Analysis of Real Time Operating System Based Applications

Libor Waszniowski, Zdenek Hanzalek

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

Abstract. This text is dedicated to modelling of real-time applications running under multitasking operating system. Theoretical background is based on timed automata by Alur and Dill. As this approach is not suited for modelling preemption we focus on cooperative scheduling. In the addition, interrupt service routines are considered, and their enabling/disabling is controlled by interrupt server considering the specified server capacity. The server capacity has influence on the margins of the computation times in the application processes. Such systems, used in practical real-time applications, can be modelled by timed automata and further verified since their reachability problem and model checking of TCTL problem is decidable. Use of this methodology is demonstrated on the case study.

1 Introduction

The aim of this article is to show, how timed automata [1] can be applied to modelling of real time software applications running under operating system with cooperative scheduling. The application under consideration consists of several process, it includes mechanisms for interrupt handling, and it uses inter-process communication primitives like semaphores, queues etc. Model checking theory based on timed automata and implemented in model checking tools (e.g. UPPAAL[2]) can be used for verifying time parameters or safety and liveness properties of proposed models.

Timing analysis of software (especially with concurrency and synchronisation) is not trivial problem and it requires sophisticated methods and analysis tools. Several special purpose methods have been developed in the area of real time scheduling [3],[7]. These methods, e.g. rate monotonic analysis (RMA) [4], are very successful for analysis of systems with periodic processes. To deal with non-periodic processes, the standard method is to consider the non-periodic process as the periodic one using the minimal inter-arrival time as process period. The analysis based on such model is too pessimistic in some cases since inter-arrival times can vary over time [13]. Incorporation of inter-process communication primitives leads to pessimistic results as well since it does not model any internal process structure and therefore worst-case blocking time must be considered, even though it can never occur (see section 7).

To achieve more precise analysis, process models allowing more precise and complex timing constraints are needed. In [13] the timed automata are extended by asynchronous processes (i.e. processes triggered by events) to provide model for event-driven systems, which is further used for schedulability analysis. Processes (in [13] called tasks) associated to locations of timed automaton are executable programs characterised by its worst-case execution time, deadline and other parameters for scheduling (e.g. priority). Transition leading to a location in such automaton denotes an event triggering the process and the guard on transition specifies the possible arrival times of the event. Released processes are stored in a process queue and they are assumed to be executed according to a given scheduling strategy. Both non-preemptive and preemptive scheduling strategies are allowed. Such modell can deal with non-periodic processes in more accurate manner than RMA. Moreover there is a possibility to model internal process structure as it is shown in section 2, but the computation time of modelled blocks of code cannot vary.

This drawback is overcome by more detailed process model proposed in [9] providing a method for constructing models of real time Ada tasking programs. Time, safety or liveness properties of produced model based on constant slope linear hybrid automata can be automatically analysed by HyTech verifier. The state of the hybrid automaton consists of various state variables representing an abstraction of program's state and it contains also continuous variables used to measure the amount of processor time allocated to each process. A transition of the hybrid automaton represents execution of the sequential code segment. The timing constraints of the transition are derived from the time bounds of the corresponding code. Even thought the author reports that the analysing algorithm does usually terminate in practice, the reachability problem for hybrid automata is undecidable in general.

Hybrid automaton (or some of its subclass e.g. stopwatch automaton [10]) is needed to model premption since it is necessary to accumulate computing time of each process separately. The continuous variable used to measure the amount of CPU time allocated to each process must progress when the corresponding process is executed and must be stopped when the corresponding process is preempted. Such behaviour cannot be modelled by timed automaton that does not allow stopping of the clock variable (see [1]).

Based on these observations we provide the model of real time system consisting of several concurrent processes scheduled by cooperative scheduler. Since the internal structure of the processes and the scheduler are modelled by timed automata, the model of the system is more accurate than the models used for schedulability analysis (RMA and timed automata extended by processes). Opposite to the model of the system with preemption based on hybrid automata, this approach has guarantied termination of verification algorithm due to decidability of reachability problem and model checking of timed computation tree logic (TCTL) problem. Moreover timed automata are one of the most studied models for real time systems and several model checkers are available (e.g. Kronos[1] and UPPAAL[2] [2])

Preemptive schedulers are known to provide higher utilisation of processor than the cooperative ones [3]. On the other hand the processor utilisation is less important

[1] http://www-verimag.imag.fr/TEMPORISE/kronos/

[2] http://www.uppaal.com

criterion when the schedulability can be proven for a given set of processes under cooperative policy. Moreover the cooperative scheduling has some advantages important especially for hard real time applications where the highest reliability is required. In cooperative scheduling, process specifies when it is willing to release CPU to another process. Then it is easy to make sure that all data structures are in a defined state. Applications using cooperative scheduling are therefore easier to program and to debug. In this paper we present another important advantage of cooperative scheduling that is possibility to create mathematical model of the application based on timed automata and to verify its time, safety and liveness properties.

The rest of this paper is organised as follows: section 2 illustrate on an example of scheduling anomaly that when one wants to make use of the internal process structure, then it is needed to consider also lower margins of computation times. Sections 3 and 4 represents a marginal part of this article. They deal with modelling of applications running under operating system based on cooperative scheduling. Since interrupt handling can play important role in such systems, they are taken into consideration in section 5. Section 0 illustrates an extension of proposed model by inter-process communication. Presented methodology and its comparison with RMA approach is demonstrated on case study in section 7.

