Formal Specification for Building Robust Real-time Microkernels

Manuel Rodríguez, Jean-Charles Fabre and Jean Arlat

LAAS-CNRS
7, Avenue du Colonel Roche
31077 Toulouse Cedex 4 — France

E-mail: {rodriguez,fabre,arlat}@laas.fr

Abstract

This paper presents a method based on formal specifications for building robust real-time microkernels. Temporal logic is used to specify the functional and temporal properties of real-time kernels with respect to their main services (e.g., scheduling, time, synchronization, and clock interrupts). As an example of a synchronization mechanism, the specification of the Priority Ceiling Protocol is provided. The objective is to verify kernel properties at runtime in order to improve the internal kernel's detection mechanisms and complement their weaknesses. The core of this paper is a complete description of the temporal logic formulas corresponding to real-time kernel specifications. The formulas developed in this paper are the basis for the implementation of fault containment wrappers. The combination of COTS microkernels and wrappers leads to the notion of robust microkernels. The provided case study illustrates the approach on top of an instance of the Chorus microkernel.

1 Introduction

The use of commercial-off-the-shelf (COTS) real-time microkernels is today a target objective for the designers of safety-critical systems, mainly for economic reasons. However, the main worry of the designers is the correctness of this vital component. Providing a solution to guarantee that the preferred candidate fulfills the expected specifications is of very high interest for system integrators. The expression of the specifications using formal methods increases very much the confidence one can have in the final result. Indeed, formal methods are widely used as a mathematical support for the specification and the verification of systems, as well as for the synthesis of implementations whose correctness is then given by construction. The main contribution of this paper is the ability to combine both formal methods and error detection mechanisms for building fault tolerant systems based on robust real-time microkernels.

The approach proposed aims at providing formal specifications for building robust real-time microkernels. The specifications describe properties that are independent of any actual implementation of a given real-time kernel. Temporal logic is used as the specification language, since it constitutes a good framework for expressing both functional and timing properties of systems, as stated in [1]. Properties of real-time kernels are specified with respect to their main services (e.g., scheduling, time, synchronization, and clock interrupts). As an example of a synchronization mechanism, we provide the specification of the *Priority Ceiling Protocol* [10]. The set of temporal logic formulas can be verified at runtime and the violation of a formula is interpreted as the detection of an error of the kernel. The on-line verification of properties can thus be viewed as a fault containment wrapper based on formal expressions. This approach efficiently combines both an abstract model of the expected behavior of the kernel, as well as error detection mechanisms for hardening the microkernel.

The basic assumption is to consider the microkernel as a finite state machine, as stated in [2]. However, to practically verify kernel properties, it is necessary to reduce the complexity of the model. Our approach consists in defining a nondeterministic, abstract finite state machine (AFSM) of the candidate microkernel, which avoids the so-called state explosion problem [3]. This is viable because the detailed functionality of some parts of the kernel behavior are irrelevant to the properties that need to be checked.

Transitions among states of the AFSM are defined by a number of microkernel events, which are triggered by the kernel execution flow, the kernel environment and the real-time clock. Microkernel events correspond to both external stimuli and the start or termination of actions. The kernel environment can be viewed as being composed of the *user tasks*, which issue system calls, the *real world*, which generates asynchronous inputs, and the *hardware*, which raises interrupts and exceptions (e.g., internal error detection).

The runtime verification of the kernel's properties, with respect to its AFSM, is performed by a model checker, which interprets on-line the temporal logic formulas. The model checker implements a required subset of the temporal logic. In practice, the model checker is implemented as an external module that needs to access internal data and events of the kernel. In short, the model checker is responsible for both the verification of the kernel behavior at runtime, as well as the detection of errors. The description of the model checker is beyond the scope of this paper.

A number of works have been published on the application of formal methods to microkernels. For instance, in [4], a restricted Ada 95 runtime support is specified, together with a minimal model of the system, using a version of RTL expressed in PVS. In [5], some kernel services are specified in VDM, and then compared to a less abstract specification of a simple kernel written in Modula2. In [6], the Z notation is chosen to specify the behavior of the scheduler and interrupt handling components of a kernel for a diagnostic X-ray machine. These works are mainly intended to provide a formal description of the kernel which would replace its ambiguous documentation, and also aim at analyzing kernel properties that may help errors to be detected at early stages in the design process of the system, or in an actual implementation of the kernel. A very interesting architectural model for safety kernels is proposed in [2]. It suggests supplying the kernel with an interface implemented in Ada95 that offers safety services to critical applications. Such an interface is the result of successive refinements from the specification in RTL and Z of safety invariants defined by the applications.

However, to our knowledge, no published work has focused on formal specifications for the runtime verification of the correctness of real-time microkernel services. The runtime verification of formal expressions is a major step forward of our previous work done for the hardening of COTS microkernels for the development of efficient fault tolerant-systems [7]. This work investigated wrappers based on executable assertions instead of formal specifications and it did not consider temporal aspects. The main benefit of the formal specifications described in this paper is two-fold: (i) the formal expressions based on temporal logic express the detailed behavior of a real-time kernel in both time and value domains, (ii) they can be tuned according to the needs and dynamically verified at runtime by a model checker. The combination of both formal specifications and the corresponding model checker can be seen as an extended error detection wrapper.

This paper is organized as follows. Section 2 describes how temporal logic can be extended to specify microkernel services. The specification of an abstract kernel is provided in Section 3, modeling the scheduling, time, synchronization, and clock interrupt services. Verification issues are illustrated in Section 4, regarding the timing properties of an instance of the Chorus microkernel. Finally, Section 5 concludes the paper.

2 Temporal logic for the specification of kernels

Temporal logic [8] is a useful formalism to describe both functional and temporal properties of systems. However, some extensions must be provided so that it can effectively be used to specify kernels. This section presents the temporal logic used for the kernel specification developed in Section 3. The syntax and semantics of the logic are first described, as well as the microkernel events necessary for the specification. The definition of discrete time in kernels is then developed. Finally, additional derived operators and the subset of the temporal logic used are described.

2.1 Syntax and semantics

Temporal logic formulas are built from predicates. A predicate describes or checks the state of the system at a particular instant in time. Predicates are composed of a number of expressions. Expressions are made up of:

- *Constants* ($\mathcal{C}$): They comprise integer numbers (..., -1, 0, 1, ...).
- *Variables* ($\mathcal{V}$): They may be either a general variable (e.g., th_a, which refers to a thread identifier, *period*, which refers to a time interval, etc.) or a *state variable*. By convention, state variables are enclosed in square brackets. Some examples are:
  - *[Running]*: the currently running thread.
  - *[Flow]*: the current execution flow of the kernel.
- *Functions*, $f(e_1, \ldots, e_k)$, where $e_1, \ldots, e_k$ are expressions: Functions may be i) arithmetic operators (e.g., +, -), ii) *state functions*, e.g.:
  - *prio* (th_a): the priority of thread th_a.
  - *highest ([ReadyQueue])*: the highest priority thread in the ready queue.

  and also iii) *microkernel events*. Microkernel events are explained in Section 2.2. Yet, let us give some examples:
  - $\uparrow$*signal (th)*, beginning of thread *th* entering the ready queue.
  - $\downarrow$*context_switch*, end of a context switch operation.

  Inductively, formulas ($\mathcal{F}$) are built as follows:
- *Predicates*, $p(e_1, \ldots, e_k)$, where $e_1, \ldots, e_k$ are expressions: Predicates include the relational operators (e.g., =, <, >, ≤). For example:
  - *prio* (th_b) < *prio* (th_a): is the priority of thread th_b lower than the priority of thread th_a?
  - *[Running]* = th_a: is thread th_a currently running?
  - *[Flow]* = $\uparrow$*SetTimer* (tm_a, *abs, period*): does the execution flow correspond to the beginning of system call *SetTimer*, where timer parameter is equal to tm_a, absolute start time is *abs*, and timer period is *period*?

- Predicates combined with logical operators (e.g., $\neg$, $\wedge$) and temporal operators, such as *Always* ($\Box$), *Next* (O), and *Sometime* ($\Diamond$). For instance, the formula
  $\Box$ *[[ReadyQueue] $\neq \emptyset$] $\wedge$ O [[Flow] = $\uparrow$signal (th_a) $\wedge$ $\Diamond$ [[Running] = th_a]]*
  is true iff the ready queue always contains at least one thread, and at the next instant of time the execution flow corresponds to thread th_a entering the ready queue, which some time later is elected to run.

The truth of a temporal logic formula is given with respect to a model defined by the triple ($\mathcal{D}$, Σ, $\mathcal{M}$), where:

- $\mathcal{D}$ is the data domain. We take $\mathcal{D}$ to be the set of integer numbers, since all the microkernel variables can be represented by an integer. Non integer variables, such as *structs*, can be easily reduced to an integer.
- Σ is the set of microkernel states. Σ is defined by the set of all possible values taken by the microkernel variables at any instant of time. A state is a tuple of type $\mathcal{D}^k$, where k is the number of variables handled by the microkernel.
- $\mathcal{M}$ is an interpretation, giving meaning to every function and predicate symbol, i.e.:
  - Let f be a function symbol, then:
    $\mathcal{M} \| f \| \in (\mathcal{D}^k \rightarrow \mathcal{D})$
  - Let p be a predicate symbol, then:
    $\mathcal{M} \| p \| \in (\mathcal{D}^k \rightarrow \{true, false\})$

Let σ be an interval of states in Σ^+, the set of nonempty, finite sequences of states. Let $| \sigma |$ be the length of σ, by convention equal to the number of states of σ minus one. The individual states of an interval σ are denoted by $\sigma_0, \sigma_1, \ldots, \sigma_{|\sigma|}$. The truth of a temporal logic formula is given with respect to an interval σ, where the first state of the interval (i.e., σ_0) refers to the current time, and successive states refer to successive instants of time. Let C $\in \mathcal{C}$, V $\in \mathcal{V}$, and G, H $\in \mathcal{F}$. Accordingly, $\sigma_0 \|V\|$ corresponds to the value of the variable V at the current time. The semantics of an interpretation $\mathcal{M}$ for a given interval σ, is as follows:

1. $\mathcal{M}_\sigma \| C \| = C$
2. $\mathcal{M}_\sigma \| V \| = \sigma_0 \| V \|$
3. $\mathcal{M}_\sigma \| f(e_1, \ldots, e_k) \| = \mathcal{M}_\sigma \| f \| (\mathcal{M}_\sigma \| e_1 \|, \ldots, \mathcal{M}_\sigma \| e_k \|)$
4. $\mathcal{M}_\sigma \| p(e_1, \ldots, e_k) \| = \mathcal{M}_\sigma \| p \| (\mathcal{M}_\sigma \| e_1 \|, \ldots, \mathcal{M}_\sigma \| e_k \|)$
5. $\mathcal{M}_\sigma \| \neg G \| = \neg \mathcal{M}_\sigma \| G \|$
6. $\mathcal{M}_\sigma \| G \wedge H \| = \mathcal{M}_\sigma \| G \| \wedge \mathcal{M}_\sigma \| H \|$
7. $\mathcal{M}_\sigma \| O\, G \| = \mathcal{M}_{\sigma_1, \ldots, \sigma_{|\sigma|}} \| G \|$
8. $\mathcal{M}_\sigma \| \Box G \| = \mathcal{M}_{\sigma_i, \ldots, \sigma_{|\sigma|}} \| G \|, \forall i \leq |\sigma|$
9. $\mathcal{M}_\sigma \| \Diamond G \| = \neg \mathcal{M}_\sigma \| \Box \neg G \|$

For instance, an interval satisfies $G \wedge H$ if it satisfies both G and H (line 6), and it satisfies *Next* G (line 7) if G is true at the next instant of time, i.e., if the interval obtained by removing the first state of the interval satisfies G.

2.2 Microkernel events

A microkernel event denotes the current execution flow of the kernel. They can be viewed as markers corresponding to changes of the kernel state, triggered by both external stimuli and the start or termination of actions. Clock interrupts, entering a kernel function, or completing a context switch, are examples of events. The set of events needed for the specification developed in Section 3 is given in Table 1.

	Syscall	**Internal**
Scheduling		↑signal (th) ↓context_switch ↑wait ↑yield (th)
Process mgt.	↑Create (th) ↑Relinquish ↓Relinquish ↑Delay (ticks)	↑waitFromDelay
Synchro nization	↑Take (cs) ↓Take (cs) ↑Give (cs) ↓Give (cs)	↓winlock (th) ↓¬winlock (th)
Time mgt.	↑SetTimer (tm, abs, period)	↓TimeoutSet (tm, ticks) ↓TimeoutCancel (tm)

Table 1 Microkernel events

Events have been classified by functional component of the microkernel (**scheduling**, **process management**, **synchronization**, and **time management**). Those events that are triggered by a particular system call handler are referred to as *syscall events*. Otherwise, they are considered as being *internal events*.

All syscall events correspond to the start or termination of a system call handler.

For example:

- *↑Create (th)*: Start of the system call handler for the creation of a new thread *th*.
- *↑Delay (ticks)*: Start of the handler that delays the running thread for *ticks* units of time.
- *↓Take (cs)*: Termination of the handler that controls accesses to critical section *cs*.
- *↑SetTimer (tm, abs, period)*: Start of the handler that sets timer *tm*, with period *period*, and absolute start time *abs*.

The semantics of some internal events is described hereafter (see also Section 2.1):

- *↑wait*: The running thread requests to exit the ready queue.
- *↑yield (th)*: Start of thread *th* voluntarily changing of place in the ready queue.
- *↓winlock (th)*: Thread *th* gains the lock for a critical section.

The description of other microkernel events is explicitly skipped here. A thorough explanation is provided in Section 3.

2.3 Definition of discrete time in a microkernel

Two types of scheduling can be identified: *tick driven* and *event driven*, as stated in [9]. Microkernels manage time differently, depending on which type of scheduling they use. Tick (or timer-driven) scheduling based microkernels use a periodic clock interrupt. At every clock interrupt (e.g., every 10 ms), a handler is in charge of performing a number of tasks, such as updating the system time, looking for elapsed timers, moving threads from a delay queue to the ready queue, etc. Discrete time is thus given by such a periodic clock interrupt. In contrast, event scheduling based microkernels explicitly program the hardware timer to interrupt the system at the next closest timeout. In this case, discrete time is given by each count of the hardware timer (e.g., 1 μs between two counts). This paper adopts the convention of referring to each instant of discrete time (i.e., periodic clock interrupts or hardware timer counts) as *tick*.

In fact, a tick can be viewed as another type of event. Tick events are synchronous, while the other microkernel events are asynchronous. A given tick may match the elapsed time of an on-going timer, whose expiration is managed by the kernel by executing a timeout handler. Let us define two more events. Event *↑tick* denotes the occurrence of a tick. If a timeout is triggered at a given tick, then event *↓tick* denotes the end of processing of the related timeout handler. If no timeout occurs, then both *↑tick* and *↓tick* denote the same instant of time. We assume that a timeout handler finishes within its tick interval.

Since we are only interested in ticks leading to a timeout, we can revise the definition of microkernel states (Σ). Accordingly, Σ will be made up of the instantaneous values of the microkernel variables at the occurrence of either a timeout or an asynchronous event. Subsequent computation steps modifying the kernel variables are thus not to be considered, which significantly reduces the number of states.

2.4 Notation

This section presents some additional derived temporal operators and the subset of the temporal logic used for the kernel specification developed in Section 3.

The temporal operators *Next* and *Sometime* have been extended, as explained hereafter:

- $Next_{\uparrow tick}{}^{i}$: Refers to the next *i*-th *↑tick* event.
- $Next_{\downarrow tick}{}^{i}$: Refers to the next *i*-th *↓tick* event.
- $Next_{event}{}^{i}$: Refers to the next *i*-th asynchronous event.
- $Next_{event}{}^{\{evGroup\}}$: Refers to the next asynchronous event from those of the set *{evGroup}*.
- $Sometime^{<i}$ *(p)*: Refers to some instant in the future before the occurrence of *i ↑tick* events.

The use of these operators is illustrated in Section 3.

The specification uses a subset of the temporal logic. Let *p* and *q* be temporal logic formulas, as defined in Section 2.1. The specification is compliant with the following restrictions:

- *Sometime* is always guarded by an event, for example:
  Sometime ([Flow] = ↑SetTimer $\wedge$ *p)*
  where *p* is not verified until the occurrence of *↑SetTimer*.
- The kernel specification consists of a set of formulas with the following structure:
  Always ([Flow] = event $\wedge$ *p* $\Rightarrow$ *q)*
  which means that the verification of *q* is not carried out until the occurrence of *event*, as long as *p* is true, where *event* is one of the microkernel events given in Table 1, and *p*, *q* are temporal logic formulas which do not use the operator *Always*. The term *[Flow] = event* $\wedge$ *p* is the *antecedent*, which can be thought of as supplying the input or stimulus, whereas the term *q* is the *consequent*, which is the expected behavior of the kernel to the stimulus.

3 Microkernel specification

A microkernel is usually made up of a set of components that provide basic system services, such as scheduling, process management, synchronization, time management, interrupt management, etc. The number of such components varies from one microkernel to another, and most of the times the kernel can be customized so as to be kept as small as possible. The specification of the kernel can thus be made on the basis of these basic components. This paper mainly concentrates on the specification of scheduling, time management, and synchronization, since they are essential services which must be provided by any real-time system. The specification of the real-time clock interrupts is embedded in the temporal logic notation, as explained in Sections 2.3 and 2.4.

The scheduling specification describes the behavior of a general priority-based FIFO-preemptive scheduler (see Section 3.1), commonly used in real-time systems. Real-time tasks usually define a number of timing parameters, such as the period and the deadline, that greatly depend on the correct behavior of the timer service offered by the kernel. The specification of the timer service is developed in Section 3.2. In Section 3.3 we address synchronization services. Two main types of synchronization are supported by any kernel: mutual exclusion between tasks executing user code, and mutual exclusion between tasks executing kernel code. The former gathers user mutexes, condition variables, semaphores, monitors, an also several synchronization protocols for real-time systems. The *Priority Ceiling Protocol* [10], a common mechanism to synchronize real-time tasks, is specified in Section 3.3.1. Finally, synchronization within the kernel is supplied by kernel mutexes, the scheduler lock, and the interrupts lock, as described in Section 3.3.2.

3.1 Scheduling

The scheduling specification describes the behavior of those system calls which can modify the scheduling of tasks, namely, **Create**, **Relinquish**, and **Delay**. This list is not exhaustive, as we mainly aim at illustrating the approach. Synchronization system calls, like **Take** and **Give**, also modify the scheduling but, for the sake of conciseness, they will only be briefly treated in Section 3.3.1.

Consider the request for the creation of a new thread, specified by **Create_1** and **Create_2**:

Create_1

$\underline{\text{Always}}\ [\ [\text{Flow}] = \uparrow\text{Create}(th_b) \wedge th_a = [\text{Running}] \wedge$
$\quad\underline{\text{Sometime}}\ [[\text{Flow}] = \uparrow\text{signal}(th_b) \wedge [\text{Running}] = th_a \wedge$
$\quad\text{prio}(th_b) \leq \text{prio}\ (th_a)] \Rightarrow$
$\qquad\underline{\text{Next}}_{event}\ [[\text{Running}] = th_a \wedge th_b \in [\text{ReadyQueue}_{prio(thb)}]]\ ]$

Create_2

$\underline{\text{Always}}\ [[\text{Flow}] = \uparrow\text{Create}(th_b) \wedge th_a = [\text{Running}] \wedge$
$\quad\underline{\text{Sometime}}\ [[\text{Flow}] = \uparrow\text{signal}(th_b) \wedge [\text{Running}] = th_a \wedge$
$\quad\text{prio}(th_b) > \text{prio}\ (th_a)] \Rightarrow$
$\qquad\underline{\text{Next}}_{event}\ [[\text{Flow}] = \downarrow\text{context\_switch} \wedge$
$\qquad th_a \in [\text{ReadyQueue}_{prio(tha)}] \wedge [\text{Running}] = th_b]\ ]$

Whenever a thread th_a requests to create a new thread th_b, the execution flow, denoted by the variable [Flow], enters the kernel handler devoted to the creation of threads, as described by the predicate $[\text{Flow}] = \uparrow\text{Create}\ (th_b)$. Some time later, the kernel will initiate the insertion of a newly created thread th_b in the ready queue ($\uparrow$signal). If the priority of th_b is lower than or equal to th_a (**Create_1**), by the occurrence of the next event, the insertion operation is finished, thread th_a is still running, and th_b is inserted in the ready queue of its priority. However, if the priority of the newly created thread th_b is higher than the priority of the running thread th_a (**Create_2**), at the next event the kernel executes a context switch. As a result, thread th_a is preempted and inserted in the ready queue, and th_b is elected as the newly running thread.

A thread that gives up the CPU is specified by **Relinquish_1**, **Relinquish_2** and **Relinquish_3**:

Relinquish_1

$\underline{\text{Always}}\ [\ [\text{Flow}] = \uparrow\text{Relinquish} \wedge th_a = [\text{Running}] \wedge$
$[\text{ReadyQueue}]\text{-}th_a \neq \emptyset \wedge th_b = \text{highest}\ ([\text{ReadyQueue}]\text{-}th_a) \wedge$
$\text{prio}\ (th_b) = \text{prio}\ (th_a) \Rightarrow$
$\quad\underline{\text{Next}}_{event}\ [[\text{Flow}] = \uparrow\text{yield}(th_a) \wedge [\text{Running}] = th_a \wedge$
$\qquad\underline{\text{Next}}_{event}\ [[\text{Flow}] = \downarrow\text{context\_switch} \wedge$
$\qquad th_a \in [\text{ReadyQueue}_{prio(tha)}] \wedge [\text{Running}] = th_b]]\ ]$

Relinquish_2

$\underline{\text{Always}}\ [\ [\text{Flow}] = \uparrow\text{Relinquish} \wedge th_a = [\text{Running}] \wedge$
$[\text{ReadyQueue}]\text{-}th_a \neq \emptyset \wedge th_b = \text{highest}\ ([\text{ReadyQueue}]\text{-}th_a) \wedge$
$\text{prio}\ (th_b) < \text{prio}\ (th_a) \Rightarrow$
$\quad\underline{\text{Next}}_{event}\ [[\text{Flow}] = \downarrow\text{Relinquish} \wedge [\text{Running}] = th_a]\ ]$

Relinquish_3

$\underline{\text{Always}}\ [\ [\text{Flow}] = \uparrow\text{Relinquish} \wedge th_a = [\text{Running}] \wedge$
$[\text{ReadyQueue}] - [th_a] = \emptyset \Rightarrow$
$\quad\underline{\text{Next}}_{event}\ [[\text{Flow}] = \downarrow\text{Relinquish} \wedge [\text{Running}] = th_a]\ ]$

Relinquish_1 corresponds to the case where there are at least one ready thread of the same priority as the priority of the running thread th_a, i.e., the ready queue of th_a's priority is not empty. As a result of the relinquish operation, the kernel puts th_a at the end of its ready queue ($\uparrow$yield) and yields the thread at the head of the queue as the newly running thread. Conversely, if all ready threads are of lower priority (**Relinquish_2**), th_a exits the relinquish operation ($\downarrow$Relinquish) without giving up the CPU. Finally, **Relinquish_3** models the hypothetical case where the running thread is the only ready thread in the system (i.e., the idle thread) and decides to relinquish the CPU, being immediately elected to run again.

A thread may decide to delay for a certain time. This is described by **Delay_1**:

Delay_1

$\underline{\text{Always}}\ [\ [\text{Flow}] = \uparrow\text{Delay}\ (\text{ticks}) \wedge \text{ticks} > 0 \wedge th_a = [\text{Running}] \wedge$
$\quad\underline{\text{Sometime}}\ [[\text{Flow}] = \downarrow\text{TimeoutSet}\ (to_a, \text{ticks}) \wedge [\text{Running}] = th_a \wedge$
$\quad\text{systicks} = [\text{SysTicks}] \wedge$
$\qquad\underline{\text{Sometime}}\ [[\text{Flow}] = \uparrow\text{waitFromDelay} \wedge [\text{Running}] = th_a]] \Rightarrow$
$\qquad\quad\underline{\text{Next}}_{event}\ [[\text{Flow}] = \uparrow\text{wait} \wedge [\text{Running}] = th_a \wedge$
$\qquad\qquad\underline{\text{Next}}_{event}[[\text{Flow}] = \downarrow\text{context\_switch} \wedge$
$\qquad\qquad\text{toTicks} = \text{systicks} + \text{ticks} - [\text{SysTicks}] \wedge \text{toTicks} \geq 0 \wedge$
$\qquad\qquad th_a \in [\text{DelayQueue}_{toTicks}] \wedge th_a \notin [\text{ReadyQueue}] \wedge$
$\qquad\qquad [\text{Running}] = \text{highest}\ ([\text{ReadyQueue}]) \wedge$
$\qquad\qquad\quad\underline{\text{Next}}_{\downarrow tick}{}^{toTicks+1}\ [th_a \notin [\text{DelayQueue}] \wedge th_a \notin [\text{ReadyQueue}] \wedge$
$\qquad\qquad\quad [\text{Running}] = \text{highest}\ ([\text{ReadyQueue}]) \wedge$
$\qquad\qquad\qquad\quad\underline{\text{Sometime}}^{<1}\ [[\text{Flow}] = \uparrow\text{signal}\ (th_a) \wedge$
$\qquad\qquad\qquad\qquad\underline{\text{Next}}_{event}\ [th_a \in [\text{ReadyQueue}_{prio(tha)}]]]]]]\ ]$

Whenever the running thread th_a requests to delay for ticks[1] units of time (↑Delay (ticks)), the kernel will attach a timeout object to th_a (↓TimeoutSet (to_a, ticks)). A wait operation is then requested from the delay handler (↑waitFromDelay), which will be served by the scheduler (↑wait). Eventually, th_a leaves the CPU (↓context_switch) and enters the delay queue of its priority ($th_a \in$ [$\mathrm{DelayQueue}_{\mathrm{toTicks}}$]). The number of ticks that th_a should actually wait (toTicks+1) is calculated in the specification from the instant th_a effectively yields the CPU ([SysTicks], i.e., the current system time). When the waiting time is elapsed ($\mathrm{Next}_{\downarrow\mathrm{tick}}^{\mathrm{toTicks+1}}$), the clock interrupt handler must have extracted th_a from the delay queue. Only some time later within the same tick ($\mathrm{Sometime}^{<1}$), right after the clock handler is exited (↑signal), th_a should be made ready by the scheduler ($th_a \in$ [$\mathrm{ReadyQueue}_{\mathrm{prio}(th_a)}$]).

3.2 Timer management

Timer management provides two main system calls: *SetTimer* and *CancelTimer*, specified by **Timer_1** and **Timer_2**, respectively. *SetTimer* takes as input parameters a timer identifier, *tm_a*, an absolute start time, *abs*, and a timer period, *period*. Both *abs* and *period* are given in ticks. If the period parameter is null, *SetTimer* behaves as a *watchdog timer* or an *alarm*. **Timer_1** uses a number of auxiliary predicates, namely: **TooLate**, **Continue**, **LostTimeouts** and **PeriodicTimeout**. Their description is given hereafter:

