Temporal analysis of a microkernel

This item was submitted to Loughborough University’s Institutional Repository by the/an author.

Citation: HUSSAK, W., 1995. Temporal analysis of a microkernel. Software Engineering Journal, 10 (1), pp. 21-26

Additional Information:

- This article was published online in the journal, Software Engineering Journal (© IEEE). It is also available at: http://ieeexplore.ieee.org. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Metadata Record: https://dspace.lboro.ac.uk/2134/4168

Version: Published

Publisher: © IEEE

Please cite the published version.
This item was submitted to Loughborough’s Institutional Repository (https://dspace.lboro.ac.uk/) by the author and is made available under the following Creative Commons Licence conditions.

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/
Temporal analysis of a microkernel

by Walter Hussak

Temporal logic techniques have been proposed as a way of achieving a very natural transition from informal requirements to a formal specification of the requirements. The paper presents a case study of a real-life system developed using such techniques. Both a top-level specification and implementation semantics are given in temporal logic. In particular, the progression from statements in English to temporal logic is highlighted. A correctness proof that the implemented system satisfies the specification has been produced.

1 Introduction

In the formal development of any system, an important issue is the clarity of the formalisms used. Ultimately, formal methods can only give assurance of the correct functioning of the system. At the transition from informal to top-level formal specification stage of development, comprehension of the formalisms used increases assurance that the customers' requirements have been properly represented. Thus, the clarity of the formalisms contribute directly to this assurance of the system. A discussion of this aspect of assurance has been described elsewhere.

Formal developments of concurrent systems have to address additional problems. For example, what constitutes a high-level requirements specification on the concurrent system? The best known approaches are the process algebras such as CCS[2] and the μ-calculus[3], along with tools such as the Concurrency Workbench[4] which support the verification methods offered by these approaches.

In this paper, we report on the formal development of a real-life concurrent system using temporal logic. The difficulty of producing the formal high-level requirements amounts to analysing existing informal requirements written in English. The advantage of using temporal logic is that it addresses concurrency as well as providing an easy transition from informal to formal requirements. A good illustration of this is the specification of a lift system[5], where it is shown how an informal specification can translate very naturally into temporal logic. In this paper, a real-life system is developed in a similar manner[5]. Furthermore, two levels of development are given, for which a proof has been produced. The stages of development are as follows.

Specification
1. Give informal high-level requirements in English.
2. Perform a temporal analysis of the informal requirements.
3. Produce a temporal specification of the requirements.

Implementation
1. Implement the system.
2. Perform a temporal analysis of the implemented system.
3. Produce a temporal semantics of the implemented system.

Verification
1. Prove the formula valid.

The specified system is a microkernel used on the Esprit II European Declarative System (EDS) Project. The operating system for the EDS[6] was to be UNIX-like with a multi-level process model. As part of the early experimentation with this type of model, a lightweight microkernel was layered on top of standard UNIX processes. This microkernel enabled lightweight microprocesses to be scheduled, thus providing fine-grained non-deterministic multi-programming. The microkernel is documented informally elsewhere.[7] The formal specification of the microkernel contained in this paper is a derivative of the original version[8] and gives full semantics for the temporal logic used. A correctness proof for the version in this paper is documented elsewhere[9].

In the next Section, a first-order temporal logic with a slightly unusual semantics is defined. It is used to specify the microkernel. We describe a top-level formal specification of the microkernel by analysing the informal requirements given previously[7]. This is followed by the temporal semantics of the implementation produced by a temporal analysis of the implemented system as described previously[7].

*The parts of that work[7] that relate to the formal specification here are reproduced in this paper.
2 Temporal language

The system is specified in a first-order temporal logic where predicates as well as propositions have time-dependent meaning. This differs from the more common usage where predicates have time-independent meanings. The latter are termed 'rigid' predicates and the former 'flexible' predicates [lo], and so the language below is denoted FLTL. Flexible predicates are useful for specifying resources shared by several processes, as seen in examples elsewhere [11, 12]. The domain of the predicates is understood to be the non-negative integers \( \mathbb{N} \). These represent processes in the microkernel specification, and the predicates are statements about the processes. The full overall system state of the implemented system is described later. The aspects of this state that appear in the model obtained from \( \mathcal{M} \) by modifying its assignment function \( \alpha \) to map the variables \( n_1, \ldots, n_k \) to \( a_1, \ldots, a_k \), respectively.

\[
\mathcal{M}_{a_i}^P \quad \leftrightarrow \quad l_{P_0}(P, s_0)
\]

\[
\mathcal{M}_{a_i}^p(n) \quad \leftrightarrow \quad l_{P_0}(P, s_0)(n)
\]

\[
\mathcal{M}_{a_i}^{n_1} = n_2 \quad \leftrightarrow \quad \alpha(n_1) = \alpha(n_2)
\]

\[
\mathcal{M}_{a_i}^\neg \phi \quad \leftrightarrow \quad \mathcal{M}_{a_i}^\phi \text{ is false}
\]

\[
\mathcal{M}_{a_i}^{\phi_1 \land \phi_2} \quad \leftrightarrow \quad \mathcal{M}_{a_i}^{\phi_1} \text{ and } \mathcal{M}_{a_i}^{\phi_2}
\]

\[
\mathcal{M}_{a_i}^\phi \lor \phi_2 \quad \leftrightarrow \quad \mathcal{M}_{a_i}^\phi \text{ or } \mathcal{M}_{a_i}^{\phi_2}
\]

\[
\mathcal{M}_{a_i}^\forall n \cdot \phi \quad \leftrightarrow \quad \mathcal{M}_{a_i}^{(n \leftarrow a)^\top \phi}, \text{ for all } a \in \mathbb{N}
\]

\[
\mathcal{M}_{a_i}^\exists n \cdot \phi \quad \leftrightarrow \quad \mathcal{M}_{a_i}^{(n \leftarrow a)^\bot \phi}, \text{ for some } a \in \mathbb{N}
\]

\[
\mathcal{M}_{a_i}^\top \phi \quad \leftrightarrow \quad \mathcal{M}_{a_0}^{\top \phi}, \text{ for some } i \in \mathbb{N}
\]

\[
\mathcal{M}_{a_i}^\bot \phi \quad \leftrightarrow \quad \mathcal{M}_{a_0}^{\bot \phi}, \text{ and }
\]

\[
\mathcal{M}_{a_i}^\bot \phi_1 \quad (0 \leq j < i)
\]

Despite the unpleasant appearance of the semantics, it is shown below that the specifications can be easily understood by reading \( \top \) as 'next point in time', \( \bot \) as 'always', \( \neg \) as 'sometimes' and \( \bot \) as 'until'.

3 Specification of requirements

Informally, the system has the following components: processes and a scheduler.

The system involves processes being serviced by the processor and switched by the scheduler on expiry of their allotted time slice. This is caused by a timer signal. However, a problem arises:

'... when the timer causes a signal to occur whilst a process is in kernel mode. The switcher (scheduler) must not schedule another process since to do so might lead to the corruption of kernel data structures, but on the other hand to give the currently executing process another time slice would be unfair to other processes which are ready to run. Indeed if this solution was adopted then a process could continue indefinitely by always being in kernel when the timer signal is received. A compromise solution is to allow a process to continue after its time slice ends if it is in kernel mode when this happens, but to force a context switch when kernel mode is left... [7]'

In order to provide a formal specification of this, it is necessary to define a system state formally.

3.1 Aspects of system state

The full overall system state of the implemented system is described later. The aspects of this state that appear in the high-level specification are discussed here. They correspond to predicates which are either true or false at a
given point in time:

- \( p(n) \): process \( n \) is active
- \( k(n) \): process \( n \) is active and in kernel mode
- \( \text{start}(n) \): process \( n \) performing start of kernel work
- \( \text{end}(n) \): process \( n \) on the point of completing kernel work
- \( sg \): a timer signal is occurring
- \( sw \): a new process will be scheduled at the next instant

The required system is a set of sequences of states represented by a formula in FLTL whose set of models yield this set of sequences precisely.

### 3.2 Temporal specification of requirements

The following are the constraints on how the processes may be scheduled. From the informal requirements, a process is to continue after its time slice ends if it is in kernel mode, but is to force a context switch when kernel mode is left. From this, it is clear that two situations are allowed to occur.

1. The process is switched at the end of its time slice as it is not in kernel mode.
2. The process is in kernel mode when its time slice expires, and so it continues in an active state until it has completed that block of kernel work and, at the end of that, a process switch occurs.

This behaviour is expressed formally by describing what may happen between consecutive process switches. In other words, if a switch occurs at some point in time, what can happen up to the next process switch? The appropriate condition for the switching constraints is thus of the form

\[ \sigma = \square (sw \Rightarrow \sigma_1 \lor \sigma_2) \]

where \( \sigma_1 \) and \( \sigma_2 \) correspond to the two situations given above.

\[ \sigma_1 = \exists n. \bigcirc (\neg \neg sw \land \neg \neg sg \land p(n)) \Rightarrow (sg \land sw \land p(n) \land (\neg \neg k(n) \land \neg \neg end(n))) \]

is the normal situation where, after a process switch, there is no process switch, no timer signal and the process is active until a timer signal does occur, at which point there is a process switch as the active process was either not in kernel or at the point of exit from kernel mode.

\[ \sigma_2 = \exists n. \bigcirc (\neg sw \land \neg sg \land p(n)) \Rightarrow (sg \land \neg sw \land k(n) \land \neg end(n)) \land \bigcirc (\neg sg \land \neg sw \land \neg \text{start}(n)) \Rightarrow (sw \land \neg end(n))) \]

is when, after a process switch, there is a period of no switching and no timer signal until a timer signal occurs when the active process is in kernel mode. Thus, it continues, without being interrupted by either a timer signal or process switch and without starting new kernel work, up to the point at which it completes its portion of kernel work and finally a process switch occurs. Notice how closely the verbal description follows the temporal logic.

### 4 Implementation semantics

#### 4.1 System state

The implemented system has as its parallel components an arbitrary number of processes and a scheduler. The execution of the overall system takes place in discrete steps. Each discrete step is associated with an overall system state or program state [13]. A program state [13] comprises

1. the values of variables accessed by the components.
2. the label of the next instruction to be executed in each individual component.
3. the next component to be scheduled.

Condition 1 is a statement about the values of variables. Conditions 2 and 3 are statements relating to the scheduling of components. Here, the system (program) state is given by 21 propositions or predicates as follows:

1. (Boolean) variables:
   - \( c \): the critical flag is set to true
   - \( sx \): the switch_on_exit flag is set to true
2. (a) Label of instruction being executed by a process \( n \), one of:
   - \( \text{label1}(n) \), \( \text{label2}(n) \), \( \text{label3}(n) \), \( \text{label4}(n) \), \( \text{label5}(n) \), \( \text{label6}(n) \), \( \text{label7}(n) \), \( \text{label8}(n) \)
3. Properties of instruction being executed:
   - \( \text{p}(n) \): process \( n \) is active
   - \( \text{k}(n) \): process \( n \) is in kernel mode
   - \( \text{start}(n) \): process \( n \) is performing start of kernel work
   - \( \text{end}(n) \): process \( n \) on the point of completing kernel work
   - \( \text{sc} \): the critical flag is being set
   - \( \text{psetc} \): the switch_on_exit flag is being set by a process
   - \( \text{psuw} \): a process switch is being initiated by a process
3. Scheduling:
   - \( \text{sw} \): a process switch is being initiated
   - \( \text{ssw} \): a process switch is being initiated by the scheduler
   - \( \text{sssetc} \): the switch_on_exit flag is being set by the scheduler
   - \( \text{sg} \): a timer signal is occurring

**Remarks**

(i) The required scheduling is implemented by the use of two flags, critical and switch_on_exit, which are set at various times, and on the basis of which a process switch is initiated either by a process or the scheduler component.

(ii) The system state is seen to last for the whole duration of the current time, rather than be some entry or exit condition [13]. For example, if \( sw \) is true, a process switch is being initiated, although the old process remains active...
the 'chop' operator and a hed-point constructor, at the expense of obtaining and having this new value throughout this point in time. A new process will be active throughout the next point in time. As another example, the setting of a flag, such as described by sefc, lasts for the duration of current time and is deemed to coincide with obtaining and having this new value throughout the current time (and so, in this last respect, it differs from the process switching situation).

(iii) As is seen below, the test of the condition in an if statement executed by a process takes an instant in time.

The semantics of the system are the sequences of system states that are allowed to occur. These will be expressed as solutions to FLTL formulae. The allowable sequences of states are affected by the constraints of the process components and the constraints of the scheduler component.

The constraints of the process components are given below, by analysing the previous C code [7]. The scheduler constraints are given by analysing the description of the low-level scheduler used previously [7].