2 On Scheduling Anomaly in Multitasking Operating System

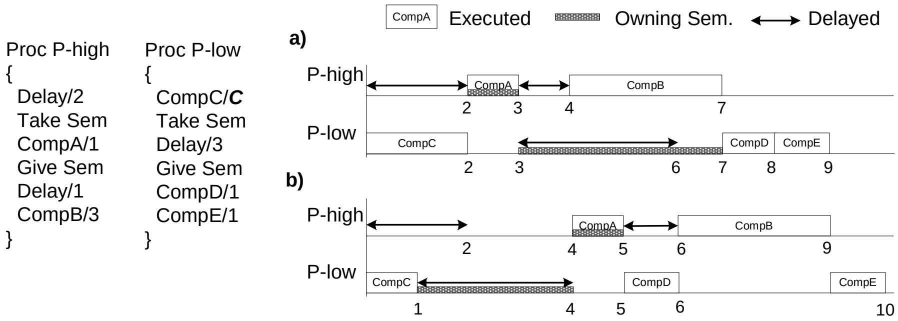
Several multiprocessor time anomalies are known in the scheduling theory [3],[5],[7]. Similar non-linear behaviour (a shortening of the computation time leading to the prolongation of the completion time) can be found on one processor regardless the scheduling policy (preemptive or cooperative), when the processes contain computations, resource sharing and idle waiting (notice that the idle waiting is processed in parallel with computation of another process).

Example depicted in Fig. 2.1 shows a high priority processes *P-high* and a low priority process *P-low* sharing one resource represented by a semaphore *Sem*. The processes consist of computations with specified deterministic computation time, of idle waiting with specified deterministic delay and of shared resource guarded by semaphore. The computation times and delays given behind slash are assumed to be constants. The computation time of *CompC* is *C*=2 in the instance a) or *C*=1 in the instance b).

The semaphore is taken by *P-high* first in the instance a) regardless the scheduling policy (priority based preemptive or priority based cooperative). Consequently the process *P-high* is completed in 7 time units and the process *P-low* is completed in 9 time units, see Fig. 2.1 a). In the instance b), the semaphore is taken by process *P-low* first and consequently the process *P-high* is completed in 9 time units and the process *P-low* is completed in 10 time units, see Fig. 2.1 b).

The shortening of the computation time in the process P-low (*C* shorted from 2 to 1) leads to the prolongation of the completion time of both processes. As a consequence this example illustrates a necessity to consider also lower margins of computation times when process internal structure is modelled.

This result is important to modelling process internal structure by timed automata extended by tasks [13]. Timed automata extended by tasks allow to model precedence constraints over tasks by boolean variables shared between tasks and automaton. Therefore it is possible to model each process as timed automaton and to associate to its locations tasks representing corresponding computation. Precedence constraints are used to prevent starting of next computation before the previous one is finished. Since tasks associated to locations is characterised only by its worst-case computation time, some mechanisms must be used to prevent occurrence of anomalies described in this section. One solution can be to leave processor idling when some computation is finished sooner than it was supposed.



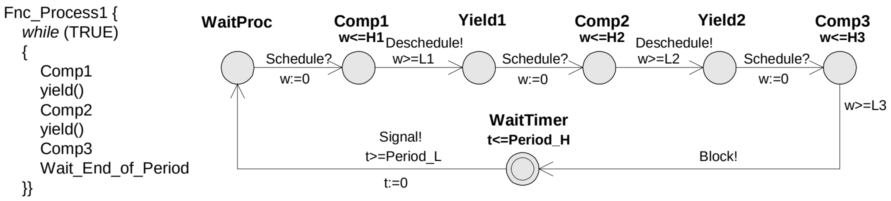
```
Proc P-high        Proc P-low
{                  {
  Delay/2            CompC/C
  Take Sem           Take Sem
  CompA/1            Delay/3
  Give Sem           Give Sem
  Delay/1            CompD/1
  CompB/3            CompE/1
}                  }
```

Fig. 2.1. Example of the monoprocessor scheduling anomaly

3 Cooperative Scheduling Model

Cooperative scheduling enables to deschedule currently executed process only in explicitly specified points, where the system call *yield()* is called or where the process is waiting.

An example of the application process model is depicted in Fig. 3.1. There are four types of locations. *Computation* locations (*Comp1*, *Comp2*, *Comp3* for short) corresponding to non-preemptible blocks of code (the *Computations* do not contain any blocking operation). Each two successive *Computation* locations are separated by one *Yield* location corresponding to yield instruction where the process can be descheduled and then it waits until it is scheduled again. On *WaitTimer* location the process does not require the processor. *WaitTimer* location is followed by *WaitProc* location where the signalled process waits until it is scheduled. The double circle used for *WaitTimer* location specifies that this is initial location.



```
Fnc_Process1 {
  while (TRUE)
  {
    Comp1
    yield()
    Comp2
    yield()
    Comp3
    Wait_End_of_Period
}}
```

Fig. 3.1. Model of the application process executed under cooperative scheduling policy

As each part of the program modelled by *Computation* location cannot be affected by the preemption, its finishing time is equal to the computation time which is supposed to be known a priory and bounded by interval ⟨L,H⟩ (lover and upper margins allow to involve uncertainty of execution time due to non-modelled code branching inside the computations, bus errors, cache faults, page faults, cycle stealing by DMA device, etc.).

The following behaviour of the cooperative scheduler is assumed: if the processor is free, the process with the highest priority among all processes in a queue of ready processes is scheduled. The currently executed process will run until it voluntarily relinquishes processor by calling system call *yield()* or until it is blocked. The model of the cooperative scheduler is created as the network of automata synchronised with application processes through synchronisation channels as depicted in Fig. 3.2. The scheduler chooses the highest priority ready process and enables its execution through *Schedule* channel. *Deschedule* channel is used to signal that the process relinquishes the processor (by *yield()*). The *Block* channel is used to relinquish processor on some blocking system call and the *Signal* channel announce that the blocking is finished and the process is ready to be executed on the processor.