Timer_1

Always [[Flow] =↑SetTimer(tm_a,abs,period) ∧ $tm_a \neq \varnothing$ ∧
abs > 0 ∧ period ≥ 0 ∧ th_a=[Running] ∧
 Sometime [[Flow] = ↓TimeoutSet (tm_a, ticks) ∧
 [Running] = th_a] ⇒
  TooLate (tm_a, abs, period) ∨ (Continue (abs, period) ∧
  offset = 0 ∧ LostTimeouts (tm_a, abs, period, offset, 0) ∧
  ticks = abs + offset – [SysTicks] – 1 ∧ ticks ≥ 0 ∧
  $tm_a \in$ [$\mathrm{TimeoutQueue}_{\mathrm{ticks}}$] ∧
   ($\mathrm{Next}_{\uparrow\mathrm{tick}}^{\mathrm{abs+offset-[SysTicks]}}$ [$tm_a \in$ [$\mathrm{TimeoutQueue}_0$] ∧
   (period = 0 ∨ (period > 0 ∧
   PeriodicTimeout (tm_a, abs, period, period+offset)))] ∨
    $\mathrm{Sometime}^{<\mathrm{abs+offset-[SysTicks]}}$ [[Flow] = ↓TimeoutCancel (tm_a)
    ∧ $tm_a \notin$ [TimeoutQueue]]))]

TooLate (tm, abs, period)

abs ≤ [SysTicks] ∧ period = 0 ∧ [LostTimeoutCount(tm)] = 1 ∧
tm ∉ [TimeoutQueue]

Continue (abs, period)

abs > [SysTicks] ∨ period > 0

LostTimeouts (tm, abs, period, offset, lost)

to = abs + offset – [SysTicks] ∧
 ((to > 0 ∧ [LostTimeoutCount(tm)] = lost) ∨
 (to ≤ 0 ∧ period > 0 ∧
 LostTimeouts (tm, abs, period, period+offset, lost+1)))

[1] See Section 2.3 for the definition of ticks.

PeriodicTimeout (tm, abs, period, offset)

$\mathrm{Sometime}^{<1}$ [[IntFlow] = ↓TimeoutSet (tm, ticks) ∧ ticks = period
∧ ticks = abs + offset – [SysTicks] ∧ tm ∈ [$\mathrm{TimeoutQueue}_{\mathrm{ticks}}$] ∧
  ($\mathrm{Next}_{\uparrow\mathrm{tick}}^{\mathrm{period}}$ [tm ∈ [$\mathrm{TimeoutQueue}_0$] ∧
  PeriodicTimeout (tm, abs, period, period+offset)] ∨
   $\mathrm{Sometime}^{<\mathrm{period}}$ [[Flow] = ↓TimeoutCancel (tm) ∧
   tm ∉ [TimeoutQueue]])]

A running thread th_a may set a timer tm_a (**Timer_1**), with absolute start time abs, and period equal to period. However, since the kernel is preemptive, an arbitrary amount of time can passed between the *SetTimer* request and its completion by the kernel (↓TimeoutSet). As a consequence, the absolute start time might become lower than the current time (abs ≤ [SysTicks]). In this case, if tm_a behaves as an alarm (period = 0), then it is too late to set the timer (**TooLate**), and a check is made on whether the kernel notified an error ([LostTimeoutCount(tm)] = 1) and tm_a was not inserted in the timeout queue (tm ∉ [TimeoutQueue]). Otherwise (**Continue**), the kernel should set tm_a either at abs (abs > [SysTicks]), or to the next closest period (abs ≤ [SysTicks] ∧ period > 0). **LostTimeouts** checks the number of lost periods, so that the next release time can be computed (ticks = abs + offset – [SysTicks] – 1). Accordingly, the kernel inserts tm_a in the timeout queue with a timeout value of ticks units of time ($tm_a \in$ [$\mathrm{TimeoutQueue}_{\mathrm{ticks}}$]). Then, unless the timer is cancelled (↓TimeoutCancel), it eventually elapses ($\mathrm{Next}_{\uparrow\mathrm{tick}}^{\mathrm{abs+offset-[SysTicks]}}$). A check is made right before the timeout handler is executed on whether tm_a is in the timeout queue of 0 ticks ($tm_a \in$ [$\mathrm{TimeoutQueue}_0$]). This process is periodically repeated, as specified by **PeriodicTimeout**, i.e., at every expiration of the period, the clock interrupt handler inserts tm_a in the timeout queue with a timeout value of period ticks, until it is cancelled.

The specification for *CancelTimer*, **Timer_2**, is the following:

Timer_2

Always [[Flow] = ↑CancelTimer (tm_a) ∧ $tm_a \in$ [TimeoutQueue]
∧ th_a = [Running] ⇒
 Sometime [[Flow] = ↓TimeoutCancel (tm_a) ∧ [Running] = th_a ∧
 $tm_a \notin$ [TimeoutQueue]]

Timer_2 expresses that whenever a request to cancel an on-going timer tm_a is issued, some time later, the kernel extracts tm_a from the timeout queue.

3.3 Synchronization

3.3.1 Priority Ceiling Protocol

The Priority Ceiling Protocol (PCP) [10] provides two main system calls, namely: **Take**, to ask for access to a critical section, and **Give**, to release a critical section. The protocol also defines two special queues: the *block queue* ([BlockQueue]), which is a priority-ordered list of threads blocked by the ceiling protocol, and the *lock* *queue* ([LockQueue]), corresponding to a list of currently locked critical sections ordered according to their priority ceilings. The specification is given by a number of formulas, describing **Take** (**Take_winlock**, **Take_1**, **Take_2**), as well as **Give** (**Give_owner**, **Give_1**, **Give_winlock**, **Give_2**, **Give_3**, **Give_lock_queue**, **Give_4**, **Give_5**). Finally, as stated in Section 3.1, **Take** and **Give** can also modify the scheduling of tasks. This point is illustrated by **Sched_take_2**, which is the scheduling version of **Take_2**.

An auxiliary constant, ↕EvCeiling, and two boolean functions, TakeOccurs and GiveOccurs, have been defined, which are described hereafter:

↕EvCeiling = {↑Take, ↓Take, ↑Give, ↓Give, ↓winlock, ↓¬winlock }

TakeOccurs (th_a, cs_a) = ([Flow] = ↑Take(cs_a) ∧ th_a = [Running] ∧ ceiling (cs_a) ≥ static_prio (th_a))

GiveOccurs (th_a, cs_a) = ([Flow]=↑Give(cs_a) ∧ th_a=[Running] ∧ [Owner]=th_a ∧ cs([Owner])=cs_a)

↕EvCeiling is a set containing the ceiling events (↑Take, ↓Take, ↑Give, ↓Give, ↓winlock, ↓¬winlock). TakeOccurs expresses that a **Take** operation (↑Take) on a critical section cs_a is permitted only if the ceiling of cs_a is higher than or equal to the static priority of the running thread. GiveOccurs states that a critical section cs_a cannot be released (↑Give) unless the running thread is the owner of the lock ([Owner] = th_a), and the current locked critical section is cs_a (cs ([Owner]) = cs_a).