4.2 Process component constraints

4.2.1 Interleaving of labelled statements: each process is modelled as executing infinitely, occasionally in kernel mode, and sometimes executing an exit protocol on completion of kernel mode. Precisely, each process $n$ repeatedly executes the 'cycle' of labelled statements given below

\[
\begin{array}{l}
\text{label0}(n): \text{some non-kernel mode instruction}; \\
\text{label1}(n): \text{critical} = \text{TRUE}; \\
\text{label2}(n): \text{some kernel mode instruction}; \\
\text{label3}(n): \text{critical} = \text{FALSE}; \\
\text{label4}(n): \text{if (switch\_on\_exit) } \{ \\
\text{label5}(n): \text{switch\_on\_exit} = \text{FALSE}; \\
\text{label6}(n): \text{do\_switch}; \\
\text{label7}(n): \text{some non-kernel mode instruction}; \\
\end{array}
\]

where the C code on the right-hand side of the labels is taken from the earlier work [7]. The function _do_switch is a routine which performs a process switch and resets the _critical flag.

The behaviour of all the processes together is an interleaving of the labelled statements executed by the individual processes. To avoid excessive use of brackets, the following notation is used:

\[
\begin{array}{l}
\text{period} \phi_1 \quad \text{point} \phi_2 \ldots \quad \text{period} \phi_{2i-1} \quad \text{point} \phi_{2i} \\
\ldots \quad \text{period} \phi_{2m-1} \quad \text{point} \phi_{2m}
\end{array}
\]

\[
\phi_1 \triangledown ((\phi_2 \land \bigcirc(\ldots(\phi_{2i-1} \land \bigcirc(\ldots(\phi_{2m-1} \land \bigcirc(\ldots)\ldots)))) \\
\text{2m-1 brackets})
\]

\[\text{def}\]

† The intention in this paper is to keep the basic connectives as simple as possible and to introduce suitable 'higher level' ones to aid readability. More brevity was achieved previously [8] by use of the 'chop' operator and a fixed point constructor, at the expense of readability. A compositional [14] specification was also given.

It indicates several steps taking place over a period of time alternated with a step at a single point in time. The interleaving of the processes is given by the following four formulae:

\[
i_1 = [\square \forall n. \text{label1}(n)
\Rightarrow (\text{period} (\neg p(n)) \\
\text{point} (\text{label1}(n)) \\
\text{period} (\text{label2}(n) \lor \neg p(n)) \\
\text{point} (\text{label3}(n)) \\
\text{period} (\neg p(n)) \\
\text{point} (\text{label4}(n)) \\
\text{period} (\neg p(n)) \\
\text{point} (\text{label5}(n)) \\
\text{period} (\neg p(n)) \\
\text{point} (\text{label6}(n)) \\
\text{period} (\neg p(n)) \\
\text{point} (\text{label7}(n)) \\
\text{period} ((p(n) \land \neg k(n)) \lor \neg p(n)) \\
\text{point} (\text{label8}(n)))
\]

\[
i_2 = \forall n. ((\neg p(n) \lor (p(n) \land \neg k(n))) \square (\text{label1}(n))
\]

\[
i_3 = [\square \exists i. \text{label1}(n)
\]

\[
i_4 = [\square \forall n \forall m \forall i. ((\neg m = n) \lor (\neg i = j))
\Rightarrow \neg (\text{label1}(m) \land \text{label1}(j))]
\]

The effect of these four expressions is to say that the behaviour of all the processes together is an interleaving of the cycles of labelled statements executed by the individual processes. The last two expressions state that exactly one labelled statement is being executed at a given time. In the first expression, the large disjunction results from the testing of the switch\_on\_exit flag and the execution of additional code if the value is true.

4.2.2 Properties of labelled statements: the interpretation of the labelled statements in terms of $p(n), k(n)$, start($n$), end($n$), psu, sefc, psetsx, $c$ and $s$ based on the C code given above is as follows:

\[
\begin{array}{l}
\text{label1}(n) \iff p(n) \land k(n) \land \text{start}(n) \land \text{end}(n) \\
\text{label2}(n) \iff p(n) \land k(n) \land \neg \text{start}(n) \land \neg \text{end}(n) \\
\text{label3}(n) \iff p(n) \land k(n) \land \text{start}(n) \land \text{end}(n)
\end{array}
\]

\[\square \forall m \forall \forall i. ((\neg m = n) \lor (\neg i = j))\]

§ Quantifying over $i$ in label1 is a slight abuse of notation used to shorten the expression. The meaning should be clear.

