TIMED AUTOMATA MODEL OF PREEMPTIVE MULTITASKING 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
{xwasznio, hanzalek}@fel.cvut.cz

Abstract: The aim of this article is to show, how a multitasking application running under a real-time operating system compliant with the OSEK/VDX standard can be modeled by timed automata. The application under consideration consists of several tasks, it includes resource sharing and synchronization by events. For such system, model-checking theory based on timed automata and implemented in model-checking tools can be used to verify time and logical properties of the proposed model. It is shown that the proposed model is over-approximation in the case of preemptive scheduling policy. This methodology is demonstrated on automated gearbox case study. *Copyright © 2005 IFAC*

Keywords: Verification, Real-Time Operating Systems.

1 INTRODUCTION

This paper deals with modeling of applications running under a real-time operating system (OS). Typical application under assumption, shown as a case study in Section 7, is a controller consisting of periodic and aperiodic tasks constrained by deadlines and synchronized via communication primitives.

The model-checking (Larsen, *et al.*, 1995) approach, shown in this paper, provides timed automata (Alur and Dill, 1994) model of an operating system, application tasks and the controlled environment. In the scheduling theory, the task model usually consists of its execution time, the blocking time and the inter-arrival time. Our approach assumes a *fine grain model* of the task internal structure consisting of computations, system calls, selected variables, code branching and loops. Therefore the model combines both, logic and timing parameters of a discrete event system enabling to check rather complex properties (safety and bounded liveness properties, schedulability, state reachability) by model-checking tools (e.g. UPPAAL (Behrmann, *et al.*, 2001) and Kronos (Daws, *et al.*, 1996)) in finite time.

Even though timed automata and model-checking (analogous to other formal methods) allows modeling and verifying almost everything, it is generally known, that they are susceptible to state space explosion. This fact restricts the size of verified application to the small size that seems to be unusable in praxis (compared with matured schedulability analysis methods (Liu, 2000)). Therefore we try to show in this paper, how to build a compromise model of reasonable size on one side and of reasonable granularity allowing detailed formal analysis of real-time properties that can not be done by schedulability analysis on the other side.

Methods for schedulability analysis, e.g. rate monotonic analysis (RMA) (Liu, 2000) have been widely used in praxis. However they can lead to pessimistic results when non-periodic tasks, shared resources and other features are incorporated (Bailey, *et al.*, 1995). The schedulability analysis based on model-checking of the fine grain model provides less pessimistic results in some cases.

Fersman, *et al.* in (2002) and (2003) extended timed automata by asynchronous tasks (i.e. tasks triggered by events) to provide model for event-driven systems. This approach provides good results for aperiodic tasks but it is not suited to model the task internal structure as follows from results of (Krčál and Yi, 2004).

Corbet in (1996) provides model of real time Ada tasking programs based on hybrid automata. Opposite to timed automata used in our approach, reachability problem is undecidable for hybrid automata and the termination of the verification algorithm is therefore not guarantied in general.

Timed automata are used to model primitives of Ravenscar run-time kernel for Ada in (Lundqvist and Asplund, 2003). However, the time in application is discrete opposite to our approach where the time is dense.

Preemptive Petri Nets (Bucci, *et al.*, 2004) or Scheduling Petri Nets (Lime and Roux, 2004) can be also used to model multitasking application. Both formalisms are very similar. Their semantics can be either similar to hybrid automata or to semantic of timed automata.

This paper is organized as follows: Section 2 describes *fine grain model* used in this paper. Sections 3, 4, and 5 presents the main result of this paper – timed automata models of tasks and OSEK compliant OS (OSEK, 2003). This model is an over-approximation from the model-checking point of view in the case of the preemptive scheduling and WCET (worst-case execution time) differing from BCET (best-case execution time) as it is shown in Section 6. Section 7 presents automated gearbox case study.

2 MULTITASKING APPLICATION FINE GRAIN MODEL

The *fine grain model* treats tasks and interrupt service routines (ISR) internal structure, 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, and code branching and loops (affected by values of selected variables).

When a general property of the fine grain model is analyzed by exhaustive state space search (done by model checking tool), the execution time of a *computation* must be specified by an interval covering all possible cases, i.e. ⟨BCET, WCET⟩. Due to scheduling anomaly, WCET of *computations* do not necessary lead to the worst case finishing time of the whole task.

The structure of entire model is on Fig. 2.1. Rectangular blocks represent particular timed automata. Synchronization is expressed by arcs labeled by name of the synchronization channel. The most important data structures are shown in the right side of the figure. The essential components are explained in the following sections.

