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 [1].

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 π-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 σ of the requirements.

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

Verification
1. Prove the formula $\iota \Rightarrow \sigma$ 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 possibly the more common usage where predicates have time-independent meanings. The latter are termed 'rigid' predicates and the former 'flexible' predicates [10], 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 $\mathbf{N}$. These represent processes in the microkernel specification, and the predicates are statements about the processes. The full syntax and semantics of FLTL are given below.

2.1 Syntax

Symbols
The language FLTL has the following symbols:

- a set of proposition symbols *Pro*
- a set of predicate symbols *Pre*
- the equality symbol $=$
- global variable symbols $m, n, \ldots$
- connectives $\neg$, $\wedge$, and $\vee$
- temporal operators $\bigcirc$, $\Box$, $\Diamond$ and $\mathcal{U}$
- quantifiers $\forall$ and $\exists$

Formation rules
The formulae of FLTL are as follows:

- a proposition symbol P is a formula
- if p is a predicates symbol and n is a variable symbol, then $p(n)$ is a formula
- if m and n are variables, then $m = n$ is a formula
- if ϕ_1 and ϕ_2 are formulae, then so are $\neg\phi_1$, $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$
- if ϕ_1 and ϕ_2 are formulae, then so are $\bigcirc\phi_1$, $\Box\phi_1$, $\Diamond\phi_1$ and $\phi_1 \mathcal{U} \phi_2$
- if n is a variable and ϕ is a formula, then $\forall n \,.\, \phi$ and $\exists n \,.\, \phi$ are formulae

2.2 Semantics

Time is assumed to be linear and discrete. Thus, a model $\mathcal{M}$ is a pair $\langle \alpha, I \rangle$, where

- α is an assignment to variables, i.e. a function from the set of variables to the non-negative integers.
- the interpretation I gives a meaning to proposition and predicate symbols at each state, so $I = (I_{Pro}, I_{Pre})$, where

$$I_{Pro}: Pro \times \mathbf{N} \to \{\mathbf{true}, \mathbf{false}\},$$

$$I_{Pre}: Pre \times \mathbf{N} \to (\mathbf{N} \to \{\mathbf{true}, \mathbf{false}\})$$

The semantics is given by a satisfaction relation $\models$ between model/current state pairs and formulae

$$\mathcal{M}_{s_0} \models \phi$$

If $a_1, \ldots, a_k$ are integers and $n_1, \ldots, n_k$ are variables, then

$$\mathcal{M}.\langle n_1 \leftarrow a_1, \ldots, n_k \leftarrow a_k \rangle$$

denotes the model obtained from $\mathcal{M}$ by modifying its assignment function α to map the variables $n_1, \ldots, n_k$ to $a_1, \ldots, a_k$, respectively.

$$\mathcal{M}_{s_0} \models P \quad \Leftrightarrow I_{Pro}(P, s_0)$$

$$\mathcal{M}_{s_0} \models p(n) \quad \Leftrightarrow I_{Pre}(p, s_0)(n)$$

$$\mathcal{M}_{s_0} \models n_1 = n_2 \Leftrightarrow \alpha(n_1) = \alpha(n_2)$$

$$\mathcal{M}_{s_0} \models \neg\phi \quad \Leftrightarrow \mathcal{M}_{s_0} \models \phi \text{ is } \mathbf{false}$$

$$\mathcal{M}_{s_0} \models \phi_1 \wedge \phi_2 \Leftrightarrow \mathcal{M}_{s_0} \models \phi_1 \text{ and } \mathcal{M}_{s_0} \models \phi_2$$

$$\mathcal{M}_{s_0} \models \phi_1 \vee \phi_2 \Leftrightarrow \mathcal{M}_{s_0} \models \phi_1 \text{ or } \mathcal{M}_{s_0} \models \phi_2$$

$$\mathcal{M}_{s_0} \models \forall n \,.\, \phi \quad \Leftrightarrow \mathcal{M}.\langle n \leftarrow a \rangle_{s_0} \models \phi, \text{ for all } a \in \mathbf{N}$$

$$\mathcal{M}_{s_0} \models \exists n \,.\, \phi \quad \Leftrightarrow \mathcal{M}.\langle n \leftarrow a \rangle_{s_0} \models \phi, \text{ for some } a \in \mathbf{N}$$

$$\mathcal{M}_{s_0} \models \bigcirc\phi \quad \Leftrightarrow \mathcal{M}_{s_0+1} \models \phi$$

