The International Workshop
on Discrete-Event System Design, DESDes'01,
June 27÷29, 2001; Przytok near Zielona Gora, Poland

CORRECTNESS PROOF OF AN OPERATING SYSTEM KERNEL FOR HARD REAL TIME COMPUTING

Grzegorz HAMUDA, Wolfgang HALANG

Akademia Górniczo-Hutnicza w Krakowie, al.Mickiewicza 30, 30-074 Kraków,
POLAND, *gha@ia.agh.edu.pl*

FernUniversität Hagen, Faculty of Electrical Engineering, 58084 Hagen, GERMANY,
wolfgang.halang@fernuni-hagen.de

***Abstract**. An architecture (including both hardware and software solutions) for an operating system kernel to be employed in hard real time environments is shortly described, and its functionality is presented. By considering its application areas, which comprise safety critical systems, the need for correctness of such a kernel is pointed out. Ways to achieve this property are identified in the context of appropriate correctness criteria. It is discussed how proper formal methods are selected for verification, and to which particular task each method is applicable. Experiences and observations are presented. As one of the latter, the need to apply both theoretical (formal) and practical methods is underlined. Therefore, a simulator for the kernel was developed, whose functionality is described as well.*

***Key Words**. Hard real time computing, operating system kernel, correctness, formal methods, proofs, simulation*

1. THE ARCHITECTURE

The architecture of an operating system kernel considered here (see [2] for details) can be characterised by an asymmetrical dual processor configuration: applications consisting of independent, co-operating tasks execute on a general purpose processor (Slave), whereas all OS kernel activities are executed on its own dedicated co-processor (Master). The latter acts as supervisor of all activities of the entire system, including execution of user tasks, scheduling, time management, memory management, interrupt and input/output event servicing.

Task scheduling is based on the earliest deadline first algorithm. This algorithm can be employed, because estimations of the execution times of all tasks are known a priori. The task processor always executes the task having the shortest deadline. Unnecessary context switches are thus avoided. What is more important, however, is that the execution of system calls does not cause task pre-emptions (with the exception of activation or continuation of tasks with the shorter deadlines).

As shown on Fig. 1, the kernel is divided into three co-operating layers. The task interface to the OS Secondary Reaction Layer is organised in the form of system calls, which can be

divided into the different groups of tasking operations (activate, terminate, prevent, suspend, continue, resume, end), task scheduling (scheduler), task synchronisation (sync_test, sync_resume), task communication, and input/output operations.