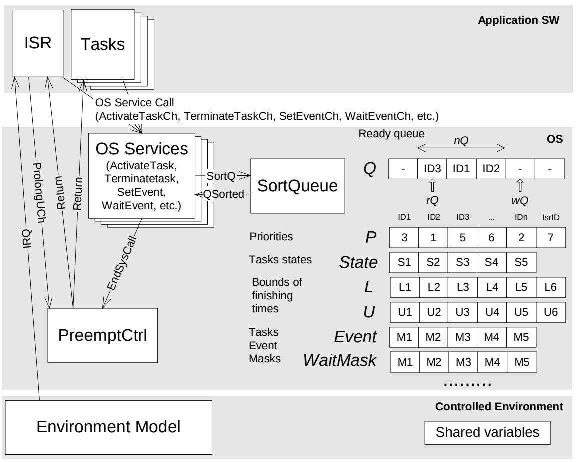


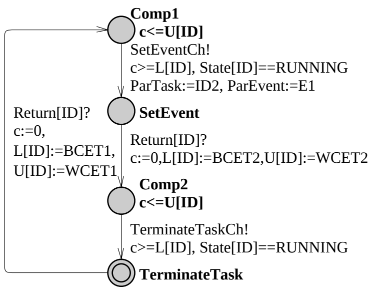
Fig. 2.1 Overview of entire timed automata model

3 TASK MODEL

Each task instance is modeled by one timed automaton that is synchronized with the OS model via channels depicted in Fig. 2.1. Fig. 3.1 presents an example of a simple task executing *computations Comp1* and *Comp2* and calling OS services *SetEvent(task,event)* and *TerminateTask*.

The UPPAAL notation is used in figures of timed automata models. The location with double circle represents initial location. Each location can be labeled by its name and a time invariant. The invariant in the form *"c<=U"*, allows to stay in the location only when a valuation of the clock variable *c* is smaller or equal to integer *U*. Each transition can be labeled by a synchronization (channel name with '?' or '!'), a guard (comma separated logical terms, e.g., *c>=L[1],State[1]==RUNNING*) and an assignment (comma separated assignments by sign ':=').

```
Task1()
{
   Comp1;
   SetEvent(ID2,E1);
   Comp2;
   TerminateTask();
}
```



a) Pseudo-code b) Task automaton

Fig. 3.1 Simple task example

Each *computation* is represented by one location of the same name (e.g. *Comp1*). The time spent in this location (measured by clock *c*) represents *computation's* finishing time (i.e. time necessary to its execution including preemption) and it is bounded by values stored in integers *L[ID]* and *U[ID]* (elements of arrays *L* and *U* respectively, where index *ID* is unique tasks identifier). These bounds are initialized to BCET and WCET, and they are increased when the task is preempted (provided by other timed automaton called *PreemptCtrl* as it is expressed later).

OS services calls are modeled by transitions synchronized by channels of corresponding names (e.g. *SetEventCh!*) and by locations of corresponding names (e.g. *SetEvent*) where the task is waiting return from services (channel *Return[ID]?*). OS service parameters are delivered through shared variables *ParTask* and *ParEvent*. Notice that OS services can block the calling task or cause higher-priority task becoming ready. Therefore return from OS service can occur after a preemption.

4 OS KERNEL MODEL

The OS kernel model consists of some variables representing OS objects (e.g. ready queue), timed automata representing OS services functionality, and of timed automata managing preemption (*PreemptCtrl*) and sorting ready queue according to priorities (*SortQueue*). See Fig. 2.1.

4.1 Kernel Variables

A task state and priority are stored in arrays *State* and *P* respectively, at index corresponding to the task *ID*. Higher number represents higher priority. The task state is either *SUSPENDED* (before activation), *WAITING* (after calling *WaitEvent*), *READY* (after activation and before first run), *PREEMPTED* (after preemption) or *RUNNING*.

*ID*s of all tasks, which are ready for execution (*State[ID]* is equal to *READY* or *PREEMPTED*), are stored in the ready queue modeled as a global array *Q* representing a circular buffer (see Fig. 2.1). Tasks are ordered in descending order according to their priorities in *Q* (*rQ* points to the ready task with the highest priority). The queue ordering is provided by automaton *SortQueue* (neglected in this paper). The reordering mechanism is started by synchronization channel *SortQ* after writing new *ID*.

For inter-task communication purposes, OSEK operating system provides events represented by one byte *Event[ID]* for each task. Each bit in *Event[ID]* represents one event that can be set or cleared. Moreover integer array *WaitMask* represents events, which the corresponding task is waiting for.

4.2 OS services

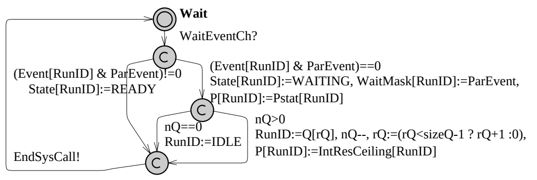
Each OS service is modeled by a timed automaton representing its functionality defined by OSEK specification (OSEK, 2003). The automaton is waiting in its initial state until its function is called from the task model. Then it manipulates tasks states, the ready queue (*Q*) and other operating system objects (e.g. events) and chooses the highest priority task to run and store its *ID* in variable *RunID.* Then it invokes *PreemptCtrl* automaton modeling the context switch and providing a preemption modeling.

As an example of a service model we introduce *WaitEvent(ParEvent)* service that cause the task wait for events in *ParEvent*. Fig. 4.1 shows *WaitEvent* OS service functionality in a pseudo-code. First the service checks, if at least one event specified in *ParEvent* is already set in the task's event mask *Event[RunID]*. If yes, the service simply returns. If no, the running task must wait for at least one. Therefore the task state is set to *WAITING*, *ParEvent* is stored in *WaitMask*, *internal resource* is released and the highest-priority task from ready queue is assigned to *RunID* variable. Then the context switch occurs.

```
WaitEvent (ParEvent)
{
  if ((Event[RunID] & ParEvent) == 0)
  {
    State[RunID] := WAITING;
    WaitMask[RunID] := ParEvent;
    Release Internal Resource;
    RunID := Extract Top of ReadyQ;
    ContextSwitch;                      // modeled in PreemptCtrl
    Get Internal Resource;
    State[RunID] := RUNNING;            // modeled in PreemptCtrl
  }
  return E_OK;
};
```

Fig. 4.1 *WaitEvent* pseudo-code

WaitEvent OS service automaton is depicted in Fig. 4.2. Locations marked by "**c**" are so called committed locations in UPPAAL notation. It must be left immediately, without any interference of other automaton that is not in a committed location. Since all locations in the automaton in Fig. 4.2, except the initial one, are committed locations, therefore the whole service seems to be atomic from the point of view of tasks and controlled environment models.



Wait
WaitEventCh?
(Event[RunID] & ParEvent)!=0
State[RunID]:=READY
(Event[RunID] & ParEvent)==0
State[RunID]:=WAITING, WaitMask[RunID]:=ParEvent,
P[RunID]:=Pstat[RunID]
nQ==0
RunID:=IDLE
nQ>0
RunID:=Q[rQ], nQ--, rQ:=(rQ<sizeQ-1 ? rQ+1 :0),
P[RunID]:=IntResCeiling[RunID]
EndSysCall!

Fig. 4.2 *WaitEvent* service automaton corresponding to pseudo-code depicted in Fig. 4.1

The OS service automaton waits in the initial state until the synchronization by channel *WaitEventCh* occurs. The context switch is modeled by *PreemptCtrl* automaton invoked by channel *EndSysCall.*

4.3 Preemption Modeling

PreemptCtrl automaton, depicted in Fig. 4.3, starts execution of scheduled task (*RunID*) and provides prolongation of finishing time bounds *L[i]* and *U[i]* of all preempted tasks. The automaton introduced here is simplified by omitting the part corresponding to interrupt service routines.

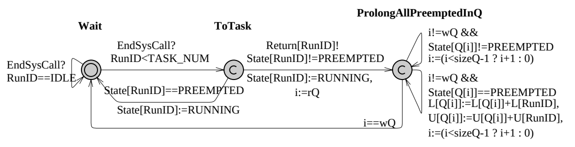


Fig. 4.3 *PreemptCtrl* automaton

At the end of each OS service, the function of *PreemptCtrl* automaton is invoked by synchronization *EndSysCall*. If a task should be scheduled (*RunID<TASK_NUM*), location *ToTask* is reached. If the task that should be scheduled now (*RunID*) has been preempted by a task released by an ISR in the past, its state has been set *PREEMPTED* by the ISR model. In this case, the *RunID* task model is in the location corresponding to some *computation* and its progress must be allowed now by setting its state *RUNNING* by *PreemptCtrl*.

If the *RunID* task model waits for synchronization *Return[RunID]* in a location corresponding to an OS service call (its state is *READY*), its state is also set *RUNNING*, and the progress in the task model is allowed by the synchronization via channel *Return[RunID]*. Since a new *computation* is started in *RunID* task in this case, bounds *L[i]* and *U[i]* of all *PREEMPTED* tasks *i* (i.e. all tasks that are in location corresponding to a *computation*) are moreover increased by bounds of the currently beginning *computation* (*L[RunID]* and *U[RunID]*).

5 INTERRUPT SERVICE ROUTINE MODEL

The ISR is modeled by timed automaton modeling an application dependent code in the same way as the task code. Moreover there is an initialization part that prevents a rescheduling inside the ISR and a finalization part that provides the rescheduling at the end of the ISR (as it is required by OSEK specification (OSEK, 2003)). An example of the ISR pseudo-code is in Fig. 7.2.

The state of *RunID* task is set to PREEMPTED, *RunID* content is stored in variable *InterruptedID* and an identifier of the ISR (*IsrID*) is written to variable *RunID* in the initialization part. Since the priority of the ISR is higher than all task priorities, OS services called from the ISR code, cannot cause a rescheduling. At the finalization part, the highest priority ready task *ID* is written to variable *RunID*. It can be either *InterruptedID* or *ID* from the top of the ready queue. In the second case, the *InterruptedID* is written to the ready queue.