The operation **Take** is described by **Take_winlock**, **Take_1** and **Take_2**:

Take_winlock

$\underline{\text{Always}}$ [TakeOccurs (th_a, cs_a) ∧ [Owner] ≠ ∅ ⇒
$\underline{\text{Next}}_{event}{}^{\updownarrow EvCeiling}$ [([Flow] = ↓winlock (th_a) ∨
[Flow] = ↓¬winlock (th_a)) ∧ [Running] = th_a]]

Take_1

$\underline{\text{Always}}$ [TakeOccurs (th_a, cs_a) ∧ ([Owner] = ∅ ∨
$\underline{\text{Next}}_{event}{}^{\updownarrow EvCeiling}$ [[Flow] = ↓winlock (th_a) ∧ [Running] = th_a]) ⇒
$\underline{\text{Next}}_{event}{}^{\updownarrow EvCeiling}$ [[Flow] = ↓Take(cs_a) ∧ [Running] = th_a ∧
cs_a ∈ [LockQueue] ∧ [Owner] = th_a]]

Take_2

$\underline{\text{Always}}$ [TakeOccurs (th_a, cs_a) ∧ [Owner] ≠ ∅ ∧
$\underline{\text{Next}}_{event}{}^{\updownarrow EvCeiling}$ [[Flow] = ↓¬winlock (th_a) ∧ [Running] = th_a] ⇒
$\underline{\text{Sometime}}$ [[Flow] = ↑wait ∧ [Running] = th_a] →
$\underline{\text{Next}}_{event}$ [[Flow] = ↓context_switch ∧ prio ([Owner]) = prio (th_a)
∧ th_a ∈ [BlockQueue] ∧
¬∃cs_t ∈ [LockQueue]: thread (cs_t) = th_a]]

Take_winlock states that whenever the running thread th_a initiates a **Take** operation on a critical section cs_a, and the lock is not free ([Owner] ≠ ∅), by the occurrence of the next ceiling event ($\text{Next}_{event}{}^{\updownarrow EvCeiling}$), either th_a has gained the lock (↓winlock), or it has not (↓¬winlock). **Take_1** expresses that if the lock is free ([Owner] = ∅) or th_a gains the lock (↓winlock), at the next ceiling event th_a exits the **Take** operation (↓Take) being the new owner of the lock ([Owner] = th_a), and also cs_a is inserted in the lock queue (cs_a ∈ [LockQueue]). On the contrary, **Take_2** specifies that whenever the lock is not free and th_a does not gain the lock, th_a is blocked (↑wait). Hence, by the end of the next context switch, the owner inherits th_a's priority (prio ([Owner]) = prio (th_a)), th_a is inserted in the block queue (th_a ∈ [BlockQueue]), and also it must be true that th_a was not running within a critical section (¬∃cs_t ∈ [LockQueue]: thread (cs_t) = th_a]).

The specification of **Give** is given by the next formulas:

Give_owner

$\underline{\text{Always}}$ [[Flow] = ↑Give (cs_a) ∧ th_a = [Running] ⇒
[Owner] = th_a ∧ cs ([Owner]) = cs_a]

Give_1

$\underline{\text{Always}}$ [GiveOccurs (th_a, cs_a) ∧ [BlockQueue] = ∅ ⇒
$\underline{\text{Next}}_{event}{}^{\updownarrow EvCeiling}$ [[Flow] = ↓Give (cs_a) ∧ [Running] = th_a ∧
[Owner] = thread (highest ([LockQueue]))]]

Give_winlock

$\underline{\text{Always}}$ [GiveOccurs (th_a, cs_a) ∧ [BlockQueue] ≠ ∅
∧ th_b = highest ([BlockQueue]) ⇒
$\underline{\text{Next}}_{event}{}^{\updownarrow EvCeiling}$ [([Flow] = ↓winlock(th_b) ∨
[Flow] = ↓¬winlock(th_b)) ∧ [Running] = th_a]]

Give_2

$\underline{\text{Always}}$ [GiveOccurs (th_a, cs_a) ∧ [BlockQueue] ≠ ∅ ∧
th_b = highest ([BlockQueue]) ∧ prio (th_b) = prio (th_a) ∧
$\underline{\text{Next}}_{event}{}^{\updownarrow EvCeiling}$ [[Flow] = ↓winlock (th_b) ∧
[Running] = th_a] ⇒
$\underline{\text{Sometime}}$ [[Flow] = ↑signal (th_b) ∧ [Running] = th_a] →
$\underline{\text{Next}}_{event}$ [[Flow] = ↓context_switch ∧
cs (th_b) ∈ [LockQueue] ∧ [Owner] = th_b ∧
th_b ∉ [BlockQueue] ∧ [Running] = th_b]]

Give_3

$\underline{\text{Always}}$ [GiveOccurs (th_a, cs_a) ∧ [BlockQueue] ≠ ∅ ∧ th_b =
highest ([BlockQueue]) ∧ prio (th_b) < prio (th_a) ∧
$\underline{\text{Next}}_{event}{}^{\updownarrow EvCeiling}$ [[Flow] = ↓winlock (th_b) ∧ [Running] = th_a] ⇒
$\underline{\text{Sometime}}$ [[Flow] = ↑signal (th_b) ∧ [Running] = th_a] →
$\underline{\text{Next}}_{event}$ [[Flow] = ↓ Give(cs_a) ∧ [Running] = th_a ∧
cs (th_b) ∈ [LockQueue] ∧ [Owner] = th_b ∧
th_b ∉ [BlockQueue]]]

Give_lock_queue

$\underline{\text{Always}}$ [GiveOccurs (th_a, cs_a) ∧ [BlockQueue] ≠ ∅ ∧
$\underline{\text{Next}}_{event}{}^{\updownarrow EvCeiling}$ [[Flow] = ↓¬winlock] ⇒
[LockQueue]-cs_a ≠ ∅ ∧ [Running] = th_a]

Give_4

$\underline{\text{Always}}$ [GiveOccurs (th_a, cs_a) ∧ [BlockQueue] ≠ ∅ ∧
th_b = highest ([BlockQueue]) ∧
$\underline{\text{Next}}_{event}{}^{\updownarrow EvCeiling}$ [[Flow] = ↓¬winlock (th_b) ∧ [Running] = th_a ∧
prio (th_b) = prio (th_a)] ⇒
$\underline{\text{Next}}_{event}{}^{\updownarrow EvCeiling}$ [[Flow] = ↓Give(cs_a) ∧ [Running] = th_a ∧
[Owner] = th_a ∧ prio ([Owner]) = prio (th_b)]]

Give_5

Always [GiveOccurs (th$_a$, cs$_a$) $\wedge$ [BlockQueue] $\neq \varnothing$ $\wedge$
th$_b$ = highest ([BlockQueue]) $\wedge$
Next$_{event}{}^{\updownarrow EvCeiling}$ [[Flow] = $\downarrow\neg$winlock (th$_b$) $\wedge$ [Running] = th$_a$ $\wedge$
prio (th$_b$) < prio (th$_a$)] $\Rightarrow$
Next$_{event}{}^{\updownarrow EvCeiling}$ [[Flow] = $\downarrow$Give (cs$_a$) $\wedge$ [Running] = th$_a$ $\wedge$
([Owner] = th$_a$ $\vee$
([Owner] $\neq$ th$_a$ $\wedge$ [Owner] = thread (highest ([LockQueue]))
$\wedge$ prio ([Owner]) = prio (th$_b$)))]]

