

# Formal Verification of Multitasking Applications Based on Timed Automata Model

**Libor Waszniowski, Zdeněk Hanzálek**

Centre for Applied Cybernetics, Department of Control Engineering,  
Faculty of Electrical Engineering, Czech Technical University,  
Karlovo nám. 13, 121 35 Prague 2, Czech Republic,  
{xwasznio, hanzalek}@fel.cvut.cz

**Abstract.** The aim of this paper is to show, how a multitasking application running under a real-time operating system compliant with an OSEK/VDX standard can be modeled by timed automata. The application under consideration consists of several non-preemptive tasks and interrupt service routines that can be synchronized by events. A model checking tool is used to verify time and logical properties of the proposed model. Use of this methodology is demonstrated on an automated gearbox case study and the result of the worst-case response time verification is compared with the classical method based on the time-demand analysis. It is shown that the model-checking approach provides less pessimistic results due to a more detailed model and exhaustive state-space exploration.

**Keywords:** Formal methods, Verification, Model-checking, Timed automata, OSEK/VDX, Multitasking

## 1 Introduction

This paper deals with formal modeling of applications running under real-time operating system (OS). The typical application under assumption, shown as a case study in Section 7, is a complex controller consisting of periodic and aperiodic tasks constrained by deadlines and synchronized via inter-task communication primitives. The objective is to use model-checking approach (Larsen, *et al.*, 1995), (Berard, *et al.*, 2001) for automatic verification of the model described in this paper.

The model based on timed automata (Alur and Dill, 1994) considers an operating system, application tasks and a controlled environment behavior. It assumes a *fine grain model* of the task internal structure consisting of computations, OS calls, selected variables, code branching and loops. Therefore the model combines both, logic and timing characteristics of the discrete event system enabling one to check rather complex properties (safety and bounded liveness properties, state reachability or schedulability) by model checking tools (e.g. UPPAAL (Larsen, *et al.*, 2001) and Kronos (Daws, *et al.*, 1996)) in finite time. Deadlock freeness of the application, occurrence of the race condition during access to shared data structures, a concrete value of some essential variable under certain conditions, end to end response time of

an arbitrary event, proper ordering and timing of events in the control application or the controlled environment can be verified, for example.

Due to the composability of timed automata, models produced by different authors can be directly combined together. For example, a single processor system model can be simply expanded to a distributed system model by adding a communication layer model (Krákora, *et al.*, 2004).

Even though timed automata and model-checking (analogous to other formal methods) allow one to model and verify almost everything, it is generally known, that they are susceptible to the state space explosion. This fact restricts the size of verified application to a small size that seems to be unusable in practice (compared with matured response time analysis methods (Klein, *et al.*, 1993) or the offset-based analysis proposed by Palencia and Harbour (1998)). Therefore we try to show in this paper, how to build a compromised model of a reasonable size on one side and reasonable granularity on the other side, allowing for a detailed formal analysis of real-time properties that can not be made by response time analysis.

Methods for response time analysis based on time-demand analysis (Buttazzo, 1997), (Klein, *et al.*, 1993), (Liu, 2000) are well known and used in practice. These methods, e.g. rate monotonic analysis (RMA) (Sha, *et al.*, 1991), are straightforward for systems with independent periodic tasks but incorporation of non-periodic tasks and inter-task communication primitives can lead to pessimistic results (Bailey, *et al.*, 1995). This is caused by limited information inherent in the simple model of a task consisting of the worst-case execution time (WCET), the worst-case inter-arrival period and the worst-case blocking by lower-priority tasks. Analysis of end-to-end response time of transactions in a distributed system has been successfully solved by Tindell and Clark (1994) and later extended by Palencia and Harbour (1998). Also, these approaches however do not consider the detailed model of the controlled environment and the tasks internal structure as our approach does.

The response time analysis based on an exhaustive analysis of the fine grain model provides more precise (less pessimistic) results in some cases as is shown in Section 8. The price paid for this is higher memory requirements and time complexity of the model-checking method. Therefore the model-checking-based response time analysis cannot be seen as a universal method but as a less pessimistic and more demanding alternative to classical scheduling theory-based methods.

This paper focuses on a non-preemptive scheduling since tasks consisting of non-preemptive blocks of code can be modeled by timed automata, for which effective verification algorithms based on symbolic and reduction methods (see e.g. Larsen, *et al.*, 2003) exist.

Modeling of preemptive tasks has been studied by Corbet in (1996). This work provides a method for constructing models of real-time Ada tasking programs based on constant slope linear hybrid automata. Even though the author reports that the analyzing algorithm does usually terminate in practice, the reachability problem for hybrid automata is undecidable and therefore the analyzing algorithm termination is not guaranteed in general. The termination of the timed automata model verification is guaranteed, which is the advantage of our approach.

When modeling preemption in a multitasking application, it is necessary to stop a clock variable measuring the execution time of a preempted task and remember its value until the task is scheduled again. This can be done in hybrid automata, but not in timed automata. On the other hand the reachability problem is decidable for timed automata. This is a motivation of work (Waszniowski and Hanzálek, 2005) providing a timed automata based over-approximate model of preemptive tasks. The over-approximation of the model means that besides the real behavior of the system, also some additional behavior is modeled. Therefore only properties preserved by this approximation (e.g. safety and bounded liveness properties) can be verified by a model-checking tool. Similarly, the model presented in this paper is over-approximate (in some cases, see Section 6) due to interrupts.

There are also extensions of Time Petri Nets allowing one to model systems with preemption; Preemptive Time Petri Nets (pTPN) (Bucci et al., 2004) and Scheduling Extended Time Petri Nets (SETPN) (Lime and Roux, 2004). However, states of these formalisms are represented by a general convex polyhedra and the problem of state reachability is undecidable. It has been shown in (Henzinger et al., 1998) that the problem of state reachability is undecidable for any formalisms that is expressive enough for modelling preemption. Therefore, decidable (finite state space) over-approximations that preserve safety and bounded liveness properties are usually used for verification of preemptive systems.

Timed automata are used to model primitives of Ravenscar run-time kernel for Ada in (Lundqvist and Asplund, 2003). However, the variable used to measure the execution time of tasks (modeling the system clock) is an integer, periodically incremented by a timed automaton after each “tick”. Therefore the notion of time in the application is discrete opposite to our approach where time is dense.

Discrete time for modeling a real-time application is also used in (Campos and Clarke, 1999) presenting a modeling language and a symbolic algorithm for quantitative analysis (providing minimum and maximum time between events) of synchronous real-time systems. Discrete time is also used in (Fredette and Cleaveland, 1993) where a generalized approach to schedulability analysis based on process algebra is proposed. Even though these approaches consider the task internal structure, the controlled environment affecting release times of tasks is not modeled. Our approach considers the controlled environment model.

Another interesting approach to schedulability analysis is based on timed automata extended by asynchronous tasks (i.e. tasks triggered by events) that provide a model for event-driven systems (Fersman, *et al.*, 2002), (Fersman, *et al.*, 2003). Each task specified by its execution time is associated to one timed automaton location. A transition leading to the location denotes an event releasing the task. Released tasks are stored in a queue and they are assumed to be executed according to a given scheduling strategy. The problem of the system schedulability is transformed to the reachability problem in a timed automaton. This approach provides good results for aperiodic tasks (due to the detailed model of the environment releasing the tasks) but it does not consider the task internal structure. It would be possible to model the task internal structure by splitting the task to blocks of code and assigning them to locations of extended timed automaton representing control structure of the original task. Shared variables can be used to synchronize the end of one block of code

execution, with the transition of the control structure timed automaton that starts the next block of code. However, the reachability problem of such a model is decidable only for non-preemptive scheduling or when all tasks have constant execution times (Krčál and Yi, 2004).

In recent years several approaches integrating the schedulability analysis to some formal description methods have been published. Alvarez, *et al.* in (2003) developed a method for computation of response time of tasks integrated to specification and description language (SDL). Similarly, Wang and Tsai in (2004) present an approach to extend message sequence chart (MSC) by tasks parameters, and by response time analysis. Both these methods are an application of standard response time analysis without considering the internal structure of tasks and controlled environment.

This paper is organized as follows: Section 2 gives an overview on the basic features of OSEK compliant operating system. Readers familiar with OSEK specification do not need to read this section. Section 3 describes the *fine grain model* used in this paper. Sections 4, 5, and 6 presents the main results of this paper – timed automata models of tasks, OSEK compliant OS kernel and interrupt service routine (ISR). Section 7 demonstrates the proposed approach on an automated gearbox case study and Section 8 compares the task's response time analysis made by the model-checking approach and by the classical time-demand analysis. The paper is concluded with Section 8.4.

## 2 OSEK/VDX overview

This section surveys the basic features of an operating system compliant with OSEK/VDX Operating System specification, version 2.2.3 (OSEK, 2005) (further called OSEK). OSEK is a simple static multitasking singleprocessor executive for electronic control units (ECU) used in automotive applications. Small memory demand requires simple services, which can be modeled by timed automata of reasonable size. All objects of the system are created in compilation time. Therefore they can be modeled by timed automata and static data structures.

### 2.1 Task management

OSEK provides static priority based, preemptive and non-preemptive scheduling (OSEK, 2005), but we consider only non-preemptive scheduling in this paper. Even though OSEK distinguishes basic and extended tasks, we consider only extended ones, since basic tasks are only a subset variant of extended ones and both are modeled in the same way.