$$\mathcal{M}_{s_0} \models \Box\phi \quad \Leftrightarrow \mathcal{M}_{s_0+i} \models \phi, \text{ for all } i \in \mathbf{N}$$

$$\mathcal{M}_{s_0} \models \Diamond\phi \quad \Leftrightarrow \mathcal{M}_{s_0+i} \models \phi, \text{ for some } i \in \mathbf{N}$$

$$\mathcal{M}_{s_0} \models \phi_1 \mathcal{U} \phi_2 \Leftrightarrow \text{for some } i \in \mathbf{N}, \mathcal{M}_{s_0+i} \models \phi_2 \text{ and } \mathcal{M}_{s_0+j} \models \phi_1 \ (0 \leqslant j < i)$$

Despite the unpleasant appearance of the semantics, it is shown below that the specifications can be easily understood by reading $\bigcirc$ as 'next point in time', $\Box$ as 'always', $\Diamond$ as 'sometimes' and $\mathcal{U}$ 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 alloted 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
- $start(n)$: process n performing start of kernel work
- $end(n)$: 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

$$\sigma \stackrel{\text{def}}{=} \Box(sw \Rightarrow \sigma_1 \vee \sigma_2)$$

where σ_1 and σ_2 correspond to the two situations given above.

$$\sigma_1 \stackrel{\text{def}}{=} \exists n \,.\, \bigcirc((\neg sw \wedge \neg sg \wedge p(n)) \mathcal{W} (sg \wedge sw \wedge p(n) \wedge (\neg k(n) \vee end))))$$

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.

$$\sigma_2 \stackrel{\text{def}}{=} \exists n \,.\, \bigcirc((\neg sw \wedge \neg sg \wedge p(n)) \mathcal{W} ((sg \wedge \neg sw \wedge k(n) \wedge \neg end(n)) \wedge \bigcirc((\neg sg \wedge \neg sw \wedge \neg start(n)) \mathcal{W} (sw \wedge end(n)))))$$

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:
   $label1(n)$, $label2(n)$, $label3(n)$, $label4(n)$, $label5(n)$, $label6(n)$, $label7(n)$, $label8(n)$
   (b) Properties of instruction being executed:
   $p(n)$: process n is active
   $k(n)$: process n is in kernel mode
   $start(n)$: process n is performing start of kernel work
   $end(n)$: process n on the point of completing kernel work
   $setc$: the *critical* flag is being set
   $psetsx$: the *switch_on_exit* flag is being set by a process
   psw: a process switch is being initiated by a process
3. Scheduling:
   sw: a process switch is being initiated
   ssw: a process switch is being initiated by the scheduler
   $ssetsx$: the *switch_on_exit* flag is being set by the scheduler
   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

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 *setc*, lasts for the duration of current time and is deemed to coincide with *c* obtaining and having this new value *throughout* this *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

```
.....................................................
label8(n): ⟨some non-kernel mode instruction⟩;
label1(n): _critical = TRUE;
label2(n): ⟨some kernel mode instruction⟩;
.....................................................
label2(n): ⟨some kernel mode instruction⟩;
label3(n): _critical = FALSE;
label4(n): if (switch_on_exit) {
label5(n): _critical = TRUE;
label6(n): _switch_on_exit = FALSE;
label7(n): _do_switch()};
label8(n): ⟨some non-kernel mode instruction⟩;
.....................................................
```

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:†

$$\textbf{period}\ \phi_1\ \textbf{point}\ \phi_2 \ldots \textbf{period}\ \phi_{2i-1}\ \textbf{point}\ \phi_{2i} \ldots \textbf{period}\ \phi_{2m-1}\ \textbf{point}\ \phi_{2m} \stackrel{\text{def}}{=} \phi_1 \mathcal{W} (\phi_2 \wedge \bigcirc(\ldots(\phi_{2i-1} \mathcal{W} \bigcirc(\phi_{2i} \wedge \bigcirc(\ldots \phi_{2m-1} \mathcal{W} (\phi_{2m} \underbrace{)) \ldots))}_{2m-1 \text{ brackets}}$$

† 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:

$$\iota_1 \stackrel{\text{def}}{=} \Box\, \forall n\,.\, (label1(n) \Rightarrow ((\ \textbf{period}\ (\neg p(n))\ \textbf{point}\ (label1(n))\ \textbf{period}\ (label2(n) \vee \neg p(n))\ \textbf{point}\ (label3(n))\ \textbf{period}\ (\neg p(n))\ \textbf{point}\ (sx \wedge label4(n))\ \textbf{period}\ (\neg p(n))\ \textbf{point}\ (label5(n))\ \textbf{period}\ (\neg p(n))\ \textbf{point}\ (label6(n))\ \textbf{period}\ (\neg p(n))\ \textbf{point}\ (label7(n))\ \textbf{period}\ ((p(n) \wedge \neg k(n)) \vee \neg p(n))\ \textbf{point}\ (label1(n)))\ \vee (\ \textbf{period}\ (\neg p(n))\ \textbf{point}\ (label1(n))\ \textbf{period}\ (label2(n) \vee \neg p(n))\ \textbf{point}\ (label3(n))\ \textbf{period}\ (\neg p(n))\ \textbf{point}\ (\neg sx \wedge label4(n))\ \textbf{period}\ ((p(n) \wedge \neg k(n)) \vee \neg p(n))\ \textbf{point}\ (label1(n)))))$$

$$\iota_2 \stackrel{\text{def}}{=} \forall n\,.\, (\neg p(n) \vee (p(n) \wedge \neg k(n)))\mathcal{W}(label1(n))$$

$$\iota_3 \stackrel{\text{def}}{=} \Box\, \exists n\, \exists i\,.\, labeli(n)$$

$$\iota_4 \stackrel{\text{def}}{=} \Box\, \forall n\, \forall m\, \forall i\, \forall j\,.\, ((\neg m = n) \vee (\neg i = j)) \Rightarrow \neg(labeli(m) \wedge labelj(n))\S$$

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 $p(n)$, $k(n)$, $start(n)$, $end(n)$, psw, $setc$, $psetsx$, c and sx based on the C code given above is as follows:

$$label1(n) \Leftrightarrow p(n) \wedge k(n) \wedge start(n) \wedge \neg end(n) \wedge \neg psw \wedge setc \wedge c \wedge \neg psetsx$$
$$label2(n) \Leftrightarrow p(n) \wedge k(n) \wedge \neg start(n) \wedge \neg end(n) \wedge \neg psw \wedge \neg setc \wedge \neg psetsx$$
$$label3(n) \Leftrightarrow p(n) \wedge k(n) \wedge \neg start(n) \wedge end(n) \wedge \neg psw \wedge setc \wedge \neg c \wedge \neg psetsx$$

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

$$label4(n) \Leftrightarrow p(n) \wedge \neg k(n) \wedge \neg start(n) \wedge \neg end(n) \wedge \neg psw \wedge \neg setc \wedge \neg psetsx$$
$$label5(n) \Leftrightarrow p(n) \wedge k(n) \wedge \neg start(n) \wedge \neg end(n) \wedge \neg psw \wedge setc \wedge c \wedge \neg psetsx$$
$$label6(n) \Leftrightarrow p(n) \wedge k(n) \wedge \neg start(n) \wedge \neg end(n) \wedge \neg psw \wedge \neg setc \wedge psetsx \wedge \neg sx$$
$$label7(n) \Leftrightarrow p(n) \wedge k(n) \wedge \neg start(n) \wedge end(n) \wedge psw \wedge setc \wedge \neg c \wedge \neg psetsx$$
$$label8(n) \Leftrightarrow p(n) \wedge \neg k(n) \wedge \neg start(n) \wedge \neg end(n) \wedge \neg psw \wedge \neg setc \wedge \neg psetsx$$

and

$$p(n) \vee k(n) \vee start(n) \vee end(n) \Rightarrow \exists i \,.\, label i(n)$$

The conjunction of these nine conditions is denoted by ι_5.

4.2.3 Properties of flag variables: if the *critical* and *switch_on_exit* flags are not set, conditions have to be given to indicate that they do not change

$$\iota_6 \stackrel{def}{=} \Box\,((c \wedge \bigcirc \neg c) \vee (\neg c \wedge \bigcirc c) \Rightarrow \bigcirc setc)$$

$$\iota_7 \stackrel{def}{=} \Box((sx \wedge \bigcirc \neg sx) \vee (\neg sx \wedge \bigcirc sx) \Rightarrow \bigcirc setsx)$$

The *switch_on_exit* flag may be set either by a process or the scheduler (see below)

$$\iota_8 \stackrel{def}{=} \Box(setsx \Leftrightarrow psetsx \vee ssetsx)$$

Initially, both flags are false

$$\iota_9 \stackrel{def}{=} \neg c \wedge \neg sx$$

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

$$\iota_{10} \stackrel{def}{=} \Box\ \forall n \,.\, (\neg sw \wedge p(n) \Rightarrow \bigcirc p(n))$$

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

$$\iota_{11} \stackrel{def}{=} \Box\ \forall n \,.\, (sw \wedge p(n) \Rightarrow \bigcirc \neg p(n))$$

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

$$\iota_{12} \stackrel{def}{=} \Box(sw \Leftrightarrow psw \vee ssw)$$

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'

$$\iota_{13} \stackrel{def}{=} \Box ssw \vee ssetc \vee ssetsx \Rightarrow sg$$

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

$$\iota_{14} \stackrel{def}{=} \Box(sw \Rightarrow \bigcirc((\neg sg \wedge \neg sw) \mathcal{W} (sg \vee sw)))$$

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'

$$\iota_{15} \stackrel{def}{=} \Box((sg \wedge \neg sw) \Rightarrow \bigcirc((\neg sg \wedge \neg sw) \mathcal{W} (sw \wedge \neg sg)))$$

'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'

$$\iota_{16} \stackrel{def}{=} \Box sg \Rightarrow ((\neg c \Rightarrow (ssw \wedge ssetsx \wedge \neg sx)) \wedge (c \Rightarrow (\neg ssw \wedge ssetsx \wedge sx)))$$

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 occurs and the *critical* flag is false. The following condition is needed for verification:*

$$\iota_{17} \stackrel{def}{=} \Box(ssw \Rightarrow (sg \wedge \neg c))$$

4.4 Overall semantics

The overall temporal behaviour ι of the implementation can now be given

$$\iota \stackrel{def}{=} \bigwedge_{j=1}^{17} \iota_j$$

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 microkernel correct, it is necessary to show that the implementation satisfies the specification, which amounts to demonstrating the validity of the FLTL formula

$$\iota \Rightarrow \sigma$$

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

* 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

[1] BARROCA, L.M., and McDERMID, J.A.: 'Formal methods: use and relevance for the development of safety-critical systems', *Comput. J.*, 1992, **35,** (6), pp. 579–599

[2] MILNER, R.: 'Communication and concurrency' (Prentice-Hall, 1989)

[3] MILNER, R., PARROW, J.G., and WALKER, D.J.: 'A calculus of mobile processes I', *Inf. Computat.*, 1992, **100,** pp. 1–40

[4] CLEAVELAND, R., PARROW, J., and STEFFEN, B.: 'The Concurrency Workbench: a semantics-based tool for the verification of concurrent systems', *ACM TOPLAS*, 1993, **15,** (1), pp. 36–72

[5] BARRINGER, H.: 'Up and down the temporal way', *Comput. J.*, 1987, **30,** (2), pp. 134–148

[6] ISTAVRINOS, P.: 'Specification of the process control language (PCL)'. EDS Project document, EDS.DD.1S.0007

[7] HOLDSWORTH, S.: 'A proposal for the provision of an environment for studying distributed operating system issues'. EDS Project document, Department of Computer Science, University of Manchester, UK, 1989

[8] HUSSAK, W.: 'Specification of a distributed operating system environment'. EDS Project document, Department of Computer Science, University of Manchester, UK, 1989

[9] HUSSAK, W.: 'Correctness proof for a microkernel'. Internal Report 884, Department of Computer Studies, Loughborough University of Technology, UK, 1994

[10] ABADI, M.: 'Temporal-logic theorem proving'. PhD Thesis, Stanford University, California, STAN-CS-87-1151

[11] BARRINGER, H., FISHER, M., GABBAY, D., GOUGH, G., and OWENS, R.: 'MetateM: a framework for programming in temporal logic'. REX Workshop on Stepwise Refinement of Distributed Systems (*Lect. Notes Comput. Sci.*, 1989, **430**)

[12] KEANE, J.A., and HUSSAK, W.: 'A formal approach to determining parallel resource bindings'. Proc. 16th Int. Conf. on Software Engineering, ICSE '94, Italy, May 1994, pp. 15–22 (IEEE Press)

[13] KRÖGER, F.: 'Temporal logic of programs' (Springer-Verlag, 1987)

[14] BARRINGER, H.: 'Using temporal logic in the compositional specification of concurrent systems'. UMCS-86-10-3, Department of Computer Science, University of Manchester, UK, 1986

[15] GOUGH, G.: 'Decision procedures for temporal logic'. MSc Dissertation, Department of Computer Science, University of Manchester, UK, 1984

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.