Over-approximate Model of Multitasking Application Based on Timed Automata Using Only One Clock

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 real-time operating system compliant with 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, we use model checking theory based on timed automata and we verify time and logical properties of proposed model by existing model checking tools. Since a complexity of the model-checking verification exponentially grows with the number of clocks used in a model, the proposed model uses only one clock for measuring execution time of all modeled tasks.

Key words: Timed Automata, OSEK/VDX Operating System, Model-Checking

1 Introduction

This paper deals with formal modeling of applications running under real-time operating system (OS). Typical application under assumption is a complex controller consisting of periodic and aperiodic tasks constrained by deadlines and synchronized via inter-task communication primitives.

Model checking [6] approach, shown in this paper, provides timed automata [1] model of operating system, application tasks and controlled environment. 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 properties of discrete event system. It enables to check rather complex properties (safety and bounded liveness properties) by model checking tools (e.g. UPPAAL [2] or Kronos [3]) in finite time.

Both non-preemptive and preemptive scheduling strategies can be modeled by this approach. In the case of the preemptive scheduling, the model is an over-approximation from the model checking point of view. This drawback is the price paid for decidability of the model checking problem. Also methods for schedulability analysis e.g. rate monotonic analysis (RMA [5]), are based on over-approximate model, and in some cases, they are more pessimistic than model checking approach.

The complexity of the timed automata model checking is exponential on the number of clocks and on the largest time constant appearing in the timing constraints. Even though the complexity growing due to time constants can be avoided by symbolic representation of sets of states, the number of clocks remains a problem determining the feasibility of the model checking for the praxis [12]. Event though a method for reducing the number of clocks of a timed automata has been proposed in [12], we have minimized the number of clocks already at the phase of constructing the model of tasks.

The straightforward approach for modeling a multitasking application by timed automata is to model each task by one timed automaton and to measure execution time of each task by one clock. Completion of the task (or its part) is modeled by expiration of corresponding execution time measured by corresponding clock. But only one task can be running on the processor at given time and only the running task can be finished. Therefore only the clock of the currently running task can affect progress of the model. Timed automata of the other tasks cannot progress.

These facts have motivated us to use only one clock measuring execution time of the currently running task. When the running task is preempted in such model, its remaining execution time is computed and stored in a queue (called delta list). Then the clock can be used for the preempting task.

Therefore only one clock is used to model time evolution in all tasks. Of course other clocks are used to model controlled environment and for verification purposes.

Related work. In [10] and [11] the timed automata are extended by asynchronous tasks (i.e. tasks triggered by events) to provide a model for event-driven systems. Schedulability analysis of tasks associated to the model is encoded as reachability of timed automata with subtraction or as reachability of standard timed automata in the case of fixed priority scheduling policy [11]. Moreover in [11] it is shown, that only two clocks are necessary to implement schedulability analysis of n independent tasks. However $n+1$ clocks are necessary in the case of sharing variables between tasks and timed automata. This approach provides good results of schedulability analysis for aperiodic tasks but it is not suited to model the task internal structure as it follows from results of [21].

Corbet in [8] provides a method for constructing models of multitasking programs based on hybrid automata. Since the reachability problem is undecidable for hybrid automata, the termination of analyzing algorithm is not guarantied in general. Opposite to hybrid automata, the reachability problem is decidable for timed automata.

Timed Automata are used to model primitives of Ravenscar tasking system in [9]. However, the variable used to measure execution time of tasks (system clock) is integer, which is periodically incremented by timed automaton modeling system cock after each "tick". It allows to model preemption simply but the granularity of "ticks" must be very high to model all time instances (nothing can occur between two "ticks"). Discrete time for modeling real-time application is used also in the work [19] 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 used also in work [18] where a generalized approach to schedulability analysis based on process algebra is proposed. Even though these approaches consider a task internal structure, the controlled environment affecting releasing times of tasks is not modeled.

Outline. Following sections briefly present main ideas of our approach and the most important properties of the proposed model. Its applicability is demonstrated on simple static multitasking operating system compliant with OSEK/VDX [13] (the standard used in automotive applications). Fine grain model of the multitasking application is proposed in section 2. Section 3 describes model of one task and section 4 describes model of OS kernel objects and OS services. The main contribution of this paper is *one-clock* approach to preemption modeling. It is described in section 5. The over-approximation of the model is discussed in section 6. Experimental results, demonstrating performance improvement of model checking based on proposed *one-clock* approach, are presented in section 7.