Tasks, created as *suspended* at the system generation time, become *ready* after activation by the OS service *ActivateTask* called from ISR or another task. The highest priority ready task starts *running*. The running task may terminate its execution by calling the service *TerminateTask* and become suspended or it may voluntarily relinquish the processor by calling the service *Schedule* and become ready. If there is no higher-priority ready task, calling of the service *Schedule* does not affect the task execution. Extended tasks are, moreover, allowed to use the system call *WaitEvent*,

which may result in a *waiting* state. To become ready, the waiting task requires the event (which it is waiting for) to be set.

## 2.2 Event management

OSEK provides an event management for task synchronization. The event is represented by one bit in a byte assigned to an extended task - the event's owner. The event is therefore identified by its owner and its name (or mask specifying more than one event). The event owner may wait for the event and clear the event (services *WaitEvent* and *ClearEvent*). All tasks may set or get the binary value of a non-suspended task event (services *SetEvent* and *GetEvent*).

## 2.3 Resource management

Resource management is used to coordinate the access of several tasks (and interrupt service routines) to the critical section. The resource access protocol is used to provide mutually exclusive access, to prevent priority inversion and deadlock. According to this protocol, the priority ceiling is statically assigned to the resource at the system generation time. Its value is equal to the highest priority of all tasks (or ISR) accessing the resource. At run time, the priority of the task occupying the resource is increased to the resource priority ceiling. Task priority is reset to the previous value after releasing the resource. Consequently, no task (or ISR) ever tries to access the occupied resource and therefore no task can be blocked on the resource (notice that the OSEK specification does not allow any blocking OS services inside the critical section).

In non-preemptive scheduling, the mutually exclusive access of several tasks to the critical section is provided just by its non-preemptability and by the restriction of calling the OS services *Schedule*, *WaitEvent* and *TerminateTask* from the critical section. Simultaneous access of a task and an ISR to the critical section can be prevented by disabling interrupts within the critical section or by using resources.

## 2.4 Interrupt management

OSEK distinguishes interrupt service routines (ISRs) of category 1 that do not use any OS services (no influence on the task management) and ISR of category 2 allowing all OS services except some services dedicated entirely to tasks (*WaitEvent*, *TerminateTask*). There is no difference between both categories from the modeling point of view. When a task with priority higher than the interrupted one is activated by an OS service called from the ISR, the interrupted task is not preempted due to non-preemptive scheduling. Even when the processor is idling, when an interrupt occurs, no rescheduling takes place at the OS service called from the ISR, but the OS service only changes states of tasks, and the rescheduling takes place at the end of the ISR. Therefore, when several tasks are activated in the ISR, it does not depend on their activation order, but the highest priority one is scheduled at the end of the ISR (when no task is currently running).

### 3 Fine grain model of multitasking application

The *fine grain model* treats the internal structure of the tasks and the interrupt service routines (ISR), the OS functionality and the controlled environment behavior. All components are modeled by timed automata synchronized via channels and by shared variables. The task model consists of several blocks of code called *computations*, calls of OS services, selected variables, code branching and loops (affected by values of selected variables). *Computations* are defined by the BCET (the best-case execution time) and the WCET (the worst-case execution time). Considering the execution time as an interval  $\langle \text{BCET}, \text{WCET} \rangle$ , it allows one to incorporate the uncertainty of the execution time due to non-modeled code branching inside the *computations*, cycle stealing by a DMA device, etc. When a general property of the model is analyzed by an exhaustive state space search (made by a model checking tool), the execution time of a task must be specified by an interval covering all possible cases, i.e.  $\langle \text{BCET}, \text{WCET} \rangle$ . Due to the possibility of a scheduling anomaly, the WCET of *computations* does not necessarily lead to the worst-case response time of the whole task.

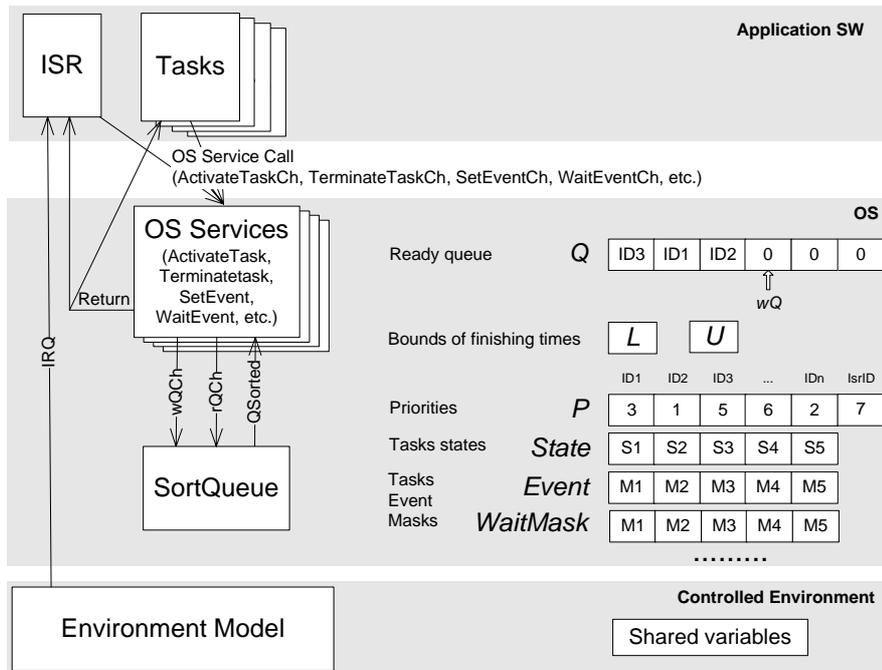


Figure 3.1 Overview of entire timed automata model

The structure of the entire model is shown in Figure 3.1. Rectangular blocks represent particular timed automata (e.g. task automaton in Figure 4.1 b) or OS service automaton in Figure 5.3). Synchronization of timed automata is expressed by arcs labeled by the name of the synchronization channel (*ActivateTask*, *EndSysCall*, etc.). The most important data structures (e.g. *Q*, *P*, *State*) are shown on the right side of the figure. The essential components are explained in Sections 4, 5 and 6.

Since the schedulability is one of the most often analyzed properties, it is attractive to compare our *fine grain model* based on timed automata with the classical scheduling theory task model based on the WCET of the whole task, its minimum inter-arrival time and its blocking time related to resources (Buttazzo, 1997), (Sha, *et al.*, 1991), (Liu, 2000). Classical response time analysis based on such model computes the worst case finishing time of the task by adding together its worst case execution time, duration of preemption by higher-priority tasks in the worst case inter-arrival times and phasing, and the worst case blocking by lower-priority tasks on shared resources. Such worst-case finishing time is a conservative abstraction of all possible finishing times but it could be a too pessimistic abstraction in many applications, since all the mentioned worst cases do not occur at the same time (Bailey, *et al.*, 1995).

Exhaustive analysis of *fine grain model* behavior (automatically completed by model checking tool UPPAAL) considers the task finishing times corresponding to the realistic phasing, the realistic blocking and the realistic execution time in relation to the modeled code branching. Therefore, the result is as precise as the model. The price paid for the exhaustive analysis is higher complexity.

We will demonstrate the advantage of the model containing the tasks internal structure in a simple example. Let us consider two tasks, higher-priority  $Task_1$  and lower-priority  $Task_2$  listed in Figure 3.2.  $Task_1$  is activated with a period 12. Depending on the variable *data*, it performs either *LongComputation* taking 8 time units or *ShortComputation* taking only 2 time units.  $Task_2$  is activated if and only if *ShortComputation* is executed.  $Task_2$  execution takes 10 time units.

```

Task1
{
  if (Data==OK)
  {
    LongComputation;
  }
  else
  {
    ShortComputation;
    ActivateTask (Task2);
  }
}

Task2
{
  Computation;
}

```

Figure 3.2 Tasks pseudocode

Figure 3.3 shows that both tasks are finished prior to their next activation in both cases,  $Data==OK$  and  $Data!=OK$ .

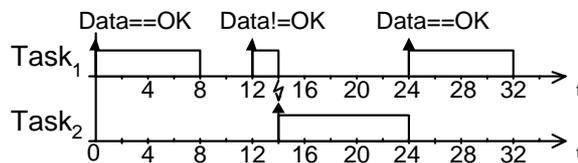


Figure 3.3 Schedule considering tasks internal structure

Let us analyze the application by demand analysis (Liu, 2000) based on a simple model considering only the WCET and the period of tasks. Without knowledge of

$Task_1$ 's internal structure, we must consider its  $WCET_1$  to be equal to 8 and activation of  $Task_2$  must be considered at each period of  $Task_1$ .

In the case of preemptive scheduling,  $Task_1$  is always finished 8 time units after the beginning of the period, but the worst-case response time of  $Task_2$  is  $10+3*8=34$  (see Figure 3.4), which exceeds its period.

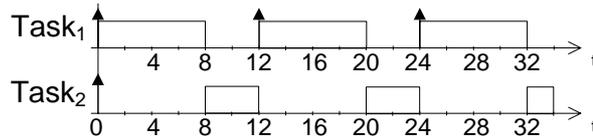


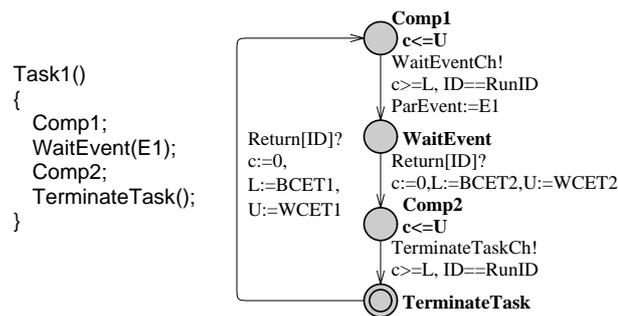
Figure 3.4 Schedule ignoring tasks internal structure

In the case of non-preemptive scheduling, the worst-case response time of both tasks is  $10+8=18$ , which is longer than the activation period.