6 MODEL OVERAPPROXIMATION

When the preemption occurs the finishing time bounds *L[Preempted]* and *U[Preempted]* of the preempted *computation* should be prolonged by the duration of the preemption. Since the right duration of the preemption cannot be measured in timed automata (a clock variable cannot be stopped or stored), the bounds *L[Preempted]* and *U[Preempted]* are increased by bounds of the possible preemption that are *L[Preempting]* and *U[Preempting]*, the finishing time bounds (in this time equal to execution time bounds) of the preempting task *computation*. This introduces an additional non-determinism to the model since the duration of the preempted task preemption is not necessary equal to the duration of the preempting task execution (what holds in the real system). Therefore the set of real system behaviors is subset of the modeled behaviors, i.e. the model is an over-approximation.

To illustrate the over-approximation let us consider for example low-priority task T_{low} with execution time $C_{low} \in [1,4]$ preempted by high-priority task T_{high} with execution time $C_{high} \in [2,4]$. All possible relative finishing times of both tasks in the real system and in the proposed model are depicted in Fig. 6.1. Finishing time of T_{high} is always equal to its execution time C_{high}. Finishing of T_{low} is equal to its execution time C_{low} plus preemption duration. Preemption duration is bounded by bounds of C_{high} in the model but it is equal to the actual execution time of T_{high} in the real system.

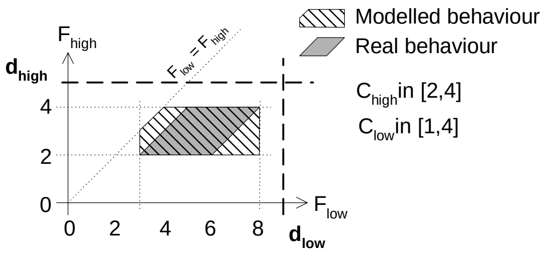


Fig. 6.1 Possible values of relative finishing times *F* of preempting task T_{high} and preempted task T_{low}

Fig. 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. 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 Section 7).

Schedulability is an often verified property, exploring whether tasks are finished prior to their deadlines (d_{high} and d_{low} in Fig. 6.1) in all situations. Fig. 6.1 shows that the worst case finishing time of each task 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 in this section. The controlled system consists of a dry clutch actuated by a servo and five-speed gearbox.

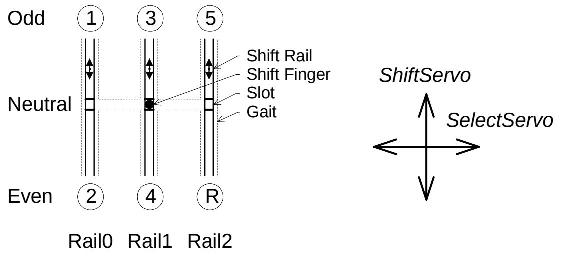


Fig. 7.1 Gear box mechanics

The gearbox mechanics is depicted in Fig. 7.1. It consists 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. *ShiftServo* engages one of two gears (even or odd) or neutral by moving the selected rail by the shift finger. A direction of the shift finger movement is limited by a gait.

```
ISR()        // IRQ sources - TIMER, Clutch, SelectServo, ShiftServo
{
   // Initialization part
   State[RunID]:=PREEMPTED;
   InterruptedID := RunID;
   RunID := IsrID;     // OS Services called from ISR returns to ISR
                       // and OS services do not schedule any task
   // User 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;
         SetEvet (GearBoxCtrlTask, ClutchEvent);
      }
      else if (bShiftServoInt) {
         bShiftServoInt:=0;
         SetEvet (GearBoxCtrlTask, ShiftServoEvent);
      }
      else if (bSelectServoInt) {
         bSelectServoInt:=0;
         SetEvet (GearBoxCtrlTask, SelectServoEvent);
      }
   }
   // Finalization part - Scheduling point
   if (readyQ.Empty)
      RunID := interruptedID;
   else if (InterruptedID == IDLE) {
      RunID := Extract Top of ReadyQ;
   }
   else if (P[ReadyQ.Top] > P[InterruptedID]) {
      Write InterruptedID to ReadyQ;
      RunID := Extract Top of ReadyQ;
   }
   else
      RunID := InterruptedID;
   InterruptReturn;           // modeled by channel EndSysCall
};
```
Fig. 7.2 Interrupt service routine pseudocode

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

The ISR (see pseudocode in Fig. 7.2) 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 boolean variables bTimerInt, bClutchInt, bShiftServoInt and bSelectServoInt.

Task *SlipCtrlTask* is periodically activated by the ISR. Its priority is 2 and its period is 10. It provides slip control and torque tracking. Task *SelectGearTask* is periodically activated by ISR. Its priority is 0 and its period is 200. It selects appropriate transmission rate, write it to variable *DesiredGear*, and if desired gear differs from the current one, it activates task *GearBoxCtrlTask* that controls the gear changing. Since a detailed functionality of *SlipCtrlTask* and *SelectGearTask* is not necessary for the verification, their models are very simple (only execution times are considered) and they are omitted here.

Task *GearBoxCtrlTask* has priority 1. Its functionality is described in details in Fig. 7.3. Notice that the task suspends himself several times, while waiting on an external event.

```
GearBoxCtrlTask()      // Activated by SelectGearTask
{
   GBReady := 0;
   ClearEvet (ClutchEvet);
   OpenClutch;                                        // Send command
   WaitEvent (ClutchEvent);
   if (CurrentShift != NEUTRAL)
   {
      // Disengage
      ClearEvent (ShiftServoEvent);
      ShiftServo_Goto (NEUTRAL);                      // Send command
      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
         WaitEvent (SelectServoEvent);
      }
      // Shift
      ClearEvent (ShiftServoEvent);
      ShiftServo_Goto (DesiredShift);                 // Send command
      WaitEvent (ShiftServoEvent);
   }
   ClearEvet (ClutchEvet);
   CloseClutch;                                       // Send command
   WaitEvent (ClutchEvent);
   GBReady := 1;
   CurrentGear:=DesiredGear;
   TerminateTask();
};
```
Fig. 7.3 Gear Box Control task pseudocode

7.2 Model

A model of the whole system consists of timed automata representing the controlled system (*Clutch*, *SelectServo* and *ShiftServo*), hardware devices

(Timer), the OS (services *ActivateTask*, *TerminateTask*, *SetEvent*, *WaitEvent* and automata *PreemptCtrl* and *SortQueue*), the tasks (*SlipCtrlTask*, *SelectGearTask*, *GearBoxCtrlTask*) and the ISR. An overview of the whole model is depicted in Fig. 7.4 (automata synchronization via channels) and Fig. 7.5 (shared variables). Except events of *GearBoxCtrlTask*, variables and timed automata modeling the OS are omitted in both figures.

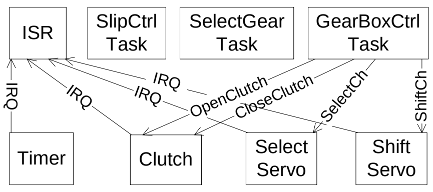


Fig. 7.4 Model overview – synchronizations via channels

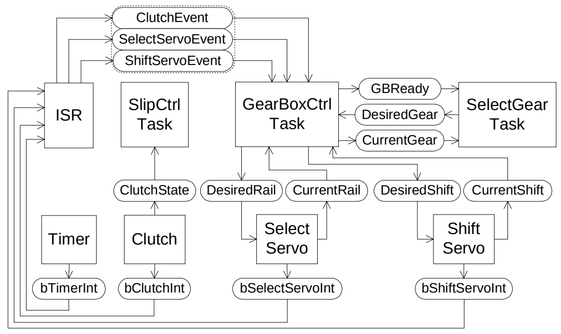


Fig. 7.5 Model overview – shared variables

Clutch timed automaton is depicted in Fig. 7.6. It is in location *Closed* or *Opened* in steady state. When the command to open or close the clutch is received (via channel *OpenClutch* or *CloseClutch* respectively), *Clutch* becomes *Opening* or *Closing* respectively. After *ShiftTime*, interrupt request (IRQ) is generated via channel *IRQ*.

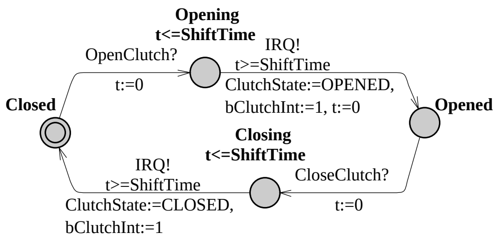


Fig. 7.6 *Clutch* timed automaton

SelectServo timed automaton is depicted in Fig. 7.7. Locations *Rail0*, *Rail1* and *Rail2* represent steady states. When the command to select a new rail is received via channel *SelectCh*, *SelectServo* moves to *DesiredRail* (*BetweenRail0andRail1* and *BetweenRail1andRail2*). When *DesiredRail* is reached, IRQ is generated via channel *IRQ*.

ShiftServo timed automaton (not depicted here) differs from *SelectServo* timed automaton only in several details. Locations *Rail0*, *Rail1* and *Rail2* are changed to *OddPos*, *NeutralPos* and *EvenPos*, and variables and channels related selecting (*DesiredRail*, *CurrentRail*, *SelectCh*, *bSelectServoInt*) are changed to variables and channels related to shifting (*DesiredShift*, *CurrentShift*, *ShiftCh*, *bShiftServoInt*).

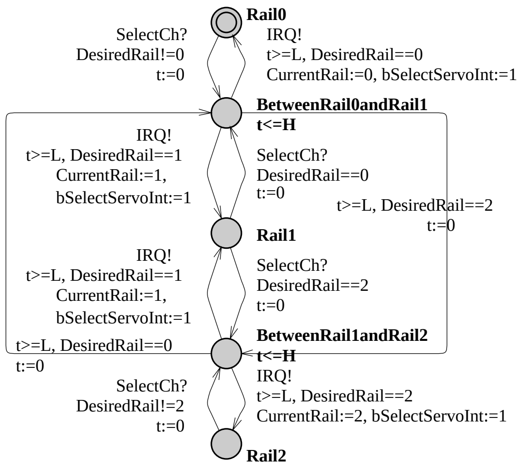


Fig. 7.7 *SelectServo* timed automaton

Tasks and the ISR respectively are translated to the timed automata models according to the methodology described in Sections 3 and 5. *GearBoxCtrlTask* timed automaton is in Fig. 7.8; *SlipCtrlTask*, *SelectGearTask* and *ISR* timed automata are omitted here. Complete model can be downloaded at: http://dce.felk.cvut.cz/waszniowski/RTVerif/RTVerif.htm

7.3 Formal verification

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

Safety properties:

P1. Shifting is not allowed when the clutch is closed

P2. Selecting is allowed only when *ShiftServo* is in neutral

P3. Shifting is allowed only when a rail is selected

P4. The clutch cannot be opened longer than 310 time units

Bounded liveness:

P5 – P11. When a new desired gear is chosen, it is engaged in 260 time units

Listed properties have been formalized in UPPAAL requirement specification language as follow:

P1. A[] Clutch.Closed imply (ShiftServo.OddPos or ShiftServo.NeutralPos or ShiftServo.EvenPos)

P2. A[] not (SelectServo.Rail0 or SelectServo.Rail1 or SelectServo.Rail2) imply ShiftServo.NeutralPos

P3. A[] not ShiftServo.NeutralPos imply (SelectServo.Rail0 or SelectServo.Rail1 or SelectServo.Rail2)

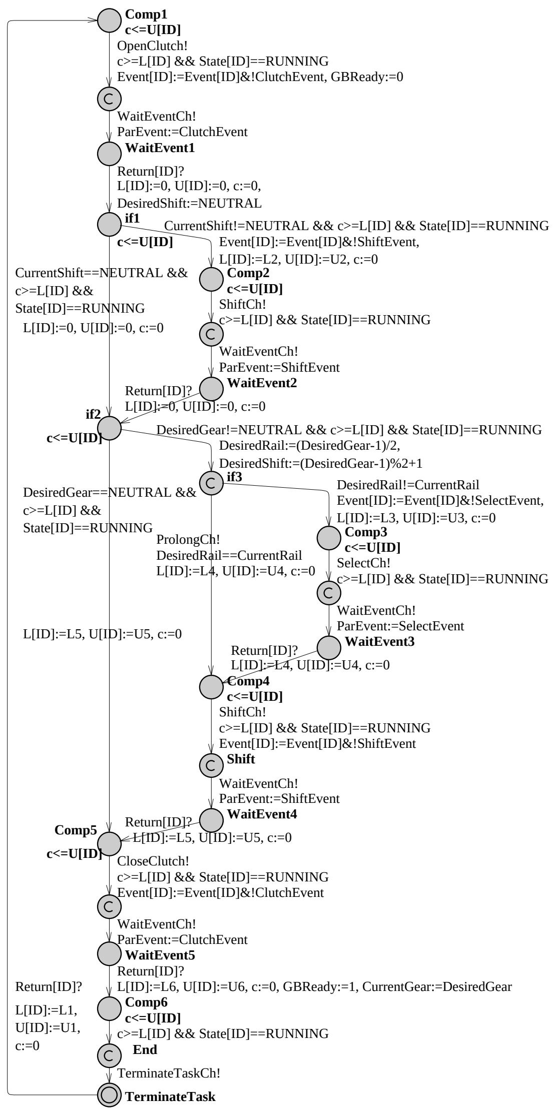
P4. A[] Clutch.Opened imply Clutch.t<=310

P5. (DesiredGear==0 and SelectGearTask.ActivateTask) --> (ShiftServo.NeutralPos and rt<=260)

P6. (DesiredGear==1 and SelectGearTask.ActivateTask) --> (ShiftServo.OddPos and SelectServo.Rail0 and rt<=260)

P7 - P11 Similar to P6.

In UPPAAL requirement specification language syntax A[] *f* represents the computation tree logic (CTL) formula $\forall\Box f$ (i.e. "invariantly holds *f*"). The syntax *p* --> *q* denotes a CTL property $\forall\Box (p \Rightarrow \forall\Diamond q)$ (i.e. "whenever *p* holds, eventually *q* will hold as well"). Clock *rt*, measuring the response time in all bounded liveness properties P5 – P11, is reset when *DesiredGear* is changed in *SelectGearTask* timed automaton.