Software Engineering Journal January 1995
Software Engineering Journal January 1995

4.3 Scheduler component constraints

4.3.1 Basic process switching constraints: first, the basic property of process switching is given. A process remains active if no switch occurs

\[ i_{10} \equiv \Box \forall n. (\neg sw \land p(n) \Rightarrow \Box p(n)) \]

If a switch occurs, there is a change in process at the next instant in time

\[ i_{11} \equiv \Box \forall n. (sw \land p(n) \Rightarrow \Box \neg p(n)) \]

A switch may be initiated by a process or by the scheduler

\[ i_{12} \equiv \Box (sw \Rightarrow ps\\wedge ssw) \]

4.3.2 Low-level scheduler constraints: the low-level scheduler used has the following properties [7]:

'A scheduler initiated action can only occur at a point at which a timer signal is occurring'

\[ i_{13} \equiv \Box ssw \lor ssetc \lor ssetcx \Rightarrow sg \]

'At the instant of a process switch the timer is set up for the next time slice'

\[ i_{14} \equiv \Box (sw \Rightarrow \Box ((\neg sg \land \neg sw) \forall (sg \lor sw))) \]

This last requirement states that a switch is followed by a period of no switching and no timer signal until a timer signal or another switch occurs.

'If there is a timer signal but no process switch, then the timer signal will not be reset, and hence will not go off until after the next process switch occurs'

\[ i_{15} \equiv \Box ((lg \land \neg sw) \Rightarrow \Box (\neg sg \land \neg sw) \forall (sw \land \neg sg)) \]

When the timer signal occurs the scheduler tests the critical flag. If it is false then a scheduler initiated process switch occurs and the switch_on_exit flag is cleared. Otherwise, if the critical flag is true, no timer initiated switch occurs and the switch_on_exit flag is set to true

\[ i_{16} \equiv \Box sg \Rightarrow (\neg c \Rightarrow (ssw \land ssetc \land \neg sx)) \land (c \Rightarrow (\neg ssw \land ssetc \land sx)) \]

Implicit in the English statement of the last property is that the only time that a scheduler initiated process switch occurs is when the timer signal is available and the critical flag is false. The following condition is needed for verification:*

\[ i_{17} \equiv \bigwedge_{j=1}^{17} i_j \]

5 Conclusions and future work

This case study has demonstrated the use of temporal logic, with a real-life system, to produce a formal specification of the requirements and implementation for the purpose of verifying the system. The route from informal description to formal specification has followed a very natural path. As such it proved to be a low-cost activity in the development of the system.

A formal proof obligation for the correctness of the system is constructed very easily. To prove the micro-kernel correct, it is necessary to show that the implementation satisfies the specification, which amounts to demonstrating the validity of the FLTL formula

\[ i \Rightarrow \sigma \]

An extended version of this paper [9] contains a lengthy correctness proof of this formula based on a rigorous argument, which details how a formal proof would proceed. The rigorous proof was used to establish the absence of errors in the system after the system had undergone extensive testing to remove errors.

There are numerous reasons why a formal proof was not envisaged. First, formal proofs are theoretically impossible for the whole of FLTL. Even with finiteness assumptions on the number of processes to reduce formulae to propositional logic (PTL), the scale of the problem would preclude the use of any of the PTL theorem-provers in

\[ \text{[9]} \]

* The condition was only noticed later when difficulties were encountered in verifying the system.
existence such as dp [15]. However, it is hoped that having available a temporal development of a real-life system will suggest suitable proof assistants that might be produced and used, perhaps in conjunction with the development of improved theorem-provers, to elevate such rigorous arguments to full formal proofs. At the very least, it provides an idea of what it will take to formally develop and verify such a real-life concurrent system.

6 Acknowledgments

The author would like to thank Sean Holdsworth who designed the microkernel and the referee who made extensive comments on a previous version of this paper. This work was partly supported under ESPRIT II grant EP2025, the EDS Project.

7 References


© IEE: 1995

The paper was first received 3 May and in revised form 24 October 1994.

The author is with the Department of Computer Studies, Loughborough University of Technology, Loughborough, Leicestershire LE11 3TU, UK.