#### 4 Task model

Each task instance is modeled by one timed automaton that is synchronized with the OS model via channels depicted as arrows in Figure 3.1. Figure 4.1 demonstrates the modeling methodology in the example of a simple task executing *computations*  $Comp1$  and  $Comp2$  and calling OS services  $SetEvent(task, event)$  and  $TerminateTask$ .

Timed automata are depicted in the UPPAAL notation (Larsen, *et al.*, 2001), where the location with double circles represents the initial location. Each location can be labeled by its name and time invariant (both in bold font). Invariant in the form " $c \leq U$ ", allows it to stay in the location only when the valuation of the clock variable  $c$  is smaller than or equal to integer  $U$ . Each transition can be labeled by synchronization (channel name with '?' or '!'), guard (logical terms separated by a comma, e.g.,  $c \geq L[1], State[1] == RUNNING$ ) and assignment (assignments using the sign ':=' separated by a comma).



a) Pseudo-code                      b) Task timed automaton

Figure 4.1 Simple task example

Each *computation* is represented by one location of the same name (e.g.  $Comp1$ ). Time spent in this location (measured by clock  $c$ ) represents the *computation's*

finishing time (i.e. time necessary for its execution including interrupts) and is bounded by values stored as integer variables  $L$  and  $U$  (provided by time invariant  $c \leq U$  and guard  $c \geq L$ ). Clock  $c$  is reset and variables  $L$  and  $U$  are initialized by the BCET and the WCET of the *computation* on the transition incoming at the corresponding location (e.g.  $c:=0, L:=BCETI, U:=WCETI$  on the transition leading to *CompI*). Variables  $L$  and  $U$  are increased when the task is interrupted (provided by timed automaton modeling ISR; see section 6). Notice that due to non-preemptive scheduling, only one clock  $c$  can be shared by all task timed automata, which considerably reduces the size of the state space. The guard  $ID \neq RunID$  prevents the task timed automaton from progressing when it is not scheduled (i.e. an ISR is executed). Constant  $ID$  is a unique identifier (0,1,2,...) of tasks and ISR.

OS service call is modeled by the transition synchronized via the channel of the corresponding name (e.g. *WaitEventCh!*) with the automaton modeling the OS service functionality, and by the location of the corresponding name (e.g. *WaitEvent*) in which the task is waiting a return from the service (channel *Return[ID]?*). OS service parameters, if they are required, are delivered through shared variables *ParTask* and *ParEvent*. Notice that some OS services (e.g. *Schedule* or *WaitEvent*) can cause rescheduling. In this case, the return from the called service occurs after finishing all higher-priority tasks.

Realize that the task can be running or interrupted when its model is in a location corresponding to a computation. When the task model is in location corresponding to an OS service, OS executes the service and then the task can continue its execution (e.g. service *SetEvent*), or the task is blocked while waiting for an external event (service *WaitEvent*), or the task is ready to execute but a higher priority task is executed (service *Schedule*) or the task is suspended (service *TerminateTask*). Even though the task code in Figure 4.1 a) is linear (it does not contain any loops), the timed automaton in Figure 4.1 b) is cyclic since the suspended task (task timed automaton is in the location *TerminateTask*) can be activated and start its execution from the beginning. This is also the reason why location *TerminateTask* is the initial one.

## 5 OS kernel model

The OS kernel model consists of integer variables representing the OS objects (e.g. ready queue), the timed automata representing OS services functionality, and the timed automaton sorting the ready queue according to priorities (*SortQueue*). See the model overview in Figure 3.1.

### 5.1 Kernel variables

The task priority is stored in a global array  $P$ , indexed by  $ID$ . Higher number represents higher priority.  $P$  can be a constant when the resource management is not modeled.

The task state is stored in the array *State* at the index corresponding to task  $ID$ . The task state is either *SUSPENDED*, *WAITING*, *READY* or *RUNNING*. However it is necessary to distinguish only the state *SUSPENDED* from all others in the proposed

model. Therefore the symbols *WAITING*, *READY* or *RUNNING* have the same value in the model.

The variable *RunID* stores *ID* of the currently running task or interrupt service routine.

*IDs* of all tasks, which are ready for execution, are stored in the ready queue modeled as a global array *Q* (see Figure 3.1). *IDs* of tasks are stored at the lowest positions in the array. The variable *wQ* contains the first empty position in *Q*. Tasks are ordered in descending order according to their priorities in *Q*. The ready task with the highest priority is always at the position zero; the ready task with the lowest priority is always at the position *wQ*-1.

The queue must be reordered according to tasks priorities after writing a new task and all elements of the queue must be shifted to the left after reading the highest priority ready task from the zero position. Both these mechanisms are provided by the automaton *SortQueue* depicted in Figure 5.1.

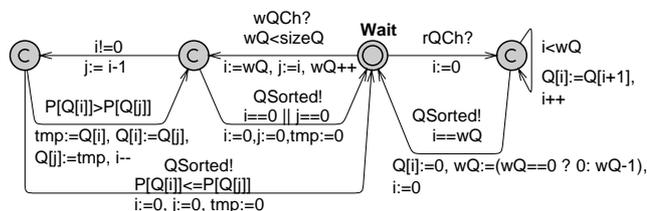


Figure 5.1 *SortQ* automaton

The reordering mechanism is started by the synchronization channel *wQCh* after writing a new *ID* to *Q[wQ]*. The pointer *wQ* is then increased and priorities of tasks in neighboring position in *Q* are compared (started from *wQ*) and if there is a higher priority task in a higher position, *IDs* are swapped. The termination of this mechanism is announced by the channel *QSorted*. Shifting of *Q* after reading *Q[0]* is started by the synchronization channel *rQCh* and its finishing is announced by the channel *QSorted*.

Notice that it would be possible to implement the ready queue as a circular buffer. The top of the queue would not always be at position zero, but it would be pointed by the pointer (lets call it *rQ*) that is increased after reading the highest priority task. It is not necessary to shift elements of *Q* in this case. A circular buffer would, therefore, be a more elegant approach from the programming point of view, but it is not appropriate for verification purposes, since such a model generates a bigger state space. Realize that two different configurations of a circular buffer containing the same tasks but stored in different positions (different *rQ* and *wQ*) are represented by two different states in the state space, but they represent the same situation from the application point of view. Contrary to that, all situations when *Q* contains the same tasks are represented by only one state in our approach, since the same set of tasks is always stored in the same position in *Q* (from zero to *wQ*-1).

Events are represented by the integer array *Event* associating one byte *Event[ID]* to each task. Each bit in *Event[ID]* represents one event that can be set or cleared.

Moreover, the integer array *WaitMask* represents events, which the corresponding task is waiting for.

Variables *L* and *U* are necessary for model execution of the task code. As it has been already stated in Section 4, the variables are used to store lower and upper margins of the finishing time of the current started *computation*. These integers are initialized by the BCET and WCET respectively, at the start of the *computation* and they are increased by an ISR BCET and WCET respectively when an interrupt occurs (provided by a timed automaton modeling the ISR).

## 5.2 OS services

Each OS service is modeled by a timed automaton representing its functionality defined by OSEK specification (OSEK, 2005). The automaton is waiting in its initial location until its function is called from the task model (by synchronization via the corresponding channel e.g. *WaitEventCh*). Then it manipulates the tasks' states, ready queue and other operating system objects (e.g. events). OS services *Schedule*, *WaitEvent* and *TerminateTask* can, moreover, reschedule the current tasks. This is done by choosing the highest priority ready task and storing its *ID* in the variable *RunID*. The next *computation* of the *RunID* task is then started by taking a transition synchronized by channel *Return[RunID]*. Models of all OSEK services can be found on: <http://dce.felk.cvut.cz/waszniewski/RTVerif/RTVerif.htm>

As an example of a service model, we introduce *WaitEvent(Mask)* service that causes the task to wait for events specified by *Mask*. Figure 5.2 shows *WaitEvent* OS service functionality in pseudo-code. The corresponding automaton is in Figure 5.3. It is supposed that interrupts are disabled within the whole service. Locations marked by “C” are so called committed locations. The committed location is left immediately without any interference from another automaton that is not in committed location. Since all locations in the OS services automata, except the initial one, are committed locations, the whole service is atomic from the point of view of the tasks and the controlled environment models. The execution time of the OS services is involved in the execution times of *computations* calling them.

```

WaitEvent (Mask)
{
  if ((Event[RunID] & Mask) == 0)
  {
    State[RunID] := WAITING;
    WaitMask[RunID] := Mask;
    RunID := Extract Top of ReadyQ;
    ContextSwitch;
    State[RunID] := RUNNING;
  }
  return E_OK;
};

```

Figure 5.2 *WaitEvent* OS service pseudo-code

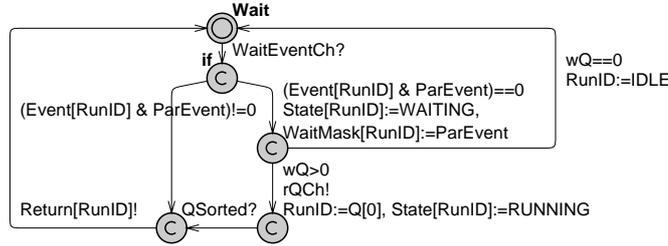


Figure 5.3 WaitEvent OS service automaton

## 6 Interrupt service routine model

In this section we present a timed automata model of an ISR. The ISR can interrupt the execution of a task. It is modeled by increasing the bounds of the interrupted task execution time (stored in variables  $L$ ,  $U$ ) by the bounds of the ISR execution time ( $BCET\_ISR$ ,  $WCET\_ISR$ ). We show that this approach brings an over-approximation to the model.