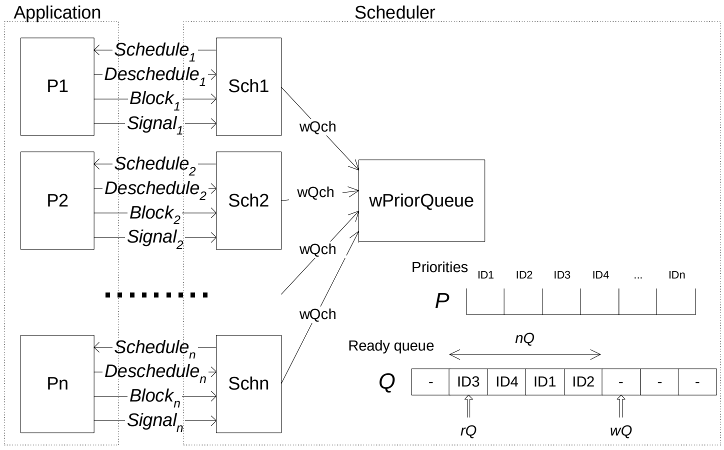


Fig. 3.2. Synchronization of cooperative scheduler with processes

One automaton of the cooperative scheduler model (*Sch*$_i$) is depicted in Fig. 3.3.

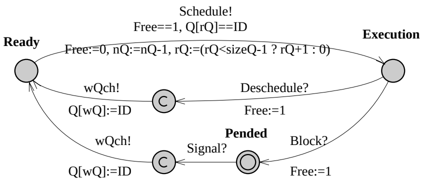


Fig. 3.3. One automaton Sch_i of the cooperative scheduler in Fig. 3.2

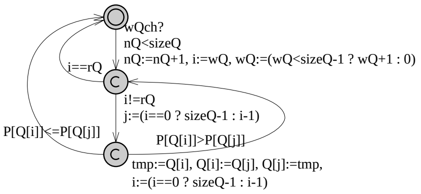

wQch?
nQ<sizeQ
nQ:=nQ+1, i:=wQ, wQ:=(wQ<sizeQ-1 ? wQ+1 : 0)
i==rQ
i!=rQ
j:=(i==0 ? sizeQ-1 : i-1)
P[Q[i]]<=P[Q[j]]
P[Q[i]]>P[Q[j]]
tmp:=Q[i], Q[i]:=Q[j], Q[j]:=tmp,
i:=(i==0 ? sizeQ-1 : i-1)
C
C

Fig. 3.4. Automaton *wPriorQueue* providing reordering of queue *Q*

Each process is identified by unique integer *ID* (0,1,2,...). Priority of the process is stored in global array *P*, indexed by *ID*. *ID*s of all processes, which are in *Ready* state, are stored in the queue modelled as global array *Q* representing a circular buffer. The integer *nQ* is the number of elements in the queue. The integer *rQ* is the position for reading of the first element in *Q* and the integer *wQ* is position of the first empty element in *Q* as is depicted in Fig. 3.2. Processes are ordered in descending order according to their priorities in *Q* (*rQ* points to the ready process with highest priority).

As shown in Fig. 3.3 mutual exclusive access to *Execution* location is guarded by two-state variable *Free*. Moreover, only the highest priority process scheduler automaton (its *ID* is at the top of ready queue) can take transition from *Ready* to *Execution* location. To prevent processor idling, the transition from *Ready* to *Execution* location must be taken as soon as it is enabled. This is provided by declaring the channel *Schedule* as *urgent channel* (no time progress is enabled when there are some enabled transitions synchronised through urgent channel). The two unnamed locations with the letter C inside the circle are so called *committed* locations providing atomicity of traversing of in-coming and out-coming transitions (committed location must be left immediately without any interference of other automaton in the model). These locations are in Fig. 3.3 necessary only due to impossibility to use two synchronizations on one transition in UPPAAL.

ID of the process leaving the *Ready* state is deleted from the ready queue by decrementing number of elements in the queue *nQ* and by moving reading pointer *rQ* to the next element in the queue. *ID* of the process entering the *Ready* state is written to the end of ready queue. The ready queue must be reordered after this operation. Ordering according priorities is provided by automaton *wPriorQueue* depicted in Fig. 3.4. Reordering mechanism is started by synchronisation channel *wQch*.

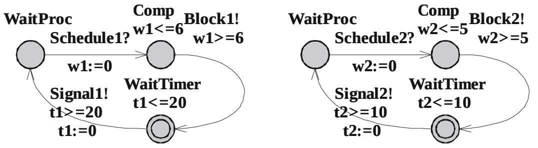
Note on the modelling of the context switch time: *Notice that the model of the scheduler automaton proposed in Fig. 3.3 is simplified by assumption that the context switch does not take any time. But for proper exploration of time properties of real-time system the context switch time should be considered. Since the context switch in cooperative scheduling occurs once per Computation location, context switch time can be simply involved in the computation time of each Computation.*

4 Modelling Deterministic Behaviour of the Scheduler

Notice that proposed model created as synchronised product of application process automata and corresponding scheduler automata (Fig. 3.3) contain non-deterministic behaviour, which does not correspond to real behaviour of the scheduler. This non-determinism occurs when the transition from *Ready* to *Execution* location of one scheduler automaton *Sch*$_i$ is enabled and simultaneously the transition from *Pended* to *Ready* location of another scheduler automaton *Sch*$_j$ is enabled. In such case the transition from *Pended* to *Ready* should be taken first since the scheduler updates states of processes first. Then the highest priority ready process should be chosen and the scheduler automaton of this process should take the transition to *Execution* location. Please realise that the model adopted in previous paragraph allows also other behaviour: the transition from *Ready* to *Execution* location of the first scheduler automaton is taken first and the transition from *Pended* to *Ready* location of the second scheduler automaton is taken afterwards. In such case the second process looses the chance to compete the processor that is undesirable since the lower priority process can take the processor even though there is some higher priority ready process at the same time.