2 Fine grain model of a multitasking application

Fine grain model treats tasks and the 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, code branching and loops (affected by values of selected variables). *Computations* are defined by BCET (best-case execution time) and WCET (worst-case execution time). It allows incorporating uncertainty of execution time.

Simplified structure of the 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.

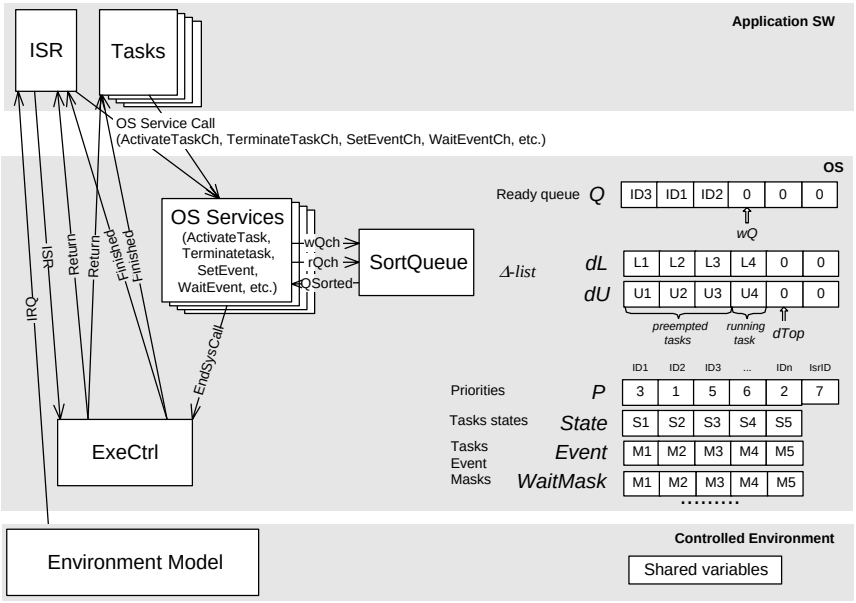


Fig. 2.1 Overview of entire timed automata model

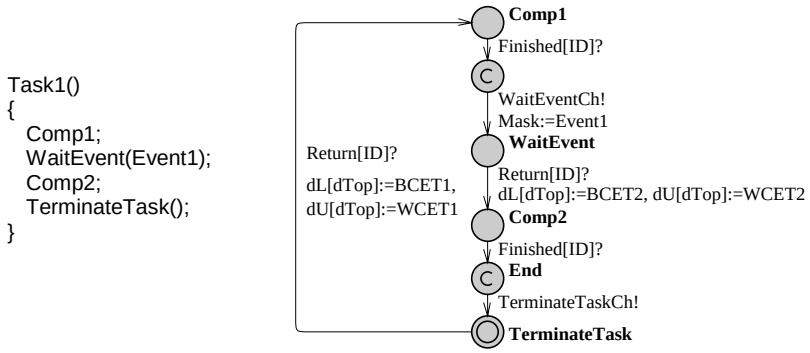
When a general property of the model is analyzed by an exhaustive state space search (done by model checking tool), an execution time of a task must be specified by an interval covering all possible cases, i.e. ⟨BCET, WCET⟩. Due to possibility of a scheduling anomaly, WCET of *computations* do not necessary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

Exhaustive analysis of *fine grain model* behavior (automatically done by model checking tool UPPAAL) considers the task response times corresponding to realistic phasing, realistic blocking and realistic execution time in relation to modeled code branching. Therefore the result is as precise as the model is precise (it will be shown later that there is some over-approximation in the model based on timed automata). The price paid for exhaustive analysis is higher complexity.

Since the schedulability is one of the most often analyzed property, it is attractive to compare our *fine grain model* based on timed automata with the classical scheduling theory task model considering just WCET of the whole task, its minimum inter-arrival time and its blocking time related to resources ([4],[5],[7]). Classical schedulability analysis based on such model computes the worst case response time of the task by adding together its worst case execution time, duration of preemption in the worst case inter-arrival times and phasing, and the worst case blocking on shared resources. Such worst-case response time could be too pessimistic abstraction in many applications, since all the mentioned worst cases do not occur at the same time [14],[16].

3 Timed automata model of task

Each task is modeled by one timed automaton synchronized with the OS model via channels depicted in Fig. 2.1 (arrows between blocks). Fig. 3.1 demonstrates the modeling methodology on the example of a simple task executing *computations Comp1* and *Comp2* and calling OS services *WaitEvent(Mask)* and *TerminateTask*. The UPPAAL notation is used [2]. The location with double circle represents the 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 ':='). Locations marked by "**c**" are so called commited locations providing atomicity of the in-coming and out-coming transitions (committed location must be left immediately without any interference of other automaton in the model).



a) Pseudocode b) Task automaton

Fig. 3.1 Simple task example

Each *computation* is represented by one location of the same name (e.g. *Comp1*). A time spent in this location represents *computation's* response time (i.e. time necessary to its execution including preemption). BCET and WCET of the *computation* are assigned to the *dTop* position of the integer array *dL* and *dU* respectively. (i.e. *dL[dTop]* and *dU[dTop]* respectively) at the transition leading to a computation location. According to these values (and scheduling policy) *ExeCtrl* automaton determines the time when the computation is finished (taking preemption into account) and by synchronization via channel *Finshed[ID]* (where index *ID* is unique task identifier) makes the task model leave the location corresponding to the *computation*.

OS service calls are modeled by transitions synchronized via channels of corresponding names (e.g. *WaitEventCh!*) and by locations of the corresponding names (e.g. *SetEvent*), where the task is waiting return from the service (channel *Return[ID]?*). OS service parameters are assigned to shared variables *ParTask* and *Mask*. Return from OS service causes start of a new *computation*.

This methodology allows to model an arbitrary structure of the task e.g. loops and branching (nondeterministic or affected by values variables).

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 timed automata managing preemption (*ExeCtrl*) and sorting ready queue according to the priority (*SortQueue*).

4.1 Kernel variables

Tasks priorities are stored in a global array *P*, indexed by *ID*. Higher number represents higher priority.

The task state is stored in the array *State* at index corresponding to its *ID*. The task state is either *PREEMPTED*, *SUSPENDED*, *WAITING*, *READY* or *RUNNING*.

Variable *RunID* stores *ID* of currently running task or interrupt service routine

*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* (see Fig. 2.1). Tasks are ordered in descending order according to their priorities in *Q*. *Q[0]* is the ready task with the highest priority and the first empty position in *Q* is stored in variable *wQ*.

Queue must be reordered according to tasks priorities after writing new task. All elements of the queue must be shifted to left after reading the highest priority ready task from the zero position. Both these mechanisms are provided by automaton *SortQueue* depicted on Fig. 4.1.

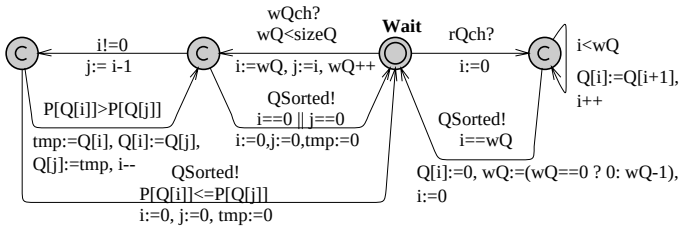


Fig. 4.1 *SortQ* automaton

Reordering mechanism is started by synchronization channel *wQch* after writing new *ID* to *Q[wQ]*. Pointer *wQ* is then increased and priorities of tasks in neighbor position in *Q* are compared (started from *wQ*) and if there is higher priority in higher position, *IDs* are swapped. Finishing of this mechanism is announced by channel *QSorted*. Shifting of *Q* after reading *Q[0]* is started by synchronization channel *rQch* and its finishing is announced by channel *QSorted*.

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

For inter-task communication purposes, OSEK operating system [13] provides some objects that are also modeled by some variables. Due to the lack of place in this paper, we mention only events here. Events are represented by array *Event* associating one byte *Event[ID]* to each task. Each bit in *Event[ID]* represent one event that can be set or cleared. Moreover array *WaitMask* represents events, which the corresponding task is waiting for.

Δ-list does not represent any object of OS, but it is a data structure used to modeling evolution of tasks execution (see section 5).

4.2 OS services

Each OS service is modeled by timed automaton representing its functionality defined by OSEK specification [13]. The automaton is waiting in its initial state until its function is called from the task model. Then it manipulates tasks states, ready queue 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 the *ExeCtrl* automaton modeling the context switch and providing preemption control.

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

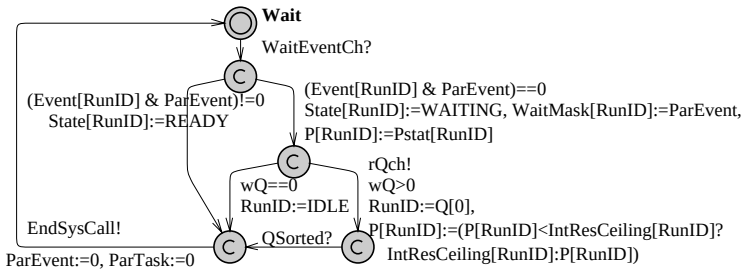


Fig. 4.2 *WaitEvent* pseudo-code

As an example of a service model we introduce *WaitEvent(Mask)* service that cause the task wait for events in *Mask*. Fig. 4.2 shows *WaitEvent* OS service functionality in a pseudo-code and the corresponding automaton. It is supposed that interrupts are disabled within the whole service. All locations in the model, except the initial one, are committed locations, therefore the whole service seems to be atomic from the point of view of the rest of the system. The execution time of the OS service is involved in the execution time of the preceding *computation*. The OS service automaton waits in the initial state until the synchronization by channel *WaitEventCh*. The context switch is modeled by *ExeCtrl* automaton invoked by channel *EndSysCall*.

5 Preemption modeling

This section describes the basic idea of a preemptive multitasking application modeling by using only one clock. The time progress in tasks models is controlled by *ExeCtrl* timed automaton depicted in Fig. 5.3. It manage remaining execution times of started *computations* in data structure called *Δ-list* that consists of two arrays of integers *dL* and *dU* (see Fig. 2.1). *dL* is used for lower bounds and *dU* is used for upper bounds. Variable *dTop* points to the top of *Δ-list*. BCET and WCET of the currently starting *computation* are assigned to *dL[dTop]* and *dU[dTop]* respectively by the task model (see Fig. 3.1) and *dTop* is increased (by automaton *ExeCtrl*). At position *dTop-1* is therefore the remaining execution time of the currently executed *computation* of the currently running task. At positions form 0 to *dTop-2* are stored remaining execution times of started *computations* of preempted tasks. Since the remaining execution times are written to *Δ-list* in LIFO (Last In First Out) order when the preemption occurs, it holds that they are ordered according to the tasks priorities. At position *dTop-1* is therefore always the remaining execution time of the task with the highest priority (*RunID*).

ExeCtrl timed automaton uses clock *c* for measuring the execution time of a started *computation* of the currently running task (*RunID*). When the *computation* is finished (its execution time stored in *Δ-list* at position *dTop-1* expired), *ExeCtrl* timed automaton synchronize with the task timed automaton by channel *Finished[RunID]*. This synchronization allows the progress in the task model.

But when the running task is preempted before the started *computation* is finished, the lover and the upper bounds of the time remaining to finishing its execution are decreased by the already executed time elapsed from the last start (measured by clock *c*). Since the preempted task cannot progress, clock *c* can be used for the new running task.

The only problem is that the valuation of clock *c* is in the domain of positive real numbers while *Δ-list* consists of integers (since real variables would lead to infinite state space). Therefore the valuation of clock *c* is over-approximated by an interval bounded by the nearest lower and the nearest upper integers of clock *c* valuation (*lc* and *uc*). Algorithm computing integers *lc* and *uc* from clock *c* by bisection is in Fig. 5.2. Its complexity is $O(\log_2 MaxC)$ where *MaxC* is the upper margin of clock *c* valuation.

Fig. 5.1 shows on an example of three tasks, how the remaining execution time of preempted task T_2 is computed and how *Δ-list* is modified.

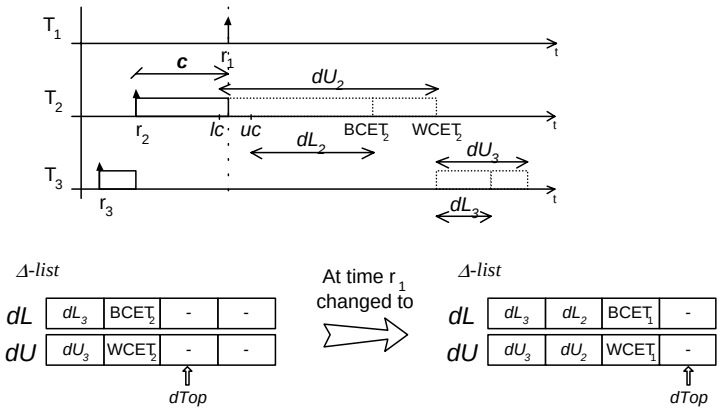


Fig. 5.1 Example of *Δ-list* modification due to task T_2 preemption by task T_1 at time r_1

Fig. 5.2 summarizes the described mechanism in a comprehensive form. There is an algorithm for modifying *Δ-list* when a new computation is started (NewCompStarted) and an algorithm used when the current *computation* is finished (CurrentCompFinished).

```
Clock2Int (clock c, int MaxC)
{
  int lc:=0;            // lover integer
  int uc:=MaxC;         // upper integer
  while ((uc-lc) > 1)
  {
    if ((lc+uc)/2 >= c)
      uc:= (lc+uc)/2;
    if ((lc+uc)/2 <= c)
      lc:= (lc+uc)/2;
  };
  if (lc == c)
    uc:=lc;
  if (uc == c)
    lc:=uc;
  return lc, uc;
};
```

```
NewCompStarted
{
  dL[dTop]:=BCET;
  dU[dTop]:=WCET;
  if (dTop>0)
  { //Preemption
    lc, uc := Clock2Int (c, dU[dTop]);
    if (dL[dTop-1] > uc)
      dL[dTop-1] := dL[dTop-1] – uc;
    else
      dL[dTop-1] := 0;
    dU[dTop-1] := dU[dTop-1] – lc;
  }
  c:=0;
  dTop++;
}
```

```
Current computation (started
computation of task RunID) is finished
when
clock c∈⟨dL[dTop-1],dU[dTop-1]⟩

CurrentCompFinished
{
  Finished[RunID]!  // Progress in task
  dTop--;
  c:=0;
  if (dTop>0)
    Wait_ CurrentCompFinished;
  else
    Idle;
}
```

Fig. 5.2 Mechanisms of modeling preemption by only one clock

Described mechanism of the preemption modeling by only one clock is implemented by *ExeCtrl* timed automaton depicted in Fig. 5.3. It provides the following functions:

- *(i)* models the context switch, i.e. send *Return[RunID]* to the scheduled task at the end of an OS service call (transitions related to locations *EndOSserv*, *ToTask* and *UpdateDlist*),
- *(ii)* manages *Δ-list* when a preemption occurs (transitions related to locations *UpdateDlist*, *C2Int_Begin*, *C2Int_End* and *CompStarted*) and
- *(iii)* measures the execution time of the started *computation* of the currently running task by clock *c* and synchronize with the task automaton by channel *Finished[RunID]* when the *computation* is finished (transitions related to locations *Wait* and *CompFinished*).

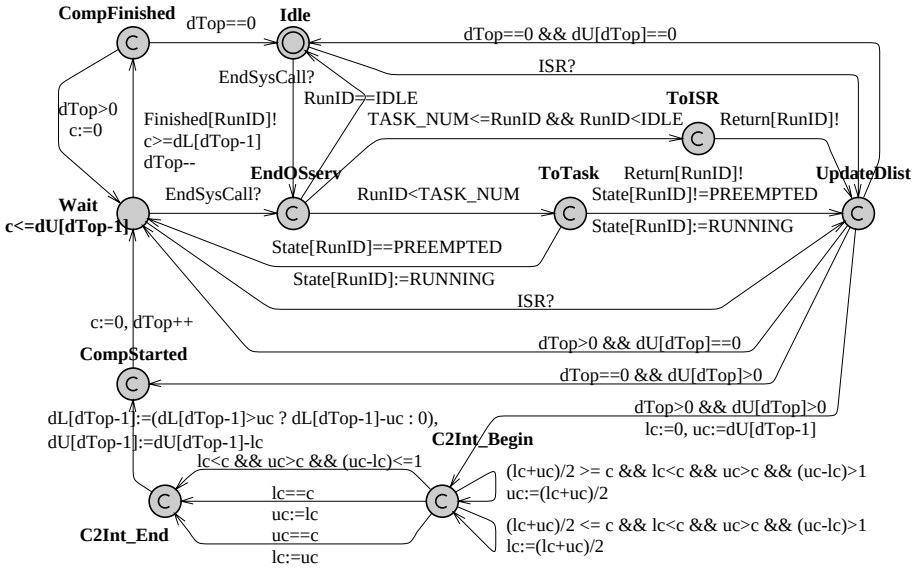


Fig. 5.3 *ExeCtrl* automaton

6 Over-approximation of preemption modeling

It has been explained in section 5 that the already executed time of the preempted computation (measured by clock c) is approximated by an interval bounded by the nearest lower and the nearest upper integers *lc* and *uc* respectively (see Fig. 6.1). The preemption of the currently executed *computation* can be caused only by an interrupt. The over-approximation therefore increases the size of interval of possible execution times ⟨BCET, WCET⟩ at most by one for each interrupt that occurs when the task model is in the location corresponding to an *computation* (when the computation is preempted its response time is affected indirectly by over-approximate response time of the preempting task). When the interrupt occurs at an integer value of clock *c* there is no over-approximation (*uc=lc*).