For reasons of simplicity, only one ISR is assumed to exist in this article, but it can be generalized when all hardware details are considered. The ISR is modeled by a timed automaton modeling application dependent code in the same way as the task code. Moreover, there is an initialization part preventing a task scheduling inside the ISR.  $RunID$  is stored in the local variable  $InterruptedID$  and the  $ID$  of the ISR ( $IsrID$ ) is written to the variable  $RunID$ . Therefore the OS services called from the  $ISR$  do not schedule any task (because  $RunID \neq IDLE$ ). Values of  $L$  and  $U$  are increased by the  $BCET\_ISR$  and  $WCET\_ISR$  respectively. Further, there is a finalization part providing task scheduling at the end of the ISR (as it is required by OSEK specification (OSEK, 2005)). Either  $InterruptedID$  or, if it is equal to  $IDLE$ , the  $ID$  from the top of the ready queue is written to the variable  $RunID$ . An example of ISR pseudocode is in Figure 7.4 and the corresponding timed automaton is in Figure 7.10.

Lets us explore the approach for taking the ISR execution time into account in the interrupted task execution time. When the interrupt occurs the execution time bounds of the interrupted *computation* (stored in variables  $L$  and  $U$ ) should be prolonged by the duration of the ISR execution. Since the right duration of the interruption cannot be measured in timed automata (a clock variable cannot be stopped or stored), the bounds  $L$  and  $U$  are increased by bounds of the possible ISR execution time  $BCET\_ISR$  and  $WCET\_ISR$ . This introduces an additional non-determinism to the model since the modeled duration of the interrupted task interruption is not necessary equal to the ISR execution (what holds in the real system). Therefore the set of real system behaviors is a subset of the modeled behaviors, i.e. the model is an over-approximation of the real system.

To illustrate the over-approximation let us consider for example a *computation* of task  $T$  with the execution time  $C_T \in [1, 4]$  interrupted by an ISR with the execution time  $C_{ISR} \in [2, 4]$ . All possible relative finishing times of the interrupted *computation* versus

the ISR execution time in the real system and in the proposed model are depicted in Figure 6.1. Finishing time of the interrupted *computation* is equal to its execution time  $C_T$  plus interruption duration. Interruption duration is bounded by bounds of  $C_{ISR}$  in the model but it is equal to the actual execution time of *ISR* in the real system.

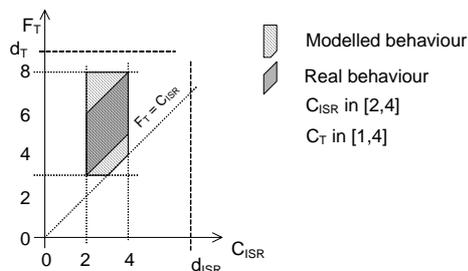


Figure 6.1 Possible values of relative finishing time  $F_T$  of the interrupted *computation* of the task  $T$  versus the *ISR* execution time  $C_{ISR}$

Figure 6.1 shows that not all modeled behaviors can occur in the real system. It is very important to keep this fact in mind during the verification process, since the over-approximation does not preserve a general property. It means that it cannot be automatically concluded that a general property satisfied by the model is also satisfied by the real system. On the other hand, it is important from the practical point of view, that over-approximation preserves safety and bounded liveness properties (Berard, *et al.*, 2001). A safety property states that, under certain conditions, an undesirable event never occurs. A bounded liveness property states that, under certain condition, some desirable event will occur within some deadline. See examples in Section 7.

Please realize that the model is over-approximate only in the case that the  $WCET_{ISR}$  differs from the  $BCET_{ISR}$ .

Schedulability is an often verified property, exploring whether *computations* are finished prior to their deadlines ( $d_{ISR}$  and  $d_T$  in Figure 6.1) in all situations. Figure 6.1 shows that the worst case finishing time of the task or *ISR* is the same in the model and in the real system. A result of the schedulability analysis based on this model is therefore correct and corresponds to reality (it is not pessimistic).

## 7 Gear box Case Study

### 7.1 System description

The proposed modeling methodology is demonstrated on an automated gearbox control system. The controlled system consists of a five-speed gearbox and a dry clutch. The gearbox mechanics are depicted in Figure 7.1. They consist of three shift rails and a shift finger actuated by *SelectServo* and *ShiftServo*. *SelectServo* can move the shift finger from a slot of one rail to another one. *ShiftServo* engages one of two gears (odd or even) or neutral by moving the selected rail by the shift finger. The direction of the shift finger movement is limited by a gait.

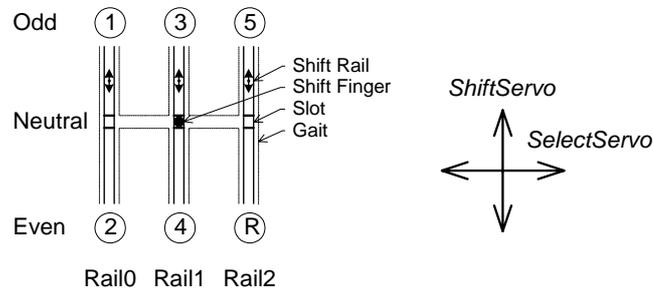


Figure 7.1 Gear box mechanism

The gearbox is controlled by a single processor control unit running an OSEK compliant OS. The application software consists of three tasks (*SlipCtrlTask*, *SelectGearTask*, *GearBoxCtrlTask*) and one ISR.

The ISR (see pseudocode in Figure 7.4) is periodically invoked by a timer (with the period 10) and by the clutch, *ShiftServo* or *SelectServo* when their position changes. The source of the interrupt is specified by bits *bTimerInt*, *bClutchInt*, *bShiftServoInt* and *bSelectServoInt*. According to the source of the interrupt, tasks are activated (*bTimerInt*) or an event is set (*bClutchInt*, *bShiftServoInt* and *bSelectServoInt*).

Task *SlipCtrlTask* (see pseudocode in Figure 7.2) is periodically activated by ISR. Its priority is 2 and its period is 10. It provides slip control and torque tracking but its detailed functionality is not relevant to verification, therefore, it is not considered here. Only its computation time is modeled.

Task *SelectGearTask* (see pseudocode in Figure 7.3) is periodically activated by ISR. Its priority is 0 and its period is 500. It selects the appropriate transmission rate, writes it to the variable *DesiredGear*, and if the desired gear differs from the current one, it activates task *GearBoxCtrlTask* that controls changing of the gear. Also the model of this task is very rough.

```
SlipCtrlTask() // Activated periodically
{
  // Slip control, torque tracking, clutch protection
  if (ClutchState == CLOSED)
    CompSlipCtrl;
  TerminateTask();
};
```

Figure 7.2 Slip control task pseudocode

```
SelectGearTask() // Activated periodically
{
  if (GBReady)
    // Select gear according to current car conditions
    CompDesiredGear;
    if (DesiredGear != CurrentGear)
      ActivateTask (GearBoxCtrlTask);
}
TerminateTask();
};
```

Figure 7.3 Select Gear task pseudocode

```

ISR()    // IRQ sources - TIMER, Clutch, ShiftServo, SelectServo
{
    // Initialization part – for modeling purposes
    InterruptedID := RunID;
    RunID := IsrID;
    U += U_IsrID;    // Prolong Interrupted task by constants U_IsrID and L_IsrID
    L += L_IsrID;

    // User defined code
    while (bTimerInt || bClutchInt || bShiftServoInt || bSelectServoInt)
    {
        Comp;
        if (bTimerInt)
        {
            bTimerInt:=0;
            clk:=(clk>MAX_CLK ? 1 : clk+1);
            if (clk% SlipCtrlTaskPeriod == 0)
                ActivateTask(SlipCtrlTask);
            if (clk% SelectGearTaskPeriod == 0)
                ActivateTask(SelectGearTask);
        }
        else if (bClutchInt)
        {
            bClutchInt:=0;
            SetEvent (GearBoxCtrlTask, ClutchEvent);
        }
        else if (bShiftServoInt)
        {
            bShiftServoInt:=0;
            SetEvent (GearBoxCtrlTask, ShiftServoEvent);
        }
        else if (bSelectServoInt)
        {
            bSelectServoInt:=0;
            SetEvent (GearBoxCtrlTask, SelectServoEvent);
        }
    }

    // Finalization part (Scheduling point) – for modeling purposes
    if (InterruptedID != IDLE)
    {
        RunID := InterruptedID;
        Return to Interrupted Task;
    }
    else if (ReadyQueue is Empty)
    {
        RunID := IDLE;
        Return to IDLE task;
    }
    else
    {
        RunID := Extract Top of ReadyQ;
        Return to RunID task;
    }
};

