared variable fulfills certain conditions. If such a function is available, then it is called by the operating system during any write operation performed on this variable. To access and control supervised shared variables, functions for opening and closing them, for reading their values and states, for setting their values, and for querying conditions, respectively, are provided.
Fig. 7. Communication via supervised shared variables

The AWAKE statement of the PEARL extension suspends the task surrounding it until one of a set of specified events occurs. These events can be occurrences of interrupts or signals, or the fulfillments of conditions involving task states and shared variables. Thanks to the latter, previously suspended tasks waiting for the fulfillment of conditions can be informed and their execution continued. Fig. 7 illustrates this approach. The waiting time in an AWAKE environment is bounded by a timeout clause, and time limits for executing the event reactions indicated are provided, too. Similarly to the LOCK statement, a violation of the time limits causes an exception signal to be generated by the operating system.

Finally, the standard input/output statements of PEARL addressing data of type BASIC were extended by timing options. Thus, sensor data can be read and actuator data written precisely at specified instants.

4. ARCHITECTURE OF THE OS KERNEL

The real-time operating system kernel of Halang and Stoeyenko (1991) and shown in Fig. 2 was based on the idea of designing a customised hardware platform to execute tasks formulated in the language PEARL under EDF scheduling. In this model it is assumed that real-time applications are composed of sets of independent, cooperating tasks, having finite execution times. The real-time programming language PEARL was selected to develop application programs for three main reasons:

1. the existence of any operating system is hidden,
2. the task concept is supported, and
3. its syntax allows for direct representation of time in task control statements.

The third reason is very important, because Earliest Deadline First scheduling requires information concerning a task’s timing, especially its deadline serving as priority.

For direct support of the execution of PEARL tasks the asymmetric two-processor architecture described in Halang and Stoeyenko (1991) was devised, which minimises the context switches by physical separation of task executions from actions performed by the operating system.

For our purposes of easier verification, this architecture was re-designed as diagrammatically shown in Fig. 8. An application composed of PEARL tasks is first loaded from a hard or flash disk drive to TASK PROCESSOR MEMORY, and then run by the TASK PROCESSOR. After necessary initialisations of code, data, and stack segments of each task, a runtime library routine fills the task control block and other data structures of the kernel, located in SHARED MEMORY and, by design, accessible by both the kernel hardware and the TASK PROCESSOR. Then, start of the first task possessing the attribute MAIN is requested by sending an implicit system call to the kernel’s SUPERVISOR block. The latter controls the entire unit including operations performed by the TASK PROCESSOR, and performs fault handling at the kernel level. The SUPERVISOR block directs incoming implicit system calls to appropriate functional blocks responsible for their implementation. Blocks involved in the realisation of a call perform their operations autonomously, communicating with one another without participation of the SUPERVISOR. The data structures used by functional blocks of the kernel for communication purposes were placed in the INTERNAL KERNEL MEMORY block.

5. REQUIREMENTS

The design decisions with respect to tasking model, scheduling policy, programming language features, and architecture of the execution platform as outlined so far gave rise to a specification expressed as a long list of requirements, which can be divided into the four groups mentioned below. To prove the kernel’s correctness, it must be shown that these requirements are met.

1. The most important and mandatory group of requirements originates from the definition of hard real-time systems, i.e., that they respond to any external event within a guaranteed amount of time.
2. Conditions to be fulfilled by an operating system implementing the tasking operations of PEARL.
3. Timing requirements of the new statements LOCK, AWAKE, and timed versions of sensor/actuator input/output statements TAKE and SEND as described in Halang and Stoeyenko (1991) as PEARL extensions. The first one was introduced for timed synchronisation of resource access, the second one for allowing a task to react to several external events, and third and fourth ones are standard PEARL instructions with extended functionality, respectively.

4. Requirements concerning identification, occurrence, and handling of faults as formulated during the informal specification of the kernel.
6. FORMAL METHODS

To show correctness, we want apply mathematical techniques known as “formal methods”. Usually, this consists of two basic consecutive steps, viz., formal specification and formal verification. When properly applied, a formal specification can result in a very precise and trustworthy description. In case of hard real-time systems, techniques based on testing are not sufficient means to assure correctness. Application of formal methods for such systems is, therefore, strongly advised. Formal verification in general is, however, commonly regarded as a difficult task for several reasons. First of all, besides experience in the area of formal methods, both knowledge and understanding of the problem domain are necessary. Identification of relevant properties is the most frequently mentioned example of the latter. What is more, they are no general techniques to find the safety-critical components of a system. Another frequently mentioned reason is the system complexity to be dealt with. Since consideration of all possible behaviours is an essential part of formal verification, the complexity leads to state space explosion, rendering verification very hard or sometimes even impossible. One common way to cope with this problem is down-scaling of a considered system’s mathematical model, viz., to make its state space finite, which, unfortunately, does not guarantee that the properties of such a down-scaled model will hold for the actual system. Selection of proper formal methods and tools is another source of difficulties, since they are more than 70 methods currently in use and, as a consequence, they are many popular tools supporting formal methods available. The various formal methods differ in their respective areas of applicability. Moreover, formal methods need to be chosen in a proper way to be useful to cope with a particular problem. Software tools supporting formal methods are not perfect, yet. Both, inadequate tools and poor tool support, has been reported. Application of formal methods may sometimes lead to erroneous results or flawed proofs. Correction of the latter requires much effort, and frequently focuses on mathematical aspects of the problem considered. Sometimes properties must be reformulated in order to be provable mathematically, e.g., state space invariants, which must be inductive to be usable in proofs based on the induction principle. It is worth noticing that the usage of formal methods does not automatically guarantee correctness, and that the correctness of a model by no means implies the correctness of the real system it models.