Comp1
c<=U[ID]
OpenClutch!
c>=L[ID] && State[ID]==RUNNING
Event[ID]:=Event[ID]&!ClutchEvent, GBReady:=0
C
WaitEventCh!
ParEvent:=ClutchEvent
WaitEvent1
Return[ID]?
L[ID]:=0, U[ID]:=0, c:=0,
DesiredShift:=NEUTRAL
if1
c<=U[ID]
CurrentShift!=NEUTRAL && c>=L[ID] && State[ID]==RUNNING
Event[ID]:=Event[ID]&!ShiftEvent,
L[ID]:=L2, U[ID]:=U2, c:=0
CurrentShift==NEUTRAL &&
c>=L[ID] &&
State[ID]==RUNNING
L[ID]:=0, U[ID]:=0, c:=0
Comp2
c<=U[ID]
ShiftCh!
c>=L[ID] && State[ID]==RUNNING
C
WaitEventCh!
ParEvent:=ShiftEvent
WaitEvent2
Return[ID]?
L[ID]:=0, U[ID]:=0, c:=0
if2
c<=U[ID]
DesiredGear!=NEUTRAL && c>=L[ID] && State[ID]==RUNNING
DesiredRail:=(DesiredGear-1)/2,
DesiredShift:=(DesiredGear-1)%2+1
if3
DesiredGear==NEUTRAL &&
c>=L[ID] &&
State[ID]==RUNNING
DesiredRail!=CurrentRail
Event[ID]:=Event[ID]&!SelectEvent,
L[ID]:=L3, U[ID]:=U3, c:=0
ProlongCh!
DesiredRail==CurrentRail
L[ID]:=L4, U[ID]:=U4, c:=0
Comp3
c<=U[ID]
SelectCh!
c>=L[ID] && State[ID]==RUNNING
C
WaitEventCh!
ParEvent:=SelectEvent
WaitEvent3
L[ID]:=L5, U[ID]:=U5, c:=0
Return[ID]?
L[ID]:=L4, U[ID]:=U4, c:=0
Comp4
c<=U[ID]
ShiftCh!
c>=L[ID] && State[ID]==RUNNING
Event[ID]:=Event[ID]&!ShiftEvent
Shift
C
WaitEventCh!
ParEvent:=ShiftEvent
WaitEvent4
Comp5
c<=U[ID]
Return[ID]?
L[ID]:=L5, U[ID]:=U5, c:=0
CloseClutch!
c>=L[ID] && State[ID]==RUNNING
Event[ID]:=Event[ID]&!ClutchEvent
C
WaitEventCh!
ParEvent:=ClutchEvent
WaitEvent5
Return[ID]?
L[ID]:=L6, U[ID]:=U6, c:=0, GBReady:=1, CurrentGear:=DesiredGear
Return[ID]?
L[ID]:=L1,
U[ID]:=U1,
c:=0
Comp6
c<=U[ID]
c>=L[ID] && State[ID]==RUNNING
C
End
TerminateTaskCh!
TerminateTask

Fig. 7.8 *GearBoxCtrlTask* timed automaton

All mentioned properties of the system have been successfully verified by model-checker UPPAAL 3.4.5 running on PC AMD Athlon 1GHz, with 1.3GB RAM. The time required for verification of all 11 properties has been 4minutes and 14 seconds. The required memory has been 203MB.

Even though our model is over-approximate (it contains more behavior than the real system), we can conclude that all safety and bounded liveness properties that are satisfied by the model are also satisfied by the real system (Berard, *et al.*, 2001).

7.4 Response time analysis

The proposed model can be used to analyze the worst-case response time (WCRT) of *GearBoxCtrlTask* by verifying formula: "Always, when *GearBoxCtrlTask* is activated, its end is unavoidably reached in the future and *rt*<=WCRT in that time". The response time is measured by clock *rt* that is reset when *GearBoxCtrlTask* is activated in *SelectGearTask*. The verification is done in UPPAAL for specific value of WCRT. The smallest value can be found in several experiments. WCRT found by this method is 312.

There also exist algorithms for parametric model-checking verifying whether a state is reachable in model with uncertain parameter (WCRT in observer automaton). However this problem is undecidable in general (Alur, *et al.*, 1993).

It is clear from the *GearBoxCtrlTask* pseudocode that the task suspends itself several times wile waiting for external events. Scheduling theory (Klein, *et al.*, 1993) proposes two ways of analyzing response times of tasks with self-suspension: (*i*) treat suspension time as execution time or (*ii*) analyze separately each computation in worst case phasing and sum all partial results together with suspension times. Both of these methods are pessimistic however (Klein, *et al.*, 1993).

The worst-case execution path of *GearBoxCtrlTask* consists of six *computations* separated by five self-suspensions. WCET of all *computations* are 1. The worst-case self-suspension time of one self-suspension is 100, the other are 50. *GearBoxCtrlTask* can be preempted by *SlipCtrlTask* and by *ISR* for 1. WCRT of *GearBoxCtrlTask* computed by method (*ii*) is 321, and by method (*i*) even 389.