```

*Figure 7.4* Interrupt service routine pseudocode

```

GearBoxCtrlTask() // Activated by SelectGearTask
{
  GBReady := 0;
  ClearEvent (ClutchEvent);
  OpenClutch; // Send command to clutch
  WaitEvent (ClutchEvent);
  if (CurrentShift != NEUTRAL)
  {
    // Disengage
    ClearEvent (ShiftServoEvent);
    ShiftServo_Goto (NEUTRAL); // Send command to ShiftServo
    WaitEvent (ShiftServoEvent);
  }
  if (DesiredGear != NEUTRAL)
  {
    // Select shifting rail
    DesiredRail := (DesiredGear-1)/2; // integer division
    DesiredShift := (DesiredGear-1)%2+1; // modulo operation
    if (DesiredRail != CurrentRail)
    {
      // Select
      ClearEvent (SelectServoEvent);
      SelectServo_Goto (DesiredRail); // Send command to SelectServo
      WaitEvent (SelectServoEvent);
    }
    // Shift
    ClearEvent (ShiftServoEvent);
    ShiftServo_Goto (DesiredShift); // Send command to ShiftServo
    WaitEvent (ShiftServoEvent);
  }
  ClearEvent (ClutchEvent);
  CloseClutch; // Send command to clutch
  WaitEvent (ClutchEvent);
  GBReady := 1;
  CurrentGear:=DesiredGear;
  TerminateTask();
};

```

Figure 7.5 Gear Box Control task pseudocode

Task *GearBoxCtrlTask* (see pseudocode in Figure 7.5) has priority 1. It sends a command to open the clutch first, then it waits for the event *ClutchEvent* signaling that the clutch is open. If NEUTRAL is not currently engaged, it disengages the current gear by sending the command to the *ShiftServo* to move the shift finger to the neutral position and waits for the event *ShiftServoEvent*. Then the new gear, stored in variable *DesiredGear*, can be engaged. First, the rail and shift direction corresponding to the *DesiredGear* are computed and stored in the variable *DesiredRail* (Rail 0, 1, or 2) and *DesiredShift* (ODD or EVEN). Then if the *DesiredRail* is not currently selected, the command to the *SelectServo* is sent to move the shift finger to the position of the *DesiredRail*. When the *DesiredRail* is selected (signalled by the event *SelectServoEvent*) the *DesiredGear* is engaged by sending the command to the *ShiftServo* to move the selected rail to *DesiredShift* position. After finishing the *ShiftServo* movement (signalled by the event *ShiftServoEvent*), the command to close the clutch is sent and when the clutch is closed (signaled by the event *ClutchEvent*), the variable *CurrentGear* is updated and the task is terminated.

## 7.2 Model

A model of the whole system consists of timed automata representing the controlled system (*Clutch*, *SelectServo* and *ShiftServo*), a hardware of a control unit (periodic timer generating interrupts), an OS (services *ActivateTask*, *TerminateTask*, *SetEvent*, *WaitEvent* and automaton *SortQueue*), three application tasks (*SlipCtrlTask*, *SelectGearTask*, *GearBoxCtrlTask*) and one ISR. An overview of the whole model is depicted in Figure 7.6 and Figure 7.7. Figure 7.6 shows the timed automata synchronization via the channels and Figure 7.7 shows events and variables shared by the timed automata in the model. The variables and timed automata modeling OS have been explained in the previous section, therefore they are omitted in both figures.

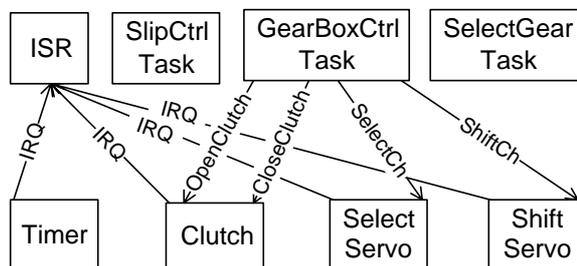


Figure 7.6 Model overview – timed automata (rectangles) and synchronizations via channels (arrows)

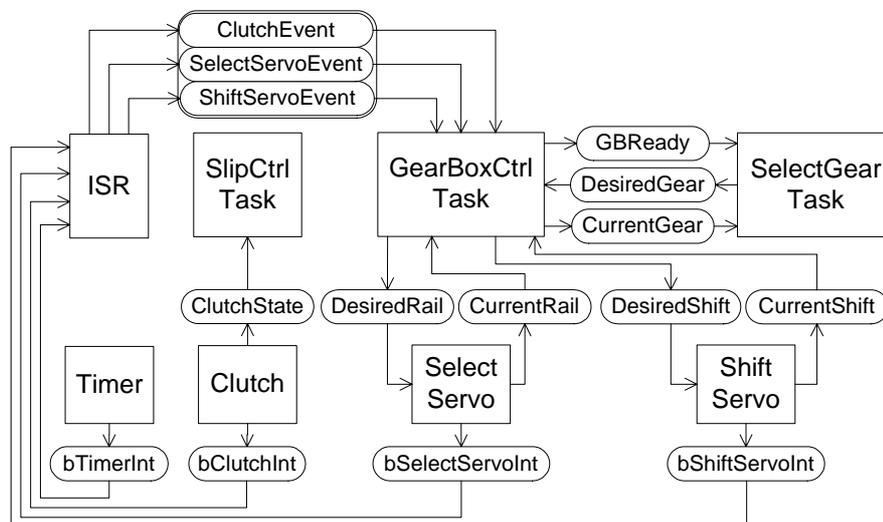


Figure 7.7 Model overview – timed automata (rectangles) and shared variables (ovals)

Since the *Timer* timed automaton is very simple, it is not depicted here. It only waits in its initial location and periodically generates interrupt via the channel *IRQ* and the variable *bTimerInt*.

The *Clutch* timed automaton is depicted in Figure 7.8. It is in the location *Closed* or *Opened* in a steady state. When the *Clutch* receives the command to open or close (via channel *OpenClutch* or *CloseClutch* respectively), it moves to *Opening* or *Closing* respectively. After the time bounded by *ShiftTime* and *ShiftTimeU*, the *Clutch* reaches a new steady state and generates an interrupt request (IRQ) via the channel *IRQ*.

The *SelectServo* timed automaton is depicted in Figure 7.9. The automaton is in one location corresponding to *Rail0*, *Rail1* or *Rail2* in a steady state. When the command to select a new rail is received via the channel *SelectCh*, the *SelectServo* automaton moves to the *DesiredRail* (*BetweenRail0andRail1* and *BetweenRail1andRail2*). When the *DesiredRail* is reached, an IRQ is generated via the channel *IRQ*.

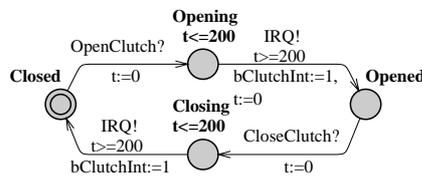


Figure 7.8 Clutch timed automaton

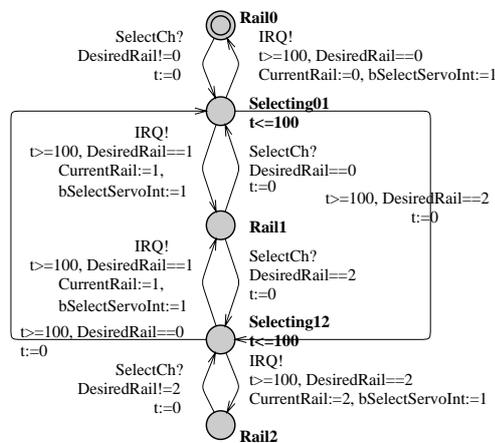


Figure 7.9 SelectServo timed automaton

The *ShiftServo* timed automaton (not depicted here) differs from the *SelectServo* timed automaton only in a few details. Locations *Rail0*, *Rail1*, *Rail2*, *BetweenRail0andRail1* and *BetweenRail1andRail2* are replaced by *OddPos*, *NeutralPos*, *EvenPos*, *BetweenNeutralAndOdd* and *BetweenNeutralAndEven*. Variables and channels related to selecting (*DesiredRail*, *CurrentRail*, *SelectCh*, *bSelectServoInt*) are replaced by variables and channels related to shifting (*DesiredShift*, *CurrentShift*, *ShiftCh*, *bShiftServoInt*). Moreover, the time spent in locations *BetweenNeutralAndOdd* and *BetweenNeutralAndEven* is not 100 time units but 200 time units.

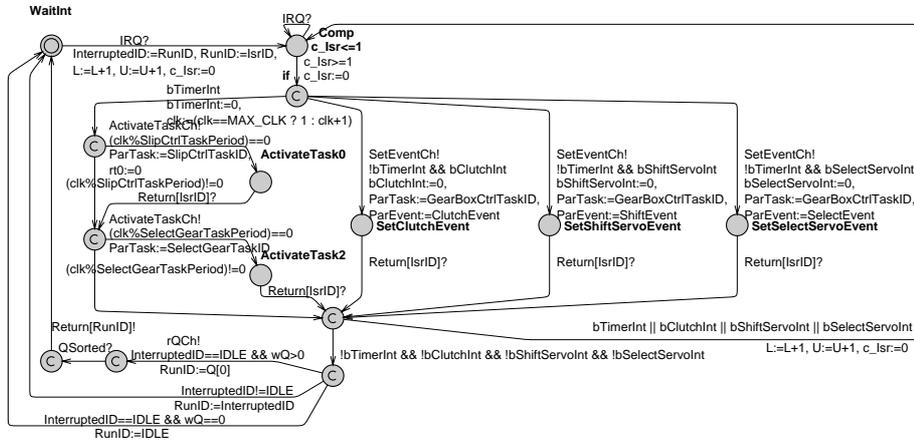


Figure 7.10 ISR timed automaton

Timed automata modeling tasks *SlipCtrlTask*, *SelectGearTask* and *GearBoxCtrlTask* and *ISR* function are depicted in Figure 7.11, Figure 7.12, Figure 7.13 and Figure 7.10, respectively. They have been obtained by translating tasks pseudocodes from Figure 7.2, Figure 7.3, Figure 7.5 as well as Figure 7.4 to timed automata according to the methodology described in Sections 4 and 6, respectively.