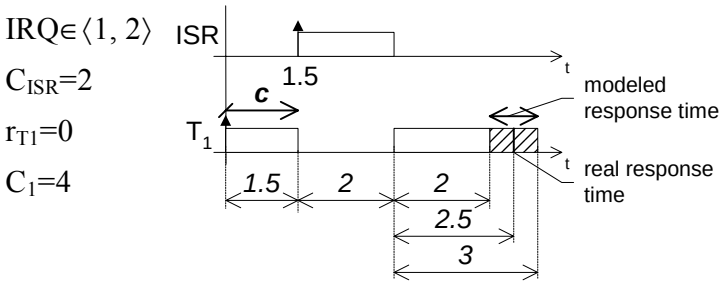


Fig. 6.1 Example of preemption

The following inequalities hold for each *computation*:

(i.) $BCRT_{one\text{-}clock\ Model} \leq BCRT_{Real\ System}$, and

(ii.) $WCRT_{one\text{-}clock\ Model} \geq WCRT_{Real\ System}$, and

(iii.) $WCRT_{one\text{-}clock\ Model} - BCRT_{one\text{-}clock\ Model} \leq WCRT_{Real\ System} - BCRT_{Real\ System} + NumInt$,

(iv.) $WCRT_{one\text{-}clock\ Model} - BCRT_{one\text{-}clock\ Model} = WCRT_{Real\ System} - BCRT_{Real\ System} + NumInt\_NonIntegerClock$,

where NumInt is the number of interrupts that occur when the task model is in location corresponding to *computation* and NumInt_NonIntegerClock is the number of interrupts that occur in a non-integer value of clock *c*, when the task model is in location corresponding to *computation*.

Due to an additional non-determinism in the model, the set of system real behaviors is a subset of the set of modeled behaviors (the model is an over-approximation of the real system). It is important to keep this fact in mind during a verification since the over-approximation does not preserve satisfaction of all general properties. It is important from the practical point of view, that the over-approximation preserves safety and bounded liveness properties [20]. 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. On the other hand, the over-approximation brings some pessimism to the mode. For an example, the property "Response time of task T_2 is always smaller or equal to 6" is satisfied in the real system but it is not satisfied in the model (see Fig. 6.1).

7 Experimental results

In this section we compare the complexity of our *one-clock* approach with *n-clock* approach. The *n-clock* approach is very similar to *one-clock* approach therefore it is not described here in details. It differs only in the following details. One clock for each task automaton is used. Each *computation* is represented by one location of the same name, as in *one-clock* approach, but the time spent in this location is measured by clock c_i dedicated to task i, it represents the *computation* response time (including preemption) and it is bounded by values stored in integers *L[ID]* and *U[ID]*. These bounds are initialized to BCET and WCET, respectively and they are increased when the task is preempted (provided by other timed automaton called *PreemptCtrl*). *PreemptCtrl* prolongs bounds of the *computation* response time (*L[ID]* and *U[ID]*) of the preempted task by duration of the preemption. Since the right duration of the preemption can not be measured in timed automata, bounds *L[ID]* and *U[ID]* are prolonged by duration of possible preemption, that are bounds *L[RunID]* and *U[RunID]* of the preempting task. *PreemptCtrl* moreover provides *context-switch* modeling as *ExeCtrl* automaton in *one-clock* approach do. Since *Δ-list* is not used, other functionality of automaton *ExeCtrl* are not necessary. Other parts of *n-clock* model are the same as in *one-clock* approach.

The complexity of timed automata model checking is exponential on the number of clocks and on the largest time constant appearing in timing constraints [12]. Therefore *one-clock* approach promises significant improve in the complexity of this model verification. The effect of number of clocks reduction is slightly counter balanced by the complexity of algorithm *Clock2Int* used to find the nearest lower and upper integer of the clock (see Fig. 5.2). Since the complexity of *Clock2Int* algorithm is only $O(\log_2 MaxC)$ the reduction of number of clocks provides significant improve of the complexity.

We compare *n-clock* approach and *one-clock* approach on the following example. Tasks codes are listed in Fig. 7.1. There is only one instance of TaskA (T0) activated by several instances of TaskB (T1, T2, ...) that are periodically activated by an interrupt service routine (ISR). Instance of TaskA has the highest priority, instances of TaskB has priority assigned rate-monotonically. ISR is periodically invoked with period 25 time units. It provides a computation taking from 0 to 5 time units and activates instances of TaskB whose period expired.