The objective of this paragraph is to eliminate such undesirable behaviour, which does not correspond to reality. The transition priorities will be used to determine the order of transitions. High priority 2 will be assigned to the transitions from *Pended* to *Ready* locations in all scheduler automata. Lower priority 1 will be assigned to all remaining transitions. Since the transition priority is not concerned in timed automata, it is incorporated by modifying guards on transitions.

This approach is demonstrated on simple example of two periodic application processes modelled by time automata depicted in Fig. 4.1. Process P1 is the low priority one and process P2 is the high priority one. Both processes are scheduled by cooperative scheduler modelled by two scheduler automata Sch1 and Sch2 depicted in Fig. 3.3.



a) Low priority process P1 b) High priority process P2

Fig. 4.1. Automata of application processes

Resulting model of whole application is a synchronised product of all concerned automata (Sch1, Sch2, P1, P2, wPriorQueue) and it is depicted in Fig. 4.2. The location names consist of the first letters of the location names of the original automata (in the order Sch1, Sch2, P1, and P2). Priorities are assigned to the transitions were non-deterministic choice can occur (high priority 2 to the transitions from *Pended* to *Ready* and low priority 1 to other transitions). Notice that urgent locations (symbol ∪ inside the location) are used to prevent processor idling (the time progress is disabled when some automaton resides in urgent location). This function was provided by urgent channel *Schedule* in automaton *Sch*$_i$ in previous section.

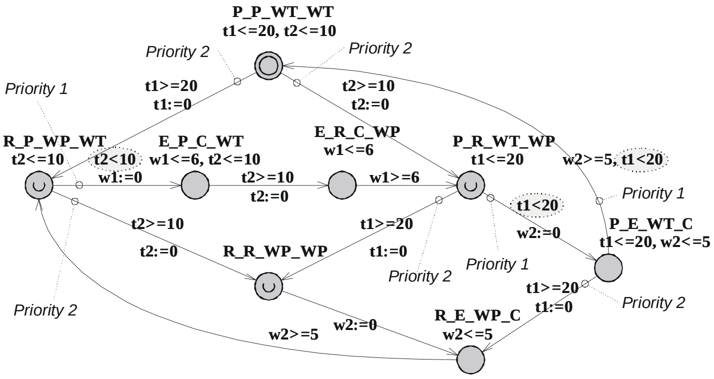


Fig. 4.2. Resulting model concerning transition priorities (synchronised product of Sch1, Sch2, P1, P2, wPriorQueue)

Our approach to transition priority is the following. Suppose the transition from *Pended* to *Ready* to be taken non-deterministically between lower and upper margin of the signalling time (within interval <*L, H*>). Since any process can be scheduled prior to another process becoming ready infinitely short time after scheduling decision, the transition priority has no sense in interval <*L, H*). In other words it is desired to preserve the non-determinism in interval <*L, H*). On the other hand, the priority of transition from *Pended* to *Ready* must be high at time *H* since the scheduler updates states of processes prior to scheduling decision (as explained above). Our approach to give priority to the transitions is to restrict the lower priority transition guard g_1 to $g_1'=g_1\wedge(t<H)$, where t is a clock and $t\leq H$ is invariant of the location where the higher priority transition begin. Restricted guards of lower priority transitions are in doted grey filled ellipsis in Fig. 4.2.

5 Interrupts

Interrupts are usually used for fast handling of asynchronous external events. Interrupt is particularly important in cooperative scheduling since low priority process cannot be preempted and therefore high priority process cannot be used to handle asynchronous event when short response time is required. When the interrupt request (IRQ) arrives from the environment and corresponding interrupt is enabled, currently executed process is interrupted and interrupt service routine (ISR) is executed. The *relative finishing time F* of currently executed *Computation* is therefore prolonged by computation time of ISR (C_{ISR}) and it is no more equal to the known *computation time*. Therefore it is needed to change upper margin *H* of each computation location in the timed automata process model. Each *H* is prolonged by *MaxSC* (maximum server capacity), the value corresponding to the processor time reserved for all interrupt service routines. Since the number of interrupt requests depends on the environment, the total computation time of all ISR (ΣC_{ISR}) is not known a priory and moreover the existence of its upper bound is not guaranteed.

The *interrupt server* limiting amount of processor time spent for interrupts is used to guarantee that ΣC_{ISR} does not exceed *MaxSC* value. Contrary to servers used for handling aperiodic tasks in scheduling theory (pooling, deferrable, sporadic servers [3], [6]), the prevention of servicing interrupt must be done at the hardware level (by disabling IRQ) and before the IRQ occur. The architecture of the system with *interrupt server* is depicted in Fig. 5.1. Interrupt service routines are not called directly when some interrupt is requested, but they are wrapped by the code of *ISR_Server()* function (see Fig. 5.3). The *interrupt server* has specified *server capacity SC*, which is filled by the value *MaxSC* at the beginning of each computation. The function *Fill_Server(MaxSC)* listed in Fig. 5.3 is used for it. When an interrupt occurs the *server capacity SC* is decreased by the value of corresponding C_{ISR} and *interrupt server* checks if the remaining capacity *SC* is sufficient for handling next *ISR*. If not the corresponding *IRQ* is disabled. This check is provided when *SC* changes, once by *Fill_Server()* and repeatedly on each interrupt by *ISR_Server()*. Notice that C_S, the computation time of *ISR_Server()*, is considered. Further *H* has to be prolonged by C_{FS}, the computation time of the function *Fill_Server()* (see Fig. 5.2). The lower margin *L* of any computation location is affected only by C_{FS}.

Notice that the function *ISR_Server()* supposes that the hardware does not support nested interrupts (*ISR_Server()* cannot be interrupted by another interrupt).

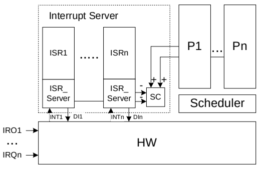


Fig. 5.1. System architecture with *interrupt server*

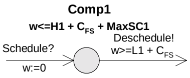


Fig. 5.2. *Computation* location considering interrupts

```
Fill_Server (MaxSC)                ISR_Server ()
{                                  {
   Disable_INT                        SC := SC – C_ISR - C_S
   SC :=MaxSC                         call ISR
   Check for all IRQ                  Check for all IRQ
      if (SC – C_ISR - C_S) < 0          if (SC – C_ISR - C_S) < 0
         Disable IRQ                        Disable IRQ
      else                               else
         Enable IRQ                         Enable IRQ
   Enable_INT                      }
}
```

Fig. 5.3. *Interrupt server* routines

Choice of MaxCS value for different locations depends on application requirements and it is specified at the design stage.

6 Inter Process Communication Primitives

Very important part of each multitasking application (and source of many possible errors) is a communication between processes and their synchronisation. Operating system usually provides many facilities to manage inter process communication. It is not intention of this paper to introduce models of all possible kinds of inter process communication.

On example of *semaphore* we show, how to extend the proposed model of the scheduler and application. The semaphore is the primitive used mostly for synchronization and mutual access to resources. It can be taken or given by the process using the system calls *Take()* or *Give()*. When the semaphore is given, its value is increased.

When the semaphore is taken, its value is decreased. When the value of the semaphore is zero, it cannot be taken and the process attempting to take it is blocked until the semaphore is given by another process. This blocking time can be bounded by timeout. When more than one processes are blocked on one semaphore, they are waiting in priority queue or FIFO (First In First Out) queue. This basic behaviour of semaphore can be modified according to the purpose it is dedicated to. We suppose the semaphore being of counting type with value ranging from zero to *MaxCount*.

Example of an application process model with semaphore is depicted in Fig. 6.1. The process attempts to take the semaphore by synchronisation *Take!*. Then it waits in location *WaitSem* until the semaphore is taken (synchronisation *Taken?*) or until timeout expires (synchronisation *TOut!*). The synchronisation *Give!* is used to give the semaphore. Notice that giving the semaphore is not blocking operation and therefore the semaphore is given on the transition entering the *Comp3* location. On the other hand taking semaphore is blocking operation and therefore transitions with *Taken?* and *TOut!* lead to the locations *WaitProc2* or *WaitProc3* resp. where the process waits for the processor.

```
Fnc_Process1 {
  while (TRUE) {
    Comp1
    Result := Take (Sem, TimeOut)
    if (Result == TOut) {
      Comp4
      Yield() }
    else {
      Comp2
      Give (Sem) }
    Comp3
    Wait_End _of_Period
  } }
```

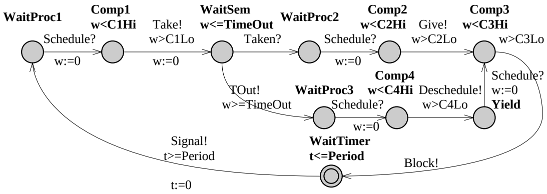


Fig. 6.1. Model of process containing *Take* and *Give* one semaphore

The scheduler model for application with two semaphores is depicted in Fig. 6.2. The scheduler of executed process is asked for taking the semaphore by synchronisation *Take?*. If the semaphore is empty (*Sem==0*), the processor is relinquished (*Free:=1*), *ID* of the process is written to the queue of the semaphore (*SemQ*) and the queue (FIFO or priority) is reordered by synchronisation *wSemQch!*. The scheduler and the process then wait in the location *WaitSem* until the semaphore is given by another process or until its time-out expires. If the semaphore is not empty (*Sem>0*) its value is decreased and the synchronisation *Taken!* is immediately followed by synchronisation *Schedule!* to continue in execution. The processor is not relinquished in this case.

The queue of the processes waiting for the semaphore (*SemQ*) can be FIFO queue or priority queue. In the case of priority queue, its elements (*ID*s of processes) must be reordered according to priorities when the next process issues *Take* on the empty semaphore. This is managed by the automaton similar to the one depicted in Fig. 3.4. The only difference is the name of the queue (*SemQ*, *wSemQch*, *nSemQ*, *rSemQ*, *wSemQ*). Reordering is not necessary when FIFO queue is used.