Notice that the computation of the *DesiredGear* in the *SelectGearTask* is modeled by non-deterministic choice in the *SelectGearTask* timed automaton. Therefore, all possibilities are explored by the model-checking tool. Notice also that the OS service *ClearEvent* is very simple ( $Event[ID] := Event[ID] \& !ClutchEvent$ ); it is not modeled by a special automaton but it is modeled directly in the *GearBoxCtrlTask* timed automaton (see Figure 7.13).

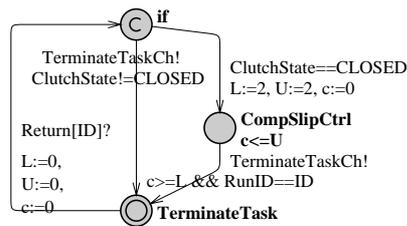


Figure 7.11 SlipCtrlTask timed automaton

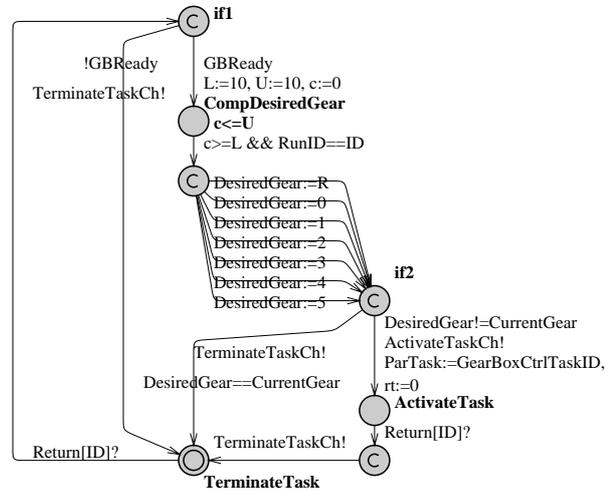


Figure 7.12 SelectGearTask timed automaton

### 7.3 Formal verification

The following properties are required for proper function of the system:

*Safety properties:*

- P1. Shifting is allowed only when the clutch is open
- P2. Selecting is allowed only when the shift servo is in neutral
- P3. Shifting is allowed only when a rail is selected
- P4. Clutch cannot be open longer than 650 time units

*Bounded liveness:*

- P5 – P11. When new desired gear (NEUTRAL, 1...5, R) is selected, it is engaged in 1020 time units

*Deadlock-freeness:*

- P12. The system is deadlock free

Notice that since multiple tasks activation is disabled in the model, deadlock-freeness (P12) guarantees that all tasks are finished prior to their new activation in the next period. Therefore the property P12 can be interpreted as schedulability (deadlines at the end of periods).

Please realize that the deadlock-freeness is not proved to be preserved by the over-approximation described in Section 6. It can be therefore verified only in the case of the WCET\_ISR equal to the BCET\_ISR that holds in this model.

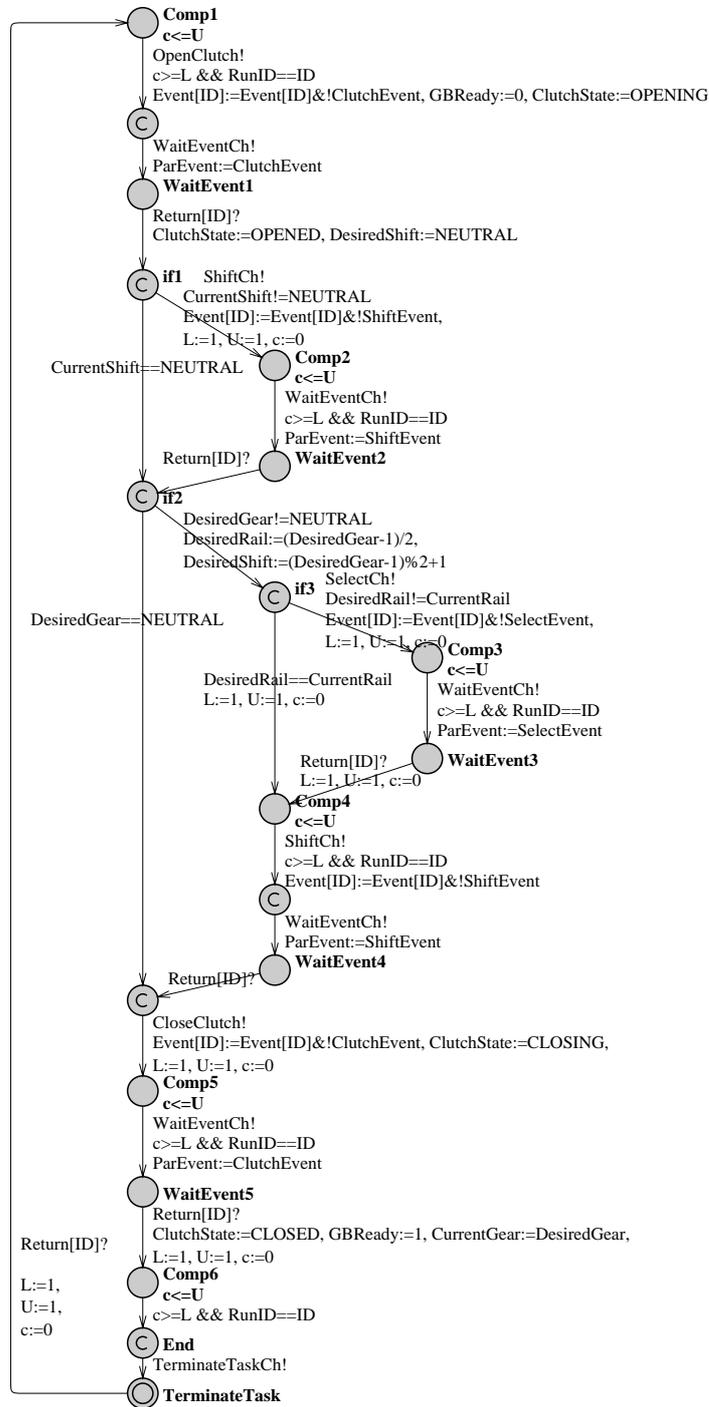


Figure 7.13 GearBoxCtrlTask timed automaton

The above listed properties have been formalized in UPPAAL requirement specification language as follows:

P1.  $A[] \text{Clutch.Closed} \text{ imply } (\text{ShiftServo.OddPos} \text{ or } \text{ShiftServo.NeutralPos} \text{ or } \text{ShiftServo.EvenPos})$

P2.  $A[] \text{ not } (\text{SelectServo.Rail0} \text{ or } \text{SelectServo.Rail1} \text{ or } \text{SelectServo.Rail2}) \text{ imply } \text{ShiftServo.NeutralPos}$

P3.  $A[] \text{ not } \text{ShiftServo.NeutralPos} \text{ imply } (\text{SelectServo.Rail0} \text{ or } \text{SelectServo.Rail1} \text{ or } \text{SelectServo.Rail2})$

P4.  $A[] \text{Clutch.Opened} \text{ imply } \text{Clutch.t} \leq 650$

P5.  $(\text{DesiredGear} == 0 \text{ and } \text{SelectGearTask.ActivateTask}) \text{ -->} (\text{ShiftServo.NeutralPos} \text{ and } \text{rt1} \leq 1020)$

P6.  $(\text{DesiredGear} == 1 \text{ and } \text{SelectGearTask.ActivateTask}) \text{ -->} (\text{ShiftServo.OddPos} \text{ and } \text{SelectServo.Rail0} \text{ and } \text{rt1} \leq 1020)$

P7.  $(\text{DesiredGear} == 2 \text{ and } \text{SelectGearTask.ActivateTask}) \text{ -}$

P8 – P11 Similar to P6 and P7->  $(\text{ShiftServo.EvenPos} \text{ and } \text{SelectServo.Rail0} \text{ and } \text{rt1} \leq 1020)$

P12.  $A[] \text{ not deadlock}$

In UPPAAL requirement specification language the syntax  $A[]f$  represents the computation tree logic (CTL) formula  $\forall \square f$  (i.e. “invariantly holds  $f$ ”), and the syntax  $p \text{ -->} q$  denotes a CTL property  $\forall \square (p \Rightarrow \forall \diamond q)$  (i.e. “whenever  $p$  holds, eventually  $q$  will hold as well”). Notice that the clock  $rt1$  measuring the response time in all bounded liveness properties P5 – P11 is reset when the new *DesiredGear* is selected in the *SelectGearTask* timed automaton depicted in Figure 7.12.

All the above mentioned properties of the system have been successfully verified by model-checker UPPAAL 3.4.7 running on Windows 2000 on PC AMD Athlon 1GHz, with 1.3GB RAM. The time required for verification of all of these twelve properties is 58 seconds. The required memory is 78 MB. The most demanding properties are the bounded liveness properties. Realize that time and memory requirements drastically grow with the complexity of the model. Even though the memory requirement of 78MB is acceptable, an augmentation of the model by other tasks can easily make the verification impossible.

## 8 Response time analysis

The previous section presents the verification possibilities of the model-checking methods. One of the many properties that can be verified by model-checking is, whether a task response time satisfies its deadline. This section compares a task response time analysis based on the classical scheduling theory (Klein, *et al.*, 1993) and (Palencia and Harbour, 1998) to the one based on the model-checking approach proposed in this paper. Both these approaches are applied on the *SlipCtrlTask* and the *GearBoxCtrlTask* and the worst-case response times (WCRT) obtained by both approaches are compared. The results show, that the worst-case response times

obtained by the model-checking approach are, due to a more detailed model and exhaustive state space analysis, less pessimistic. In the case of the *SlipCtrlTask*, the advantage of the model-checking approach is based on the information contained in the controlled environment model and the model of the task internal structure. In the case of the *GearBoxCtrlTask* containing self-suspension, the advantage of the model-checking approach is based on exhaustive analysis of the model state space.

On the other hand, exhaustive analysis of the state-space (very quickly growing with the number of tasks, computations and variables) limits the size of applications for which the model-checking method can be used. Therefore, we do not want to present the model-checking approach as a universal method for response time analysis but as an alternative approach providing some reasonable advantages in some cases.

### 8.1 WCRT of the SlipCtrlTask by Scheduling theory based approach.

Let us explore the response time of the *SlipctrlTask* from its activation within the ISR (see pseudo code in Figure 7.4) till its termination. The scheduling theory based response time analysis deals with the following information.

The WCET of the *SlipCtrlTask* is 2 time units. Even though the *SlipCtrlTask* is executed at the highest priority, its response time can be prolonged by the longest non-preemptable part of any lower-priority task, and by an ISR. The longest non-preemptable block in the system is the task *SelectGearTask* whose WCET is 10. The ISR is invoked by the timer (with period 10), the clutch (with minimal inter-arrival time 200), the *SelectServo* (with minimal inter-arrival time 100), and the *ShiftServo* (with minimal inter-arrival time 200). Servicing each of the mentioned requests takes one time unit.

The WCRT of *SlipCtrlTask* in the worst-case phasing (all interrupts occurred and the longest non-preemptable block just began) is 17 ( $2+10+2*1+1+1+1$ ).

Notice that the timer ISR is considered twice, since it is invoked twice prior the *SlipCtrlTask* is finished (after 17 time units).

### 8.2 WCRT of the SlipCtrlTask by model-checking approach

Contrary to the scheduling theory based approach presented in the previous section, the model-checking approach considers only the possible phasing determined by the detailed model of the controlled environment (the gearbox mechanism: *Clutch*, *SelectServo* and *ShiftServo*) and the control algorithm (*GearBoxCtrlTask*). It is impossible in this phasing that all interrupts are invoked by the *Clutch*, *SelectServo* and *ShiftServo* simultaneously. The model-checking tool, moreover, explores the right blocking time of the *SlipCtrlTask* by lower priority tasks.

We explore the WCRT of the *SlipCtrlTask* in the following way: The response time of the task is measured by the clock variable  $rt0$ , which is reset when the *SlipCtrlTask* is activated in *ISR*, at the transition leading to the location *ActivateTask0*. Then the following property is verified: “Always, when the end of the *SlipCtrlTask* (location *End*) is reached, the inequality  $rt0 \leq WCRT$  holds”. This property is formalized in the UPPAAL requirement specification language as follows:

$A[] \text{SlipCtrlTask.End}$  imply  $rt0 \leq \text{WCRT}$ .

Then the verification is made by UPPAAL for the particular value of the WCRT. First, the value of the WCRT must be estimated by the designer and then, if the formula is satisfied, its value can be decreased. The smallest value of the WCRT can be found in several iterations. Notice that algorithms also exist for parametric model-checking verifying whether a state is reachable in a model with an uncertain parameter (the WCRT in the observer automaton). However, this problem is undecidable in general (Alur, *et al.*, 1993). The smallest value of the WCRT is therefore found by the interval bisection. The WCRT of *SlipCtrlTask* obtained by the model-checking approach is 5. This result is a significantly smaller value than in the case of the scheduling theory approach (see Section 8.1). This result is valid only in the case when the *SlipCtrlTask* is never activated more than once before it is finished. This requirement is expressed by the property  $A[] n\text{Activated}[\text{SlipCtrlTaskID}] < 1$  and it has been successfully verified in the proposed model.

Both of these properties have been successfully verified by the model-checker UPPAAL 3.4.9 running on Windows 2000 on PC AMD Athlon 1GHz, with 1.3GB RAM. The time required for verification of both of these properties is 7 seconds. The required memory is 57 MB.

### 8.3 WCRT of GearBoxCtrlTask by Scheduling theory based approach.

It is clear from the *GearBoxCtrlTask* pseudocode listed in Figure 7.5 that the task suspends itself several times while waiting for external events. This fact must be considered in the WCRT analysis since the suspended task had to compete for the processor again after the end of the suspension. The worst-case execution path of *GearBoxCtrlTask* is in Figure 8.1. It consists of six *computations* (*Comp1* to *Comp6*) separated by five self-suspensions (*WaitEvent(...)*). The WCET of all *computations* are 1 and the worst-case self-suspension times (WCSST) are 200.

```

GearBoxCtrlTask – worst-case execution path
{
  Comp1;                               // WCET1 = 1
  WaitEvent (ClutchEvent);             // WCSST1 = 200
  Comp2;                               // WCET2 = 1
  WaitEvent (ShiftServoEvent);         // WCSST2 = 200
  Comp3;                               // WCET3 = 1
  WaitEvent (SelectServoEvent);        // WCSST3 = 2*100
  Comp4;                               // WCET4 = 1
  WaitEvent (ShiftServoEvent);         // WCSST4 = 200
  Comp5;                               // WCET5 = 1
  WaitEvent (ClutchEvent);             // WCSST5 = 200
  Comp6;                               // WCET6 = 1
};

```

Figure 8.1 *GearBoxCtrlTask* worst-case execution path

Palencia and Harbour (1998) proposed an offset-based approach for response-time analysis of tasks with static and dynamic offsets that can be successfully applied to tasks with self-suspension. The response-time analysis based on this approach has been implemented in the tool MAST (González Harbour, *et al.*, 2001) that can be downloaded at <http://mast.unican.es/mast.html>. We have used this tool to compute the

WCRT of *GearBoxCtrlTask*. The WCRT of *GearBoxCtrlTask* obtained by offset-based approach is 1021.

Realize however that the offset-based approach does not consider branching affected by values of variables. Since the contribution of the higher priority task *SlipCtrlTask* to the WCRT of the *GearBoxCtrlTask* strongly depends on the value of variable *ClutchState* (see Figure 7.2), and the blocking by the lower priority task *SelectGearTask* strongly depends on value of variable *GBReady* (see Figure 7.3), the model used for analysis by the tool MAST has been manually modified in the following way: The task *SelectGearTask* is not considered in the model, since it never blocks any computation of *GearBoxCtrlTask* (*GearBoxCtrlTask* is started at the end of *SelectGearTask* and the execution of *SelectGearTask* is blocked by the value of the variable *GBReady* that is zero during the execution of the whole *GearBoxCtrlTask*). The task *SlipCtrlTask* cannot affect computations *Comp2* to *Comp5* of the task *GearBoxCtrlTask* due to value of the variable *ClutchState* (see Figure 7.5). The contribution of the task *SlipCtrlTask* is therefore involved in the WCET of computations *Comp1* and *Comp6* and the task *SlipCtrlTask* is not considered in the model.

This modification of the model prevents the pessimisms of the analysis caused by not considering the branching affected by values of variables. Realize however that such modification would be hard or even impossible for a more complex system structure and cannot be therefore understood as a systematic approach to analysis. We have done it only for a fair comparison with the model-checking approach. The WCRT of *GearBoxCtrlTask* obtained by offset-based approach without described modification is 1034.

#### 8.4 WCRT of the *GearBoxCtrlTask* by model-checking approach

We explore the WCRT of the *GearBoxCtrlTask* in a similar way as in the case of *SlipCtrlTask*. The response time of *GearBoxCtrlTask* is measured by the clock variable *rt1*, which is reset when the *GearBoxCtrlTask* is activated in *SelectGearTask*, at the transition from location *if2* to location *ActivateTask*. Then the following property is verified: “Always, when the end of *GearBoxCtrlTask* (location *End*) is reached, the inequality  $rt \leq \text{WCRT}$  holds”. This property is formalized in the UPPAAL requirement specification language as follows:

$A[] \text{GearBoxCtrlTask.End} \text{ imply } rt \leq \text{WCRT}.$

Then the verification is made by UPPAAL for the particular value of the WCRT. The smallest value of the WCRT of *GearBoxCtrlTask* is 1011. This result is valid only in the case when *GearBoxCtrlTask* is never activated more than once before it is finished. This requirement is expressed by the property  $A[] n\text{Activated}[\text{GearBoxCtrlTaskID}] < 1$  and it has been successfully verified in the proposed model.

Both of these properties have been successfully verified by model-checker UPPAAL 3.4.9 running on Windows 2000 on PC AMD Athlon 1GHz, with 1.3GB RAM. The time required for verification of both of these properties is 8 seconds. The required memory is 61 MB.

The difference between the WCRT found by model-checking (1011) and scheduling theory (1021) seems to be minor. Realize however that the biggest part of the WCRT corresponds to the duration of the self-suspension ( $5 \cdot 200$ ). The pessimism of the scheduling theory approach (1021-1011) is therefore comparable with the exact time when the task is ready or executed ( $1011 - 5 \cdot 200$ ).