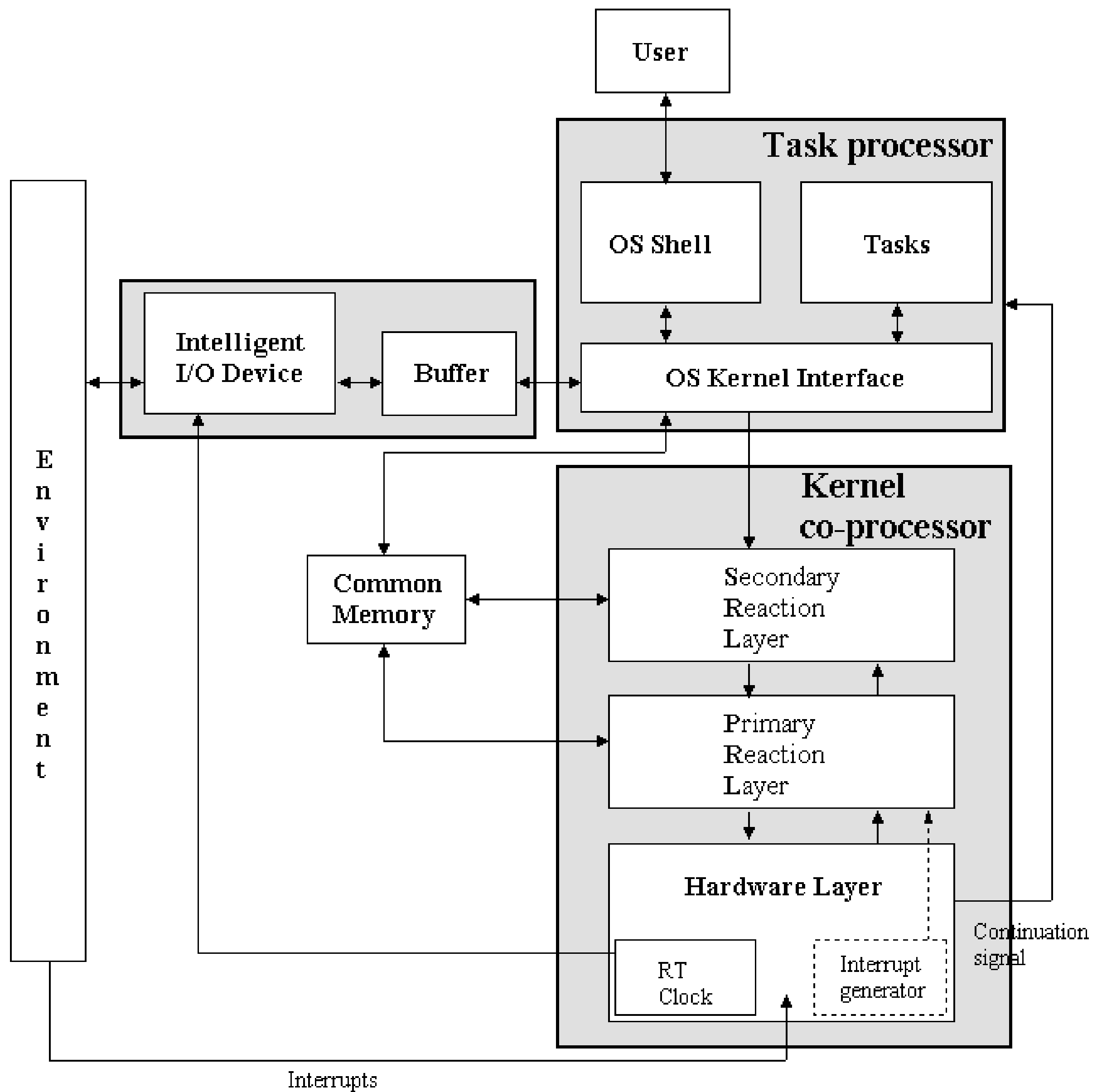

Fig. 1. Hardware architecture - functional diagram.

System calls concerning tasks, which are received by the Secondary Reaction Layer, contain time (or event) conditions for actions to be performed. The actions are stored in one of the system tables, while the calls are passed to the Primary Reaction Layer, which notifies the Secondary Reaction Layer on the occurrence of the corresponding trigger events, such as the passage of a duration or an external interrupt (Fig. 2). Such events trigger the execution of the associated actions. The OS kernel was specified, mainly in the language PEARL, in the form of about 28 algorithms.

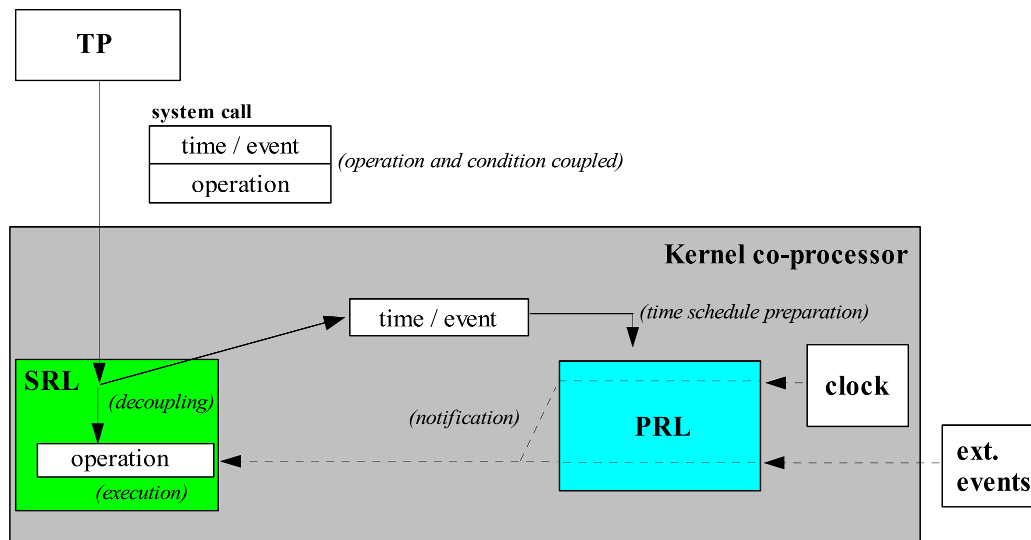

Fig. 2. System call.

2. THE PROBLEM OF CORRECTNESS

The kernel was designed to be used in hard real time environments. As it is broadly known, control systems for such environments may not crash or behave in a non-predictive manner. This concerns the control application itself (the code), the hardware it is running on, the hardware it controls, and the operating system as well. To be able to show that the proposed kernel can be used for such purposes, an attempt was made to proof - by the use of formal methods- that the kernel is reliable and dependable.

There is no universal prescription for the usage of formal methods in practice. The most useful advice and guidelines were found in [3]. There, the following phases of formal verification were mentioned (Fig. 3):

- Characterisation - that is to achieve a deep understanding of the application (and its domain area) to be verified
- Modeling - selection of a proper mathematical representation(s) (model) most suitable for the application, selection of a (formal) specification language and of appropriate tools (theorem prover, proof checker, model checker)
- Specification - decisions concerning the specification strategy (hierarchical levels, language, properties), writing the formal specification
- Analysis - interpretation of the specification prepared, proving the key properties etc.

From this short description at least four conclusions, having significance in practice, can be drawn:

1. Detailed (informal) specification of a system is very helpful (mainly in the characterisation phase),
2. Decisions concerning the selection of both a formal (mathematical) model and of tool(s) for verification purposes should be made in such a way that they fit to the system to be verified as closely as possible, and not the other way around,
3. System properties to be proven should be named (first, they should be discovered during the characterisation phase),

4. The selection of the formal method tool is (more or less) a function of the model selected and the system properties.

The formal verification of the kernel properties was based on the just described conclusions,

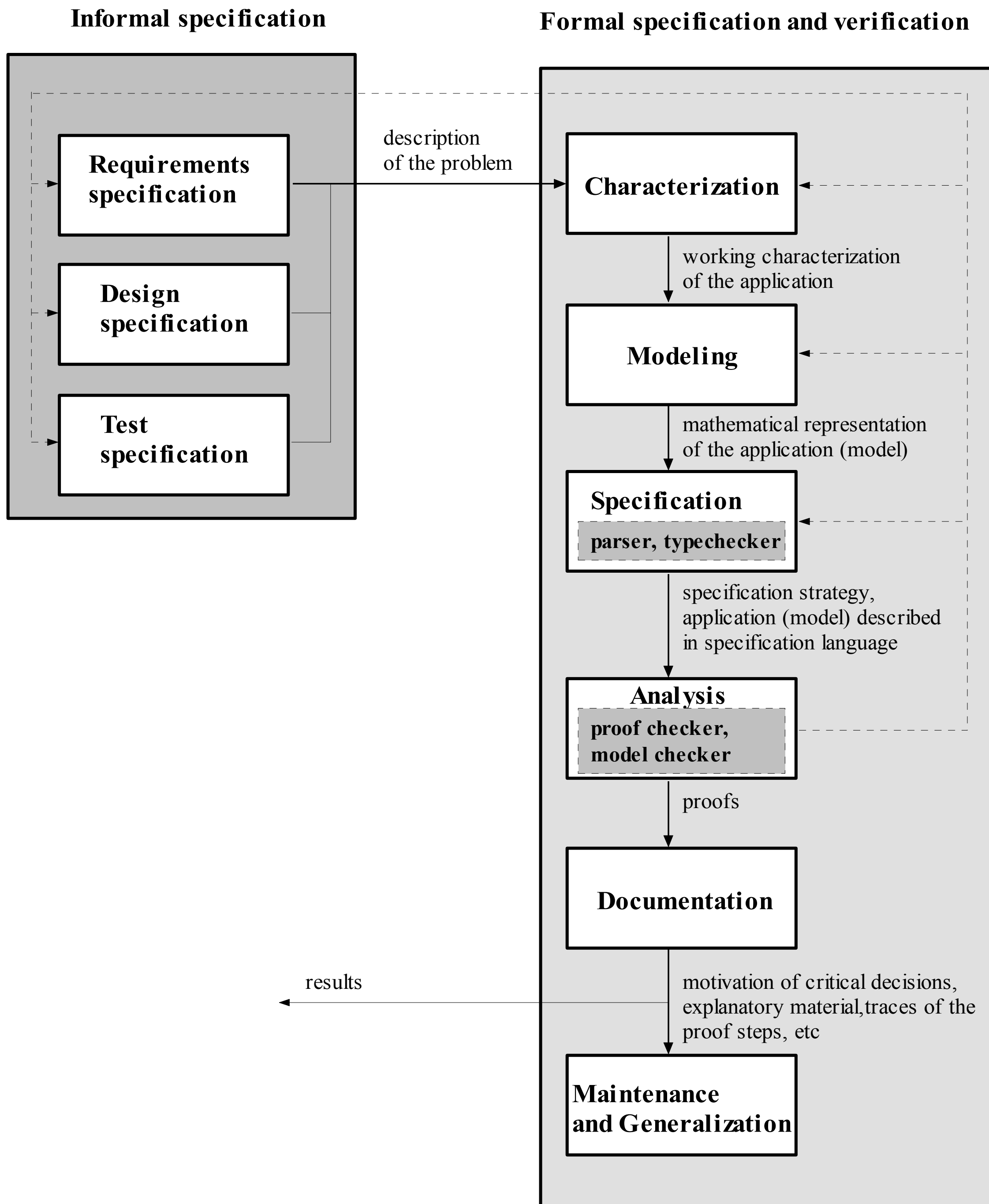

Fig. 3. Phases of formal verification.

treated as general directions. During the characterisation phase, more than 20 system properties were identified, the most fundamental on among them being:

The deadlines of all tasks listed as ready are met (normal situation), or the deadlines cannot be met (this fact is known a priori) and an overload signal is raised.