Three cases (case1, case2 or case3) are distinguished according to the number of executed tasks instances (4, 6 or 8). In the case1, there are executed three instances T1, T2 and T3 (besides ISR and T0) activated with periods 600, 900, and 1800. In the case2, there are executed five instances form T1 to T5 activated with periods 360, 450, 600, 900, and 1800. In the case3, there are executed seven instances form T1 to T7 activated with periods 225, 300, 360, 450, 600, 900, and 1800.

```
TaskA   // Only one instance T0
{
  Comp;                 // ⟨0, 5⟩
  TerminateTask();
}
```

```
TaskB   // Instances T1, T2, ....
{
  Comp1;                // ⟨0, 5⟩
  ActivateTask (T0);
  Comp2;                // ⟨0, 10⟩
  TerminateTask();
}
```

Fig. 7.1 Tasks code

Except mentioned tasks instances and ISR, also a timer generating periodically an interrupt has been modeled (it contains one clock variable) and one additional clock has been used for a verification of a bounded response time property (see property P3).

The following properties have been verified:

- P1: System is deadlock free
- P2: Computation *Comp1* is executed mutual-exclusively by all task instances T1, T2,....
- P3: The worst-case response time (WCRT) of task instance T3 is 76.

Since all tasks instances can be activated only once in the model, property P1 express the property that all tasks instances are finished before its next activation, i.e. the system is schedulable. The mutual-exclusivity of execution *Comp1* (property P2) means that *Comp1* of any instance of TaskB cannot be preempted by another instance of TaskB executing also *Comp1* in the same time. The response time of a task instance is assumed to be the time from its activation to its completion (including a preemption). WCRT of T3 explored by *n-clock* approach is 75. However, WCRT of T3 explored by *one-clock* approach is 76 due to over-approximation.

Since property P2 is the safety property and property P3 is the bounded liveness property, they are both preserved by the model over-approximation and it can be therefore concluded that they are satisfied also in the real system. Although deadlock freeness is not preserved by general over-approximation, the property that all tasks are never activated more than once is preserved since it is the safety property.

These properties are formalized in UPPAAL requirement specification language as follows:

- P1: A[] not deadlock
- P2: A[] (T1.Comp1 imply (not (T0.Comp1 or T2.Comp1 or T3.Comp1)))
  A[] (T2.Comp1 imply (not (T0.Comp1 or T1.Comp1 or T3.Comp1)))
  A[] (T3.Comp1 imply (not (T0.Comp1 or T1.Comp1 or T2.Comp1)))
- P3: (ActivateTaskSysCall.End and ParTask==3) --> (T3.End and rt<76)

In UPPAAL requirement specification language syntax A[] f represents computation tree logic (CTL) formula $\forall\Box f$ (i.e. "invariantly holds f"). Syntax p --> q denotes 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 of task instance T3 is reset when T3 is activated in ISR timed automaton. Property P2 is formalized by several formulas - one for each instance of TaskB. There are given three formulas for Case1.

All properties have been successfully verified by UPPAAL 3.4.6 running on PC AMD Athlon 1GHz, with 1.3GB RAM. Memory requirements and time necessary for verification of properties of three models by *one-clock* approach and *n-clock* approach are listed in Table 7.1. The measurement has been done by Windows 2000 task manager.

	Property	*one-clock*		*n-clock*	
		Time [min:sec]	Memory [MB]	Time [min:sec]	Memory [MB]
Case 1 (4 tasks)	P1	0:1	8.6	0:1	7.2
	P2	0:1	8.3	0:0[1]	7.1
	P3	0:2	20.4	0:0[1]	11.2
Case 2 (6 tasks)	P1	0:4	17.6	0:14	68
	P2	0:6	15.6	0:14	68
	P3	0:4	36.5	0:11	134
Case 3 (8 tasks)	P1	0:9	40.5	7:0	1788
	P2	1:22	36	7:28	1811
	P3	0:8	65	---	Out of mem.[2]

Table 7.1 Time and memory requirements of verification of *one-clock* and *n-clock* approach model

Both approaches use the same data structures to model ready queue, tasks states and priorities etc. While *n-clock* approach model uses arrays *L* and *U* for storing the responce time of each task, *one-clock* approach model uses *Δ-list* consisting of arrays *dL* and *dU* and pointer *dTop*. Therefore data structures used by *one-clock* approach are non-significantly bigger. Moreover managing preemption in *one-clock* approach (computing *lc* and *uc* by algorithm in Fig. 5.2) is usually little bit more demanding than in *n-clock* approach (increasing response times of all preempted tasks). Therefore the results of the comparison of *one-clock* and *n-clock* approach model verification (Table 7.1) show that *one-clock* approach is little bit more expensive than *n-clock* approach in memory and time requirements for small number of tasks (4 tasks). For 6 tasks however, the higher number of clocks used in *n-clock* approach causes that *one-clock* approach wins the comparison of the time and memory requirements. The verification of *one-clock* approach model consisting of 8 tasks is approximately six times faster (for P1 even forty times faster) and requires approximately forty-five times less memory than the verification of corresponding *n-clock* approach model.

8 Conclusion

We have demonstrated in this paper, how timed automata can be used for the multitasking preemptive application modeling.

The main contribution of our work is that the proposed model requires only one clock variable for all tasks in the application. Since the complexity of the timed automata model checking is exponential on the number of clocks [12], reduction of number of used clocks provides the most important reduction of the state space. Results of experiments show that improvement in verification complexity is significant for higher number of tasks (six and more).

Since 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)), reduction of the model state-space size is one of the most important issues that must be solved prior to wide use of formal methods in praxis.

Therefore the proposed model is abstract as much as possible and contains only information necessary for 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 the 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, a non-determinism and the number of tasks and *computations* to obtain the model of reasonable size.

[1] Les than 1 second
[2] More than 2GB

Even though the model is an over-approximation of the real system behavior, complex time and logical properties considering an application data and a controlled system model can be verified by a model-checking tool, since safety and bounded liveness properties (the most important groups) are preserved by an over-approximation.

Opposite to hybrid automata allowing the 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 preemptive tasks can be modeled. Opposite to models used in the 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 the less pessimistic analysis is provided by model-checking approach, especially when the analyzed application contains features that make the response time analysis pessimistic (e.g. tasks self-suspension).

Acknowledgement

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

We would like to thank Gerd Behrmann form the department of computer science at Aalborg University, Denmark for his advices related to reducing complexity of the ready queue model.

References

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

[2] Behrmann, G., David, A., Larsen, K., G., Möller, O., Pettersson, P. and Yi. W. Uppaal - Present and Future. In: Proceedings of the 40th IEEE Conference on Decision and Control (CDC'2001). Orlando, Florida, USA, December 4 to 7, 2001.

[3] Daws, C., Olivero, A., Tripakis, S., Yovine, S. The tool Kronos. In Proceedings of "Hybrid Systems III, Verification and Control", 1996. Lecture Notes in Computer Science 1066, Springer-Verlag.

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

[5] 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)

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

[9] Lundqvist, K., Asplund, L.: A Ravenscar-Compliant Run-time Kernel for Safety-Critical Systems. Real-Time Systems 24(1): 29-54. Kluwer (2003)

[10] 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

[11] Fersman, E., Pettersson, P., Yi, W.: Schedulability Analysis using two clocks. In Proceedings of TACAS'03, volume 2619 of LNCS, pages 224-239. Springer, 2003.

[12] Daws, C., Yovine, S. Reducing the number of clock variables of timed automata. In Proceedings of the 17th IEEE Real Time Systems Symposium, RTSS'96, Washington, DC, USA, December 1996. IEEE Computer Society Press.

[13] OSEK. OSEK/VDX Operating System Specification 2.2.1. http://www.osek-vdx.org/

[14] Waszniowski, L., Hanzálek, Z. Analysis of Real Time Operating System Based Applications. In Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems FORMATS'03. Springer-Verlag, 2003
[15] Burns, A., Wellings, A. J. Real-time systems and their programming languages. Addison-Wesley Publishers Ltd. 1990
[16] Bailey C.M., Burns A., Wellings A.J. and Forsyth C.H.: A Performance Analysis of a Hard Real-Time System. Control Engineering Practice, Vol. 3(4), pp. 447-464, Pergamon (1995).
[17] Krákora, J., Waszniowski, L., Píša P., Hanzálek, Z.: Timed Automata Approach to Real Time Distributed System Verification, 5th IEEE International Workshop on Factory Communication Systems, WFCS, Vienna, September 22-24, 2004.
[18] Fredette, A.N. and R. Cleaveland (1993). RTSL: A Language for Real-Time Schedulability Analysis. In: *Proceedings of the Real-Time Systems Symposium*. pp 274--283, IEEE Computer Society Press.
[19] 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.
[20] 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.
[21] 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.