## 9 Conclusions

In this paper, we have demonstrated, how timed automata can be used for modeling of multitasking, non-preemptive applications. The complex time and logical properties of the proposed model, considering the values of variables, behavior of the controlled environment and an internal structure of the control system tasks (e.g. “Shifting is allowed only when the clutch is opened”, “Desired gear is engaged in 1020 time units”, “System is deadlock free”, “A task is finished within X time units”, etc.), can be automatically verified by a model-checking tool.

Even when comparing to the task response time analysis, where classical scheduling theory can be applied, an advantage of the model-checking approach based on a fine grain timed automata model is that it considers the task internal structure and the controlled environment and it exhaustively analyzes its state space by a symbolic model-checking algorithm. Consequently, a more precise (less pessimistic) analysis is provided by the model-checking approach in the cases, when the analyzed application contains features that make the response time analysis pessimistic (e.g. branching in the tasks code, tasks self-suspension), or when the worst-case behavior, considered by the classical scheduling theory, can never occur in the controlled environment. It is clear however that the high memory requirements of the model-checking are preventing this method from becoming a universal response time analysis method.

An exhaustive analysis of the detailed timed automata model subjects to state space explosion (which is a general property of most formal methods (Corbett, 1996)). Therefore, the proposed model is abstract as much as possible and it contains only information necessary for correct verification of the system specification. The operating system model uses only modest data structures, it does not use any clock variables, it does not allow any non-determinism and all locations are committed, which prevents paths interleaving and therefore restricts the explored state space. Moreover the OS model is scalable. Therefore only features used in the modeled application are used in the OS model. Notice also that OSEK is one of the most appropriate operating systems to be modeled by timed automata since it is static (all objects are created at compilation time) and it is designed for modest runtime environment of embedded devices. For example events and resources do not require any queue of waiting tasks. The model of an application tasks must be designed as a compromise between the model precision and its state space size. It is necessary to limit the size of modeled data, non-determinism and number of *computations* in order to obtain the model of reasonable size. In spite of these restrictions, the model-checking approach is applicable for formal verification of realistic applications whose verification made manually by humans would be hard and error prone.

## Acknowledgement

We would like to thank Gerd Behrmann from the Department of Computer Science at Aalborg University, Denmark for his advice related to reducing the complexity of the ready queue model.

This work was supported by the Ministry of Education of the Czech Republic under Project 1M0567 and by Academy of Science of the Czech Republic under Project 1ET400750406.

## References

- Alur R., T. A. Henzinger and M.Y. Vardi, "Parametric real-time reasoning", *Proceedings of the 25<sup>th</sup> ACM Symposium on Theory of Computing*, pp. 592-601, 1993
- Alur, R. and D.L. Dill, "A theory of timed automata", *Theoretical Computer Science*, 126, pp. 183-235, 1994.
- Alvarez, J.M., M. Diaz, L. Lopis, E. Pimentel and J.M. Troya, "Integrating Schedulability Analysis and Design Techniques in SDL", *Real-Time Systems*, 24: pp. 267-302, 2003.
- Bailey C.M., A. Burns, A.J. Wellings and C.H. Forsyth, "A Performance Analysis of a Hard Real-Time System", *Control Engineering Practice*, 3(4), pp. 447-464, 1995.
- Berard, B., M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen and P. McKenzie, *Systems and Software Verification: Model-Checking Techniques and Tools*, Springer Verlag, 2001.
- Bucci, G., A. Fedeli, L. Sassoli and E. Vicario, "Timed State Space Analysis of Real-Time Preemptive Systems", *IEEE Transaction on Software Engineering*. 30(2): 97-111, 2004.
- Buttazzo, G., C., *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers, Boston, 1997.
- Campos, S., and E. Clarke, "Analysis and Verification of Real-time Systems Using Quantitative Symbolic Algorithms", *Journal of Software Tools for Technology Transfer*, Vol.2, n.3, pp.260-269, 1999.
- Corbett, J. C., "Timing analysis of Ada tasking programs", *IEEE Transactions on Software Engineering*, 22(7), pp. 461-483, July 1996.
- Daws, C., A. Olivero, S. Tripakis and S. Yovine, "The tool Kronos", In: *Proceedings of Hybrid Systems III, Verification and Control*, LNCS 1066, pp. 208-219. Springer-Verlag, New York, 1996.
- Fersman, E., P. Pettersson, and W. Yi, "Timed Automata with Asynchronous Processes: Schedulability and Decidability", In: *Proceedings of 8<sup>th</sup> International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2002*, LNCS 2280, pp.67-82, Springer-Verlag, 2002.
- Fersman, E., P. Pettersson, and W. Yi, "Schedulability Analysis using two clocks", In: *Proceedings of TACAS'03*, LNCS 2619 pp. 224-239. Springer-Verlag, 2003.
- Fredette, A.N. and R. Cleaveland, "RTSL: A Language for Real-Time Schedulability Analysis", In: *Proceedings of the Real-Time Systems Symposium*. pp 274--283, IEEE Computer Society Press, 1993.

- M. González Harbour, J.J. Gutiérrez García, J.C. Palencia Gutiérrez, and J.M. Drake Moyano, "MAST: Modeling and Analysis Suite for Real Time Applications", In *Proceedings of 13<sup>th</sup> Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, IEEE Computer Society Press, pp. 125-134, June 2001.
- Henzinger, T., P. Kopke, A. Puri, and P. Varaiya, 'What's decidable about hybrid automata?'. *Journal of Computer and System Sciences* 57, 94–124, 1998.
- Klein, M., T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, *A Practitioner's Handbook for Real-Time Systems Analysis*, Kluwer Academic Pub., 1993.
- Krčkora, J., L. Waszniowski, P. Piša and Z. Hanzálek, "Timed Automata Approach to Real Time Distributed System Verification", In *Proceedings of 5<sup>th</sup> IEEE International Workshop on Factory Communication Systems, WFCS*, Vienna, September 22-24, 2004, pp. 407-410.
- Krčál, P. and W. Yi, "Decidable and Undecidable Problems in Schedulability Analysis Using Timed Automata", In: *Proceedings of TACAS'04*, LNCS 2988, pp 236-250. Springer-Verlag, 2004.
- Larsen, K.G., P. Pettersson, and W. Yi, "Model-Checking for Real-Time Systems", In *Proceedings of the 10<sup>th</sup> International Conference on Fundamentals of Computation Theory*, LNCS 965, pp. 62-88. Springer Verlag, 1995.
- Larsen, K.G., P. Peterson and W. Yi, "UPPAAL in a nutshell", *International Journal on Software Tools for Technology Transfer*, 1(1/2):134-152, 1997.
- Larsen, K.G., F. Larson, P. Pettersson and W. Yi, "Compact Data Structures and State-Space Reduction for Model-Checking Real-Time Systems", *Real-Time Systems*, 25: pp. 255-275, 2003.
- Lime, D and O.H. Roux, "A translation based method for the timed analysis of scheduling extended time Petri nets", In *Proceedings of the 25<sup>th</sup> IEEE International Real-time Systems Symposium*, 187--196, December 2004, Lisbon, Portugal, 2004.
- Liu, J.W.S., *Real-time systems*, Prentice-Hall, Inc., Upper Saddle River, New Jersey 2000.
- Lundqvist, K. and L. Asplund, "A Ravenscar-Compliant Run-time Kernel for Safety-Critical Systems", *Real-Time Systems Journal*, 24(1): pp. 29-54, 2003.
- OSEK, *OSEK/VDX Operating System Specification 2.2.3*, <http://www.osek-vdx.org/>, 2005.
- Palencia, J.C. and G.Harbour, "Schedulability Analysis for Tasks with Static and Dynamic Offsets", In *Proceedings of the 19<sup>th</sup> IEEE Real-Time Systems Symposium*, pp. 26. IEEE Computer Society Press, 1998.
- Sha, L., M. Klein and J. Goodenough, "Rate Monotonic Analysis for Real-Time Systems", *Foundations of Real-Time Computing: Scheduling and Resource Management*, pp. 129-155, Kluwer Academic Publishers, Boston, 1991.
- Tindell, K. and J. Clark, "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems", *Microprocessing & Microprogramming*, Vol. 50, No. 2-3, pp. 117-134, April 1994.
- Wang, S. and G.Tsai, "Specification and Timing Analysis of Real-Time Systems", *Real-Time Systems*, 28, pp. 69-90, 2004.
- Waszniowski, L. and Z. Hanzálek, "Over-approximate Model of Multitasking Application Based on Timed Automata Using Only One Clock", In *Proceedings 19th IEEE International Parallel and Distributed Processing Symposium IPDPS*

2005 (Workshop 2: Parallel and Distributed Real-Time Systems), pp. 128, IEEE Computer Society, 2005.



**Libor Waszniowski** graduated in technical cybernetics from the Czech Technical University in Prague (CTU) in 2000 and received the PhD degree in electrical engineering and informatics from CTU in 2006. Currently, he is a researcher with the Center for Applied Cybernetics at CTU. His main research activities are in formal methods for real-time systems design and analysis and rapid application development.



**Zdeněk Hanzálek** is a member of the IEEE and the IEEE Computer Society. He obtained the Diploma in Electrical Engineering from the Czech Technical University (CTU) in Prague in 1990. He obtained his PhD degree in Control Engineering from the CTU in Prague and PhD degree in Industrial Informatics from the Université Paul Sabatier Toulouse. He was with LAAS - Laboratoire d'Analyse et d'Architecture des Systemes in Toulouse (1992 to 1997) and with LAG INPG - Institut National Polytechnique de Grenoble (1998 to 2000). In 2005, he obtained Doc. degree at the Czech Technical University in Prague. His research interests include real-time systems and scheduling.