As another result of this phase of work, the following observations were made:

- Proving correctness of the hardware elements of the kernel would require the usage of a functional description.
- In case of the system calls (specified as algorithms) Hoare Logic is needed to show, for example, that the algorithms properly manipulate the system data structures.
- Some system properties will be having the form of theorems and axioms, while the others will be expressed as invariants.
- Hardware and software (of the kernel) cannot be analysed separately.

The mentioned observations have shown the need to select both a (formal) method and tools carefully. The latter should be able to be used both as theorem provers and model checkers.

There are many popular software tools supporting different formal methods. We have finally chosen the Prototype Verification System (PVS). It is a general purpose tool based on higher order logic. In PVS, the user is required to describe system properties to be proven in a specification language similar to a programming language. Thanks to its popularity and generality, many formal methods were successfully incorporated into PVS. The experience from the current stage of the work (specification of all system properties in PVS) confirms the decisions made earlier.

3. THE KERNEL SIMULATOR

Since the OS kernel was specified as a set of algorithms, there was a need to simulate it. It is common practice that teams applying formal methods build simulators to achieve deeper understanding of the functionalities to be verified. Our simulator is composed of several co-operating programs written in Java. The communication model is based on Remote Method Invocation (i.e., the client/server model). The kernel itself forms one program (Simulator), in which threads were used to model both the task processor and the kernel co-processor activities. To both the memory modules and fifos, which should be accessible for different kernel components, exclusive access can be ensured thanks to a thread synchronisation mechanism built in Java. This program acts as a (RMI) server, providing the other program (Monitor) with all the data needed for on-line system analysis. The third program (Control Panel) was designed to both enable control over the execution of the OS kernel (which can be started, stopped, re-run, executed step by step etc.) and emulation of the external environment. The monitor program mentioned above also acts as a (RMI) server for programs performing different kinds of visualisation. The basic one is Message Sequence Chart (MSC) Trace of the whole system. The program animates the diagram showing the exchange of data between elements of the kernel. Other visualisation tools can then be added easily. Each of the mentioned visualisation tools can be started or stopped independently of the simulator.

4. CONCLUSIONS

The usage of formal methods in practice requires planning and decision making. Important steps and decisions were shown for the example of a real time operating system kernel. The role of the traditional (informal) specification was underlined. The traditional methods (state charts, transition graphs, block diagrams etc.) turned out to be very useful, and form a sound basis for the formal verification of correctness. Both the PVS specification language and the tool PVS as a formal method served our purpose very well.

REFERENCES

[1] Myla Archer and Constance Heitmeyer *Human-Style Theorem Proving Using PVS* Naval Research Laboratory, Washington, DC 20375

[2] Wolfgang A. Halang and Alexander D. Stoyenko *Constructing Predictable Real Time Systems* Boston: Kluwer Academic Publishers 1991

[3] Sandeep Kulkarni, John Rushby, Natarajan Shankar *Formal Methods Specification and Verification Handbook for Software and Computer Systems Volume I: Planning and Technology Insertion Volume II: A Practitioner's Companion* Office of Safety and Mission Assurance, NASA-GB-002-95, Release 1.0

[4] Sam Owre, Natarajan Shankar, M. K. Srivas *A Tutorial on Using PVS for Hardware Verification* SRI International, Computer Science Laboratory, Menlo Park CA 94025 USA

[5] Sam Owre, Natarajan Shankar, J. M. Rushby *PVS Language Reference (Version 2.3)* SRI International, Computer Science Laboratory, Menlo Park CA 94025 USA

[6] Sam Owre, Natarajan Shankar, J. M. Rushby *PVS Prover Guide (Version 2.3)* SRI International, Computer Science Laboratory, Menlo Park CA 94025 USA

[7] John Rushby *Formal methods and digital systems validation for airborne systems,* NASA Contractor Report 1673, August 1995

[8] C. J. Walter, R. M. Kieckhafer, A. M. Finn *MAFT: A multicomputer architecture for fault-tolerance in real-time control systems,* IEEE Real-Time Systems Symposium, December 1985