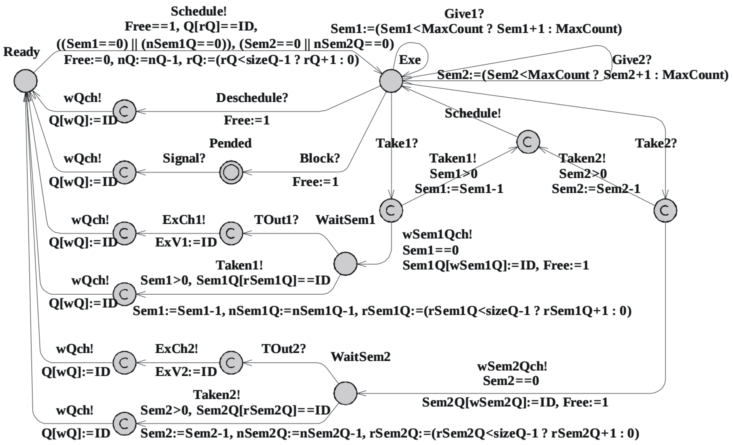

Schedule!
Free==1, Q[rQ]==ID,
((Sem1==0) || (nSem1Q==0)), (Sem2==0 || nSem2Q==0)
Free:=0, nQ:=nQ-1, rQ:=(rQ<sizeQ-1 ? rQ+1 : 0)
Ready
Give1?
Sem1:=(Sem1<MaxCount ? Sem1+1 : MaxCount)
Exe
Give2?
Sem2:=(Sem2<MaxCount ? Sem2+1 : MaxCount)
wQch!
Q[wQ]:=ID
Deschedule?
Free:=1
Schedule!
Pended
wQch!
Signal?
Block?
Q[wQ]:=ID
Free:=1
Take1?
Taken1!
Sem1>0
Sem1:=Sem1-1
Taken2!
Sem2>0
Sem2:=Sem2-1
Take2?
wQch!
ExCh1!
TOut1?
WaitSem1
Q[wQ]:=ID
ExV1:=ID
wSem1Qch!
Sem1==0
Sem1Q[wSem1Q]:=ID, Free:=1
Taken1!
wQch!
Sem1>0, Sem1Q[rSem1Q]==ID
Q[wQ]:=ID
Sem1:=Sem1-1, nSem1Q:=nSem1Q-1, rSem1Q:=(rSem1Q<sizeQ-1 ? rSem1Q+1 : 0)
wQch!
ExCh2!
TOut2?
WaitSem2
Q[wQ]:=ID
ExV2:=ID
wSem2Qch!
Sem2==0
Sem2Q[wSem2Q]:=ID, Free:=1
Taken2!
wQch!
Sem2>0, Sem2Q[rSem2Q]==ID
Q[wQ]:=ID
Sem2:=Sem2-1, nSem2Q:=nSem2Q-1, rSem2Q:=(rSem2Q<sizeQ-1 ? rSem2Q+1 : 0)

Fig. 6.2. Scheduler model containing two semaphores (extension of Fig. 3.3)

As it has been explained in section 4, rescheduling is not possible before updating states of all processes. Therefore transitions from *Ready* to *Exe* have lower priority than the transitions from *WaitSem* to *Ready*. That means that scheduling cannot occur when there is any process waiting on signalled semaphore. This is modelled by restricting transition from *Ready* to *Exe* guard *g* to $g'=g \wedge \forall_i\ (Sem_i=0 \vee nSem_iQ=0)$.

7 Case Study

This section demonstrates the methodological approach to modeling real-time operating system based applications on the example of the elevator controller.

The elevator cabin either resides in a floor, or it moves between floors, or it goes through a floor (Fig. 7.2 b). The cabin movement is controlled by a three-state variable *Go* having value UP, or DOWN, or STOP. The value of the cabin possition sensor is stored in variable *In* which is equal to 1 when the cabin resides in or goes through any floor. The motor overheating is detected by value Hi of variable *Temper* (see Fig. 7.2 c). In such case the cabin must stop in the forthcoming floor and further movement is disabled by resetting variable *Enable*.

All sensors and actuators are connected to the control system by the buss guaranteeing the message delivering time. The control system software consists of three processes scheduled by cooperative scheduler and one interrupt service routine.

The buss controller generates interrupt request (*IRQ*), when new data are received from the buss or all prepared data were transmitted to the buss (see Fig. 7.2 d). If the interrupt is enabled (*EN=1*), the hardware interrupt controller (see Fig. 7.2 e) interrupts the CPU, the *ISR_Server* is invoked (see Fig. 7.2 f) and semaphore *Sem1* is signalled. The highest priority process *ComProc*, providing communication services, takes semaphore *Sem1*, then it recognises the data receiver and it signals semaphore *Sem3* or *Sem4* (see the code in Fig. 7.1 and corresponding automaton in Fig. 7.2 g). The middle priority process *DiagProc* provides diagnostic and emergency shut-down when the motor is over-heated. It is waiting on *Sem3* (see the code in Fig. 7.1 and corresponding automaton in Fig. 7.2 h). The lowest priority process *CtrlProc* providing the cabin control is waiting on *Sem4* (see the code in Fig. 7.1 and corresponding automaton in Fig. 7.2 i). The semaphore *Sem2* provides mutual exclusive access to all shared data. Pseudocode of the control system software is in Fig. 7.1.

The goal of this case study is to create model of the system and to use the model-checking tool UPPAAL to verify the following properties:

- Prop1: No *IRQ* is lost i.e. a) handling interrupt is short enough and b) the interrupt server capacity is sufficient.
- Prop2: The execution of *ComProc* is started between two successive interrupts.
- Prop3: The execution of *ComProc* is finished within 24 ms after taking *Sem1*.
- Prop4: The execution of *DiagProc* is finished within 24 ms after taking *Sem3*.
- Prop5: The execution of *CtrlProc* is finished within 34 ms after taking *Sem4*.
- Prop6: Usage of elevator is disabled 166 ms after the motor overheating.
- Prop7: The cabin will stop in any floor 5.2 s after the motor overheating.