Give_owner states that every time the running thread th$_a$ releases a critical section cs$_a$, th$_a$ must be the owner of the lock and cs$_a$ must be the current locked critical section. Whenever the latter is true and the block queue is empty (**Give_1**), the owner is equal to the ready thread holding the critical section with the highest priority ceiling ([Owner] = thread (highest ([LockQueue]))). Conversely, if there are threads in the block queue, once cs$_a$ is released, the highest priority blocked thread tries to become the owner (**Give_winlock**). **Give_2** specifies the case where the running thread th$_a$ is blocking a thread th$_b$ (i.e., th$_a$ had inherited th$_b$'s priority) and th$_b$ gains the lock. In this case, th$_b$ is extracted from the block queue (th$_b$ $\notin$ [BlockQueue]), its critical section is inserted in the lock queue (cs (th$_b$) $\in$ [LockQueue]), and th$_b$ becomes the newly running thread. However, if th$_b$ is being blocked by another thread (and hence th$_b$'s priority is lower than th$_a$'s priority, as described by **Give_3**), th$_a$ keeps running. Further on, consider the case where the highest priority blocked thread th$_b$ does not gain the lock. This means that, apart from cs$_a$, there is at least one more critical section in the lock queue, as described by **Give_lock_queue**. If th$_a$ is blocking th$_b$ (**Give_4**), this means that th$_b$ did not gain the lock because of a nested critical section being locked by th$_a$. Hence, th$_a$ remains the owner and enters the nested critical section inheriting th$_b$'s priority. On the other hand, the priority of the blocked thread th$_b$ might be lower than th$_a$'s priority, i.e., th$_b$ is not being blocked by th$_a$ (**Give_5**). In this case, as long as th$_a$ does not hold any nested locked critical sections, the owner is substituted by the highest priority thread currently executing inside a critical section ([Owner] = thread (highest ([LockQueue]))), which inherits th$_b$'s priority.

Take_2, **Give_2** and **Give_3** also modify the scheduling state, since they involve a number of scheduling operations, as indicated by events ↑signal, ↑wait, and ↓context_switch. Consider **Sched_take_2**, which is the scheduling version of **Take_2**:

Sched_take_2

Always [TakeOccurs (th$_a$, cs$_a$) $\wedge$
Next$_{event}{}^{\updownarrow EvCeiling}$ [[Flow] = $\downarrow\neg$winlock (th$_a$) $\wedge$ [Running] = th$_a$] $\Rightarrow$
Sometime [[Flow] = $\uparrow$wait $\wedge$ [Running] = th$_a$ $\wedge$
th$_r$ = highest ([ReadyQueue]-th$_a$)] $\rightarrow$
Next$_{event}$ [[Flow] = $\downarrow$context_switch $\wedge$ th$_a$ $\notin$ [ReadyQueue] $\wedge$
[Running] = th$_r$]]

Sched_take_2 has the same antecedent as **Take_2**. However, its consequent specify the state of scheduling variables (e.g., ReadyQueue), instead of PCP variables.

3.3.2 Internal synchronization

Microkernels use three different kinds of locks for internal synchronization, namely: mutex locks (Mutext_lock), the scheduling lock (Sched_lock), and the interrupt lock (Int_lock). A mutex lock is a binary semaphore of general use, which can be open or closed. It is used to prevent a region in the microkernel from being entered by more than one thread at a time. The scheduling lock is a special mutex that controls access to the scheduling resource. When the scheduling lock is closed, the running thread cannot be preempted. Even though it provides mutual exclusion between threads, it does not prevent interrupts from getting the CPU. Finally, the interrupt lock avoids interrupts from occurring, and is equivalent to disabling interrupts at the processor level. Therefore, Int_lock is the most stringent lock mechanism, whereas Mutex_lock is the least one. This relation can be represented in the following way:

$$\text{Mutex\_lock} \supset \text{Sched\_lock} \supset \text{Int\_lock}$$

Lock_1, **Lock_2**, and **Lock_3** specify the least stringent lock that the kernel must be using at the occurrence of an event (if needed).

Lock_1

Always [[Flow] =$\uparrow$Take $\vee$ [Flow] =$\downarrow$Take $\vee$ [Flow] =$\uparrow$Give $\vee$
[Flow] =$\downarrow$Give $\vee$ [Flow] =$\downarrow$winlock $\vee$ [Flow] =$\downarrow\neg$winlock $\Rightarrow$
[Lock] $\subseteq$ Mutex_lock]

Lock_2

Always [[Flow] = $\downarrow$context_switch $\vee$ [Flow] = $\uparrow$signal $\vee$
[Flow] = $\uparrow$wait $\vee$ [Flow] = $\uparrow$yield $\vee$ [Flow] = $\uparrow$Relinquish $\vee$
[Flow] = $\downarrow$Relinquish $\vee$ [Flow] = $\uparrow$waitFromDelay $\Rightarrow$
[Lock] $\subseteq$ Sched_lock]

Lock_3

Always [[Flow] = $\downarrow$TimeoutSet $\vee$ [Flow] = $\downarrow$TimeoutCancel $\Rightarrow$
[Lock] $\subseteq$ Int_lock]

Lock_1 specifies than whenever Take, Give (both start and ending), ↓winlock, or ↓¬winlock events occur, the kernel must be at least running under the mutex lock, but it could also be running under the scheduling or interrupt locks. **Lock_2** encompasses those events whose corresponding actions use some scheduling-related data, as the priority of the running thread. Hence, the scheduling events (↓context_switch, ↑signal, ↑wait, and ↑yield), and other events, such as ↑Relinquish, ↓Relinquish, and ↑waitFromDelay, require preemption to be disable. Finally, ↓TimeoutSet and ↓TimeoutCancel (**Lock_3**) are triggered from the clock interrupt handler, which accesses time-related queues (e.g., the TimeoutQueue), so interrupts are required to be disabled (specially the clock interrupt, which is the highest priority interrupt).

4 Verification of the timer properties

To illustrate the verification and fault tolerance capabilities offered by the specification, an instance of the Chorus/ClassiX r3.1 microkernel [11] has been verified against the timer properties defined in Section 3.2 (**Timer_1** and **Timer_2**). The workload used is a general periodic task, τ_A, whose pseudo-code is shown in Figure 1.

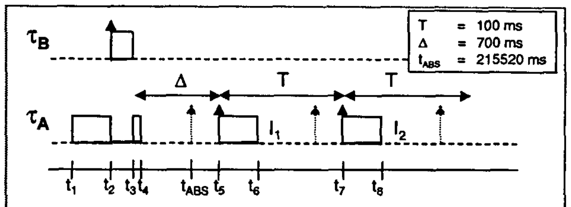
```
1.  task body Thread is
2.  begin
3.    Initialize ();
4.    set_timer (tm, tABS, T);
5.    loop
6.      wait_next_release ();
7.      Periodic_Code ();
8.    end loop;
9.  end Thread;
```

Figure 1. Periodic task

The task first initializes (line 3) and executes the `set_timer` system call, which requests to the kernel to set a periodic timer tm, with absolute start time t_{ABS}, and period equal to T (line 4). Next, it enters a loop where the task first suspends until its next release (line 6) and then executes its periodic code (line 7). Each release of the task is referred to as *instance* (I_i).

Suppose that a higher priority task, τ_B, preempts task τ_A while it is executing `set_timer`, i.e., before the call returns. Tasks τ_A and τ_B have been run on Chorus[2] along with the timer formulas, and the messages issued during their verification are shown in Figure 2.

At time t_2, the kernel computes the first release time of τ_A, namely, Δ, as the difference between t_{ABS} and the current time t_2. Unfortunately, task τ_A is preempted at time t_2, right after the kernel has computed Δ. As a result, once τ_A resumes at t_3, the kernel works out τ_A's first release as the current time t_3 plus Δ. This means that τ_A's instances are executed out of pace. This behavior leads to the violation of **Timer_1** at time t_3, as shown in lines 9 and 17, which can be viewed as the detection of a kernel error. Line 9 notifies that Δ (labeled as `delta` in Figure 2) has been given an erroneous value, whereas line 17 (some microseconds later) warns about timer tm_a being inserted in an incorrect timeout queue. The verification of the timer properties in Chorus allows this behavior to be successfully detected. Otherwise, if not taken into account, such a behavior is prone to cause missed deadlines or inexplicable delays in data transmission.



```
1.  Violation at [214893488 us]
2.        -- [Formula = Timer_1]
3.        -- [tha = 11]
4.        -- [tma = 30961816]
5.        -- [abs = 21552 ticks]
6.        -- [period = 10 ticks]
7.        -- [offset = 0 ticks]
8.        -- [SysTicks = 21489]
9.        -- [delta==abs+offset-[SysTicks]-1]:
             [70 == 62]
10. Violation at [214894028 us]
11.       -- [Formula = Timer_1]
12.       -- [tha = 11]
13.       -- [tma = 30961816]
14.       -- [abs = 21552 ticks]
15.       -- [period = 10 ticks]
16.       -- [ticks = 62]
17.       -- [tma in [TimeoutQ(ticks)]==TRUE]:
             [0 == 1]
```

Figure 2. Verification of the timer properties in Chorus (Note that the diagram is not to scale)

As illustrated in this section, our approach provides error detection in the time domain. It is worth noting however that error recovery mechanisms could also be implemented to let the kernel run in a graceful degraded mode. From a performance point of view, the overhead of verifying **Timer_1** on task τ_A was 232 µs. Further measurements are currently being carried out, but the first results obtained show that the overhead is limited.

5 Conclusion

In this paper, we have provided a method for building robust real-time microkernels based on formal specifications. The specifications consist of an abstract model of the kernel behavior that describes some of the essential services offered by any real-time kernel, namely, scheduling, time, synchronization, and clock interrupt management. Kernel services and their corresponding properties are specified in temporal logic. The specification is split into a set of temporal logic formulas that can be verified at runtime. The violation of a formula entails the detection of an error, which can further lead to error recovery actions so as to put the kernel in a safety state. Actually, the specification and the related model checker correspond to a functional and timing wrapper of the kernel. A microkernel encapsulated with such a wrapper leads to the notion of robust real-time microkernel. A very positive aspect of the method is also that such wrappers can be easily customized and ported to various COTS microkernels.

[2] `set_timer` and `wait_next_release` correspond to `timerSet` and `timerThreadPoolWait` in Chorus, resp.

The specification presented in this paper is being extended and used for the verification and hardening of microkernels, presently the Chorus/ClassiX microkernel as a first target candidate. The efficient implementation of wrappers based on formal specifications is also a current subject of research. From our first experiments, the overhead introduced by the verification of the temporal logic formulas by the runtime model checker is very limited. Moreover, this overhead can be taken into account in the design of upper level real-time applications, i.e., included in the real-time development process. It is worth noting that we are also addressing issues concerning the impact of wrappers in both hard and soft real-time systems, where predictability and performance are of primary importance. The assessment of the robustness of the resulting microkernels encapsulated with formal wrappers will be done using fault injection techniques [12] and tools [13].

Finally, the method and the experimental fault injection environment will be used to improve and assess the robustness of other real-time kernels currently used in industrial safety critical systems, in particular in the avionics domain.

Acknowledgements. Manuel Rodríguez is supported by Thomson-CSF (SODETEG). The work reported in this paper was carried out in the framework of LIS[3]. Thanks go to Scott Hazelhurst (Wits University, South Africa) who provided useful insights on temporal logic in the early phase of this work, during his stay at LAAS. Comments from François Scheerens (SODETEG) are gratefully acknowledged.

References

[1] S. Hazelhurst, J. Arlat, "Specifing and verifying fault-tolerant hardware", France-South Africa Cooperation, LAAS report No99514, 1999.

[2] A. Burns, A. J. Wellings, "Safety Kernels: Specification and Implementation", *High Integrity Systems*, vol. 1, no. 3, pp. 287-300, 1995.

[3] S. Edwards, L. Lavagno, E. A. Lee, A. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models, Validation, and Synthesis", *Proceedings of the IEEE*, vol. 85, no. 3, pp. 366-390, 1997.

[4] S. Fowler, A. J. Wellings, "Formal Analysis of a Real-Time Kernel Specification", *4th International Symposium on Formal Tecniques in Real-Time and Fault Tolerant Systems*, 1996.

[5] J. Gorski, A. Wardzinski, "Formal Specification and Verification of a Real-Time Kernel", *Proceedings of 6th Euromicro Workshop on Real-Time Systems, pp.205-211*, 1994.

[6] J. M. Spivey, "Specifying a Real-Time Kernel", *IEEE Software*, vol. 5, no. 7, pp. 21-28, September 1990.

[7] F. Salles, M. Rodríguez, J.-C. Fabre, J. Arlat, "MetaKernels and Fault Containment Wrappers", *29th IEEE International Symposium on Fault-Tolerant Computing (FTCS-29)*, pp. 22-29, Madison, Wisconsin, USA, 1999.

[8] S. Hazelhurst, C.-J. H. Seger, "Symbolic Trajectory Evaluation", in *Formal Hardware Verification: Methods and Systems in Comparison, State of the Art Survey Lecture Notes in Computer Science 1287*, T. Kropf, Ed.: Springer-Verlag, pp. 3-79, 1997.

[9] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, A. J. Wellings, "Fixed Priority Pre-emptive Scheduling: An Historical Perspective", *Real-Time Systems*, vol. 8, no. 3, pp. 173-198, 1995.

[10] L. Sha, R. Rajkumar, J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175-1185, 1990.

[11] Chorus, "Chorus/ClassiX r3 - Technical Overview", Technical Report CS/TR-96-119.8, Chorus Systems, 1996.

[12] J.-C. Fabre, F. Salles, M. Rodríguez, J. Arlat, "Assessment of COTS Microkernels by Fault Injection", *7th IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA'99) - Can we rely on computers?*, San Jose, California, USA, 1999.

[13] M. Rodríguez, F. Salles, J.-C. Fabre, J. Arlat, "MAFALDA: Microkernel Assessment by Fault Injection and Design Aid", *3rd European Dependable Computing Conference (EDCC-3)*, Prague, Czech Republic, 1999.

[3] Located at LAAS, the Laboratory for Dependability Engineering (LIS) was a Cooperative Laboratory between five industrial companies (Aerospatiale Matra Airbus, Électricité de France, Matra Marconi Space-France, Technicatome, Thomson-CSF) and LAAS-CNRS.