8 CONCLUSION

We have demonstrated in this paper, how timed automata can be used for the multitasking preemptive application modeling. Even though the model is an over-approximation of the real system behavior, complex time and logical properties considering application data and controlled system model can be verified by model-checking tool, since safety and bounded liveness properties (the most important groups) are preserved by an over-approximation.

Opposite to hybrid automata allowing precise modeling of the preemption (Corbet, 1996), termination of the verification algorithm is guaranteed for timed automata. Opposite to models based on timed automata extended by tasks (Fersman, *et al.*, 2002), the internal structure of the preemptive task can be modeled. Opposite to models used in standard response time analysis based on scheduling theory, an advantage of timed automata based model is its ability to model the task internal structure and the controlled environment. Consequently more general properties can be verifier an less pessimistic response time analysis is provided by model-checking approach, especially when the analyzed application contains features that make the response time analysis pessimistic.

Off course, an exhaustive analysis of the detailed timed automata model subjects to a state space explosion (what is a general property of most formal methods (Corbet, 1996)). Therefore the proposed model is abstract as much as possible and contains only information necessary for a correct verification of the system specification. The operating system model use only modest data structures, it does not use any clock variables (duration of OS services and context switch is involved in the execution time of *computations*), it does not allow any non-determinism and all locations are committed what prevents paths interleaving and therefore restricts explored state space. 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 the compilation time) and it is designed for a modest runtime environment of embedded devices. The model of 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 the number of *computations* to obtain a model of reasonable size.

In spite of these restrictions, model-checking approach is applicable for formal verification of realistic applications whose verification done manually by human would be hard and error prone.

ACKNOWLEDGEMENT

This work was supported by the Ministry of Education of the Czech Republic under Project 1M6840770004

REFERENCES

Alur R., T. A. Henzinger and M.Y. Vardi (1993): Parametric real-time reasoning. *Proceedings of the 25th ACM Symposium on Theory of Computing*, pp. 592-601.

Alur, R. and D.L. Dill (1994). A theory of timed automata. *Theoretical Computer Science*, **126**, 183-235.

Bailey C.M., A. Burns, A.J. Wellings and C.H. Forsyth (1995). A Performance Analysis of a Hard Real-Time System. *Control Engineering Practice*, **3**(4), 447-464.

Behrmann, G., A. David, K.G. Larsen, O. Möller, P. Pettersson and W. Yi (2001). Uppaal - Present and Future. In: *Proceedings of the 40th IEEE Conference on Decision and Control (CDC'2001)*. pp. 2881-2886, Orlando, Florida.

Berard, B., M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen and P. McKenzie (2001). *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Verlag.

Bucci, G., A. Fedeli, L. Sassoli and E. Vicario (2004): Timed State Space Analysis of Real-Time Preemptive Systems. *IEEE Transaction on Software Engineering*. 30(2): 97-111

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

Daws, C., A. Olivero, S. Tripakis and S. Yovine (1996). The tool Kronos. In: *Proceedings of Hybrid Systems III, Verification and Control*, LNCS 1066, 208-219. Springer-Verlag, New York.

Fersman, E., P. Pettersson, and W. Yi (2002). 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*, LNCS 2280, pp.67-82, Springer-Verlag

Fersman, E., P. Pettersson, and W. Yi (2003). Schedulability Analysis using two clocks. In: *Proceedings of TACAS'03*, LNCS 2619 pp 224-239. Springer-Verlag.

Klein, M., T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour (1993). *A Practitioner's Handbook for Real-Time Systems Analysis*. Kluwer Academic Publishers, Boston.

Krčál, P. and W. Yi (2004): Decidable and Undecidable Problems in Schedulability Analysis Using Timed Automata. In: *Proceedings of TACAS'04*, LNCS 2988, pp 236-250. Springer-Verlag.

Larsen, K.G., P. Pettersson, and Yi, W. (1995). Model-Checking for Real-Time Systems. In *Proceedings of the 10th International Conference on Fundamentals of Computation Theory*, LNCS 965, 62-88. Springer Verlag

Lime, D and O.H. Roux. (2004) A translation based method for the timed analysis of scheduling extended time Petri nets. In *Proceedings of the 25th IEEE International Real-time Systems Symposium*, 187--196, December 2004, Lisbon, Portugal.

Liu, J.W.S. (2000). *Real-time systems*. Prentice-Hall, Inc., Upper Saddle River, New Jersey

Lundqvist, K. and L. Asplund (2003). A Ravenscar-Compliant Run-time Kernel for Safety-Critical Systems. *Real-Time Systems Journal*, **24**(1): 29-54.

OSEK (2003). *OSEK/VDX Operating System Specification 2.2.1*. http://www.osek-vdx.org/