The automata of the proposed model are depicted in Fig. 7.2. The interconnection of all automata is depicted in Fig. 7.2 a). The scheduler automaton is similar to the one in Fig. 6.2 but it is extended for four semaphores.

```
ComProc ()
{
  while (true) {
    Take (Sem1)
    Fill_Server (S1)
    Computation1   /12
    Take (Sem2)
    Fill_Server (S2)
    Computation2   /12
    if (Data==DIAG)
      Give (Sem3)
    if (Data==CTRL)
      Give (Sem4)
    Give (Sem2)
  } }
```

```
DiagProc ()
{
  while (true) {
    Take (Sem3)
    Fill_Server (S1)
    Computation1   /12
    Take (Sem2)
    Fill_Server (S2)
    Computation2   /12
    if (Temper==Hi){
      Enable:=0
      if (Go!=STOP and In==1)
        Go:=STOP
    }
    Give (Sem2)
  } }
```

```
CtrlProc ()
{
  while (true) {
    Take (Sem4)
    Take (Sem2)
    Fill_Server (S1)
    Computation1   /22
    Yield
    Fill_Server (S2)
    Computation2   /12
    Give (Sem2)
  }
}
```

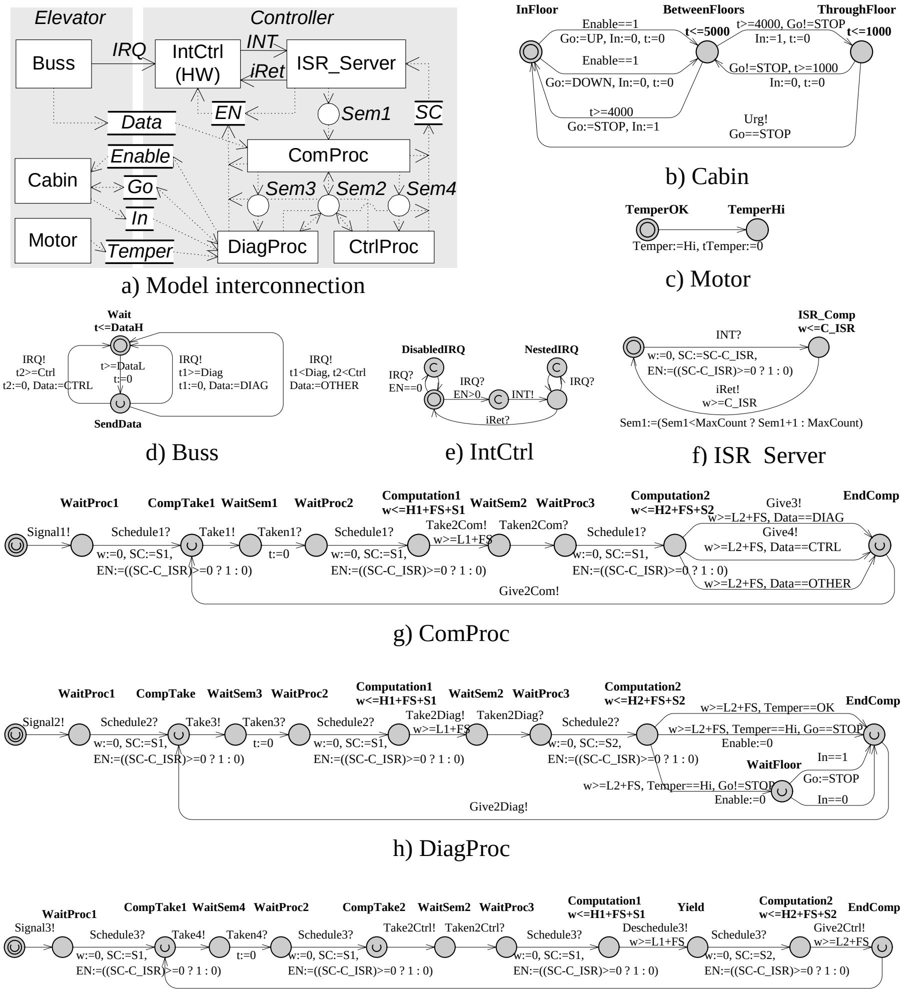
Fig. 7.1. Control system software pseudocode

The specified properties are formalized in CTL as follow:

- Prop1 a) $\forall\Box\ \neg$ IntCtrl.NestedIRQ, b) $\forall\Box\ \neg$IntCtrl.DisabledIRQ
- Prop2: $\forall\Box$ Sem1<2
- Prop3: $\forall\Box$ ((ComProc.WaitProc2 $\wedge$ ComProc.t=0) $\Rightarrow$ $\forall\Diamond$ (ComProc.EndComp $\wedge$ ComProc.t<24))
- Prop4: $\forall\Box$ ((DiagProc.WaitProc2 $\wedge$ DiagProc.t=0) $\Rightarrow$ $\forall\Diamond$ (DiagProc.EndComp $\wedge$ DiagProc.t<24))
- Prop5: $\forall\Box$ ((CtrlProc.WaitProc2 $\wedge$ CtrlProc.t=0) $\Rightarrow$ $\forall\Diamond$ (CtrlProc.EndComp $\wedge$ CtrlProc.t<34))
- Prop6: $\forall\Box$ ((Temper=Hi $\wedge$ tTemper=0) $\Rightarrow$ $\forall\Diamond$ (Enable=0 $\wedge$ tTemper<166))

- Prop7: $\forall\Box\ ((\text{Temper}=\text{Hi} \wedge \text{tTemper}=0) \Rightarrow$
  $\forall\Diamond\ (\text{Enable}=0 \wedge \text{In}=1 \wedge \text{Go}=\text{STOP} \wedge \text{tTemper}<5200))$

Result of verification: all properties except Prop6 are satisfied.



a) Model interconnection

b) Cabin

c) Motor

d) Buss

e) IntCtrl

f) ISR Server

g) ComProc

h) DiagProc

i) CtrlProc

Fig. 7.2. Elevator and control system model

Note: *Please notice that under the worst-case conditions the cabin will not stop on any floor 5,2 s after increasing of the motor temperature (Prop7 is not satisfied) even though the DiagProc will react on this situation within 166 ms (Prop6 is satisfied) and the maximal time that the cabin spends between two floors is 5 s (time invariant of the state BetweenFloors in cabin model in* Fig. 7.2 b) *is t<=5000). This re-*

sult would be hart to find by separate analysis of time and logical properties of the system. In fact the property Prop7 is satisfied for the value of tTemper<5332 since 5332<=166+5000+166.

7.1 Comparison with RMA approach

Notice that properties *Prop3, Prop4* and *Prop5* represent exploration of the worst-case completion time for processes *ComProc*, *DiagProc* and *CtrlProc*. Let's compare approach adopted in this article to RMA approach.

Let's suppose that the processes are scheduled by preemptive rate monotonic scheduling:

- the minimal interarrival times are $T_{ComProc}$=50, $T_{DiagProc}$=100 and $T_{CtrlProc}$=200,
- the worst-case computation times are $C_{ComProc}$=24, $C_{DiagProc}$=24 and $C_{CtrlProc}$=34
- critical section (*Sem2*) is locked for durations $D_{ComProc}$=12, $D_{DiagProc}$=12 and $D_{CtrlProc}$=34.

It is obvious that without internal structure knowledge, the worst-case blocking time on *Sem2* must be considered: $B_{ComProc}$=34, $B_{DiagProc}$=34.

Based on these very abstracted assumptions on system behaviour, the RMA evaluates processes *ComProc* and *DiagProc* non-schedulable.

8 Conclusion and Future Work

The cooperative scheduling approach given in this article avoids preemption modelling by hybrid automata. The model of the application processes and cooperative scheduler is based on timed automata, for which model checking of TCTL property problem is decidable (opposite to hybrid automata). Interrupts and inter-process communications – the most important aspect of real time embedded applications – are taken into consideration in the proposed model. With respect to the processor utilisation and reaction time the cooperative scheduling conceived in this article is not the most efficient one, but due to simplicity reasons many embedded applications are often based on similar cooperative scheduling mechanisms handling interrupts separately, therefore this approach is not just an academic idea.

Existing approaches for design and analysis of real-time applications, like Rate Monotonic Analysis (using preemptive scheduling based on priority assignment respecting the rate of periodic processes), use very elegant way of deciding whether the application is schedulable or not. But it is needed to mention, that the model checking approach provides a room for verifying more complex properties (e.g. detection of deadlocks in communication, specification of buffer size,…). Model checking provides also room for modelling of more complex time behaviour of the controlled system, running truly in parallel with the control system (modelled as separate automaton).

As the complexity of the model checking remains very huge in a general case it is motivating to set up the rules applied at a design phase, that would lead into the state spaces of reasonable size. Specification of such rules linked to the identification of the controlled systems represents a possible direction of our future work.

Acknowledgement

This work was supported by the Ministry of Education of the Czech Republic under Project LN00B096.

References

[1] Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126:183-235, 1994.

[2] David, A.: Uppaal2k: Small Tutorial. Documentation to the verification tool Uppaal2k. http://www.docs.uu.se/docs/rtmv/uppaal/

[3] Buttazzo, G., C.: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Kluwer Academic Publishers, Boston (1997)

[4] Sha, L., Klein, M., Goodenough, J.: Rate Monotonic Analysis for Real-Time Systems. 129-155. Foundations of Real-Time Computing: Scheduling and Resource Management. Boston, MA: Kluwer Academic Publishers (1991)

[5] Graham, R.L.: Bounds on multiprocessing timing anomalies. SIAM Journal on Applied Mathematics, Vol. 17, pp. 416-429, 1969

[6] Larsen, K.G., Pettersson, P., Yi, W.: Model-Checking for Real-Time Systems. In Proceedings of the 10th International Conference on Fundamentals of Computation Theory, Dresden, Germany, 22-25 August, 1995. LNCS 965, pages 62-88, Horst Reichel (Ed.)

[7] Liu, J.W.S.: Real-time systems. Prentice-Hall, Inc., Upper Saddle River, New Jersey 2000. ISBN 0-13-099651-3

[8] Shaw, A.: Reasoning about time in higher-level language software. IEEE Transactions on Software Engineering, vol. 15, July 1989

[9] Corbett, J. C.: Timing analysis of Ada tasking programs. IEEE Transactions on Software Engineering. 22(7), pp. 461-483, July 1996

[10] Cassez F., Larsen K.: The Impressive Power of Stopwatches. In Proceedings of CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 2000 CONCUR'2000. LNCS 1877, p. 138 ff., 2000

[11] Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? *Journal of Computer and System Sciences* 57:94--124, 1998

[12] Bouyer, P., Dufourd, C., Fleury, E., Petit, A.: Are Timed Automata Updatable?. In Proc. 12th Int. Conf. Computer Aided Verification (CAV'00), LNCS, Vol.1855, pp. 464-479, Springer (2000)

[13] Fersman, E., Pettersson, P., Yi, W.: Timed Automata with Asynchronous Processes: Schedulability and Decidability. In Proceedings of 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2002, Grenoble, France, April 8-12, 2002, pp.67-82, Springer-Verlag, 2002. Lecture Notes in Computer Science, Vol.2280