Formal methods are accepted tools to verify designs of hardware such as microprocessors, caches, or computer buses. Here, techniques based on model-checking are broadly used. On the other hand, formal methods are commonly being used in selected phases of the software development lifecycle, viz., specification, formalisation of requirements, or verification of critical system parts like communication protocols. Among several applications of formal methods in the area of real-time systems, there are some which are very interesting from our point of view. In his contribution on a formal model for partitioning the resources of an avionics computer, Di Vito (1998) formalised the proper partitioning, constructed a formal model and, finally, presented proofs. Dutere focused on the properties of a real-time scheduler. The latter was modeled as a state machine, and its execution traces were examined. Lensink et al. presented a formal model of a smartcard personalisation machine. The model was constructed in such a way that it could be used for simulations, too. The properties of the machine’s scheduling protocol were expressed as invariants and proven using a theorem prover. Finally, Fowler and Wells formally analysed the specification of a real-time kernel which is to support the set of static Ada 95 tasks running on a library level. For this purpose, the following elements were modeled as state machines using the Prototype Verification System (PVS) documented by Oware et al. (1999): the kernel, its interface, protected objects, and program. As an example, the property of mutual exclusion of tasks accessing protected objects was formulated as theorem. Problems concerning proof construction were discussed.

![Fig. 9. Formal verification related to other methods of software/hardware engineering](image)

Formal methods are usually accompanied by traditional software engineering methods, especially graphical ones appropriate to describe the most important features of the systems designed. Furthermore, “formal methods complement, but do not replace testing and other traditional quality control and assurance activities” Crow et al. (1997). Following these guidelines, we decided to organise the work of showing our kernel’s correctness in the way depicted in Fig. 9. Here, the design specification focuses on four main elements, viz., processing of implicit system calls, identification of kernel and task states, identification of faults, and description of communication between kernel modules, including external events. As it can be seen, the design specification forms the basis for both formal specification and testing, which can be performed in parallel. Here again, we followed an advice taken from IPL, saying that “if tests are not developed from a specification, then it is not testing”. When the design specification had been finished, we knew what kind of formal methods’ tool was required. We needed a general-purpose tool, based on higher-order logic and providing the user with arrays, records, lists, and abstract data types, supporting both theorem proving and model checking. We decided to choose PVS due to its generality and popularity Di Vito (1998); Dutere; de Groote and Hooman; Mokkedee et al. (2000).

The specification language of PVS is in some aspects similar to algorithmic programming languages. Among the reserved words of PVS are words both known from programming languages and from mathematics. The current version of PVS contains the module PVSio, especially writ-
ten to support animation, simulation, and testing of PVS specifications Muñoz (2005). Equivalents of loops, basic types, and variables are provided, so that PVS functions and predicates can be evaluated by iteration. It may be useful to check whether certain properties expressed as PVS predicates hold. Especially useful are input/output functions of PVSIO, to be employed in the same way as the ones of algorithmic languages. The results of any performed operation can directly be stored in text files.

The typical application of formal methods consists of four consecutive steps Crow et al. (1997), viz., characterization, modeling, specification, and analysis (cp. Fig. 10). In this context, characterization means to decide on levels of detail, data structures, and other relevant elements such as representation of time to be made on the basis of informal specifications. Advice taken from Crow et al. (1997), saying that “it is desirable to have a formal specification that very closely resembles the informal document”, was in our case rather easy to follow, since arrays, records, lists, and datatypes are directly represented in PVS. The latter were used to describe implicit system calls. We decided to represent time by natural numbers, an approach very close to the internal representation of time applied by operating systems and based on counting a hardware clock’s ticks.

![Fig. 10. Application of formal methods](image)

Formal models of computer systems are based on discrete mathematics. The execution behaviour of hardware or software is usually described using the concept of abstract state machines. Such a model consists of current state, identified by state variables, sets of inputs and outputs, and a transition function which calculates the next state. For our purpose, we decided to use an extension of this model by Shaw, known as communicating real-time state machine \( M \), as it takes communication with other machines and passage of time into account. At the level of abstraction we need, the kernel can be regarded as a parallel composition of such machines, denoted as \( M_1 \parallel M_2 \parallel \cdots \parallel M_n \), exchanging messages via communication channels. When properly constructed, such a model can serve as a basis for further examinations using both model-checking and theorem-proving techniques.

7. CONCLUSION

The paper described the first phase of a project aiming to prove the correctness of a real-time operating system kernel to be employed in safety-critical applications. The design decisions with respect to tasking model, scheduling policy, support for an extended version of the real-time programming language PEARL, and architecture of the execution platform featuring a separate co-processor for the kernel were reported, which were made to the end of reducing complexity and to render the project likely to reach its goal. Both architecture and kernel resulted from a simplifying re-design of an earlier concept. The selection of PVS as a formal method suitable for the envisaged purpose was discussed. Presently, the requirements elicited are being formulated in the specification language of PVS. The next, and final, step will then be the formal correctness proof with the help of the PVS tool suite.

REFERENCES


A. C. Shaw. A (more) Formal Definition of Communicating Real-Time State Machines.