Model Checking μC/OS-III Multi-task System with TMSVL

Jin Cui[1], Zhenhua Duan[1](✉), Cong Tian[1](✉), Nan Zhang[1], and Conghao Zhou[2]

[1] ICTT and ISN Laboratory, Xidian University, Xi'an 710071, People's Republic of China
{ctian,zhhduan}@mail.xidian.edu.cn

[2] College of Information Science and Engineering, Northeastern University, Shenyang 110819, People's Republic of China

Abstract. μC/OS-III is the third generation of real-time operating systems based on multi-task scheduling for embedded systems. The multi-task system which refers to tasks with the same priority, tasks synchronization and communication, is scheduled by the operating system kernel. It is critical to ensure the timeliness and correctness of related applications using μC/OS-III. This paper proposes a model checking approach to verify a multi-task embedded system running under μC/OS-III. To do so, the multi-task system and its properties are modelled in TMSVL. A model checker built in the toolkit MSV is used to verify the schedulabilty of the μC/OS-III multi-task system. Experiments show that our approach is effective and efficient in verifying embedded systems.

Keywords: Model checking · TMSVL · Multi-task systems · Schedulability · μC/OS-III

to formalize the problem in some formal languages, and determine the schedulability by verifying whether the model possesses the corresponding properties by a supporting tool. The validity of the verification result relies on whether the formalization is consistent with the original problem.

In [9], an abstract formal model to represent AUTOSAR OS programs for determining schedulability properties is proposed where the tasks are periodical and the deadlines and periods of tasks coincide. In [12], schedulability of preemptive event-driven asynchronous real-time systems is analyzed by a conservative approximation method on composable timed automata models. In [1], timed automata is used to find optimal schedules for the classical job-shop problem. In [13], the Uppaal model-checker is applied for schedulability analysis of a system with single CPU, fixed priorities preemptive scheduler, mixture of periodic tasks and tasks with dependencies.

Modeling, Simulation and Verification Language (MSVL) is an executable subset of Projection Temporal Logic (PTL) [7]. TMSVL [8] is a Timed version of MSVL, which is designed to model, simulate and verify real-time systems. A toolkit MSV has been developed to support the above three missions. In particular, a unified model checker can be used to verify whether or not a real-time system satisfies a specified property. An advantage of TMSVL model checking over other model checking approaches is that the model of the system and the property to be verified are both defined in TMSVL. Further, the verification process can be automatically performed with MSV.

In this paper, we verify schedulability of μC/OS-III based applications. The multi-task system consists of independent tasks, synchronous tasks, and tasks with the same priority, which are scheduled by the OS kernel. First, we model the OS scheduler and different kinds of tasks with TMSVL. Then the schedulability of the systems is formalized and the schedulability of tasks is verified with MSV.

The paper is organized as follows. The next section introduces the preliminaries of TMSVL. In particular, how timeout, delay, and timeout after time delay constraints are formalized in TMSVL is introduced. Section 3 gives an overview of μC/OS-III and Sect. 4 discusses the model checking process of a μC/OS-III multi-task application. Finally, conclusion and future work are drawn in Sect. 5.

2 TMSVL

MSVL is a temporal logic programming language consists of conjunction, selection, sequence, parallel, branching, loop as well as projection statements. TMSVL is a real-time extension of MSVL where quantitative temporal constraints are employed to limit the time duration bounded on statements or programs. Real variables T and Ts are used to describe time and time increment, respectively.

2.1 Statements in TMSVL

TMSVL consists of arithmetic expressions, boolean expressions, and basic statements. The arithmetic expression e and boolean expression b are defined by the following grammar:

$$\begin{aligned} e &::= n \mid x \mid \bigcirc x \mid \ominus x \mid e_0 \, op \, e_1 (op ::= + \mid - \mid * \mid / \mid mod) \\ b &::= true \mid false \mid e_0 = e_1 \mid e_0 < e_1 \mid \neg b \mid b_0 \wedge b_1 \end{aligned}$$

where n is a constant, x is a variable; $\bigcirc x$ and $\ominus x$ denote the value of x at the next and previous state over an interval, respectively.

1. MSVL statement	p
2. Time constraint statment	$(t_1, t_2)tp$
3. Conjunction statement	$tp_1 \wedge tp_2$
4. Selection statement	$tp_1 \vee tp_2$
5. Sequential statement	tp_1 **;** tp_2
6. Parallel statement	$tp \parallel tq$
7. Conditional statement	`if` b `then` $\{tp\}$ `else` $\{tq\}$
8. While statement	`while` (b) $\{\ tp\ \}$
9. Projection statement	$(tp_1, \ldots, tp_m)$ prj (tp)

Fig. 1. Basic TMSVL statements

Elementary statements of TMSVL are defined in Fig. 1. First, MSVL statements are included. Suppose t_1 and t_2 are arithmetic expressions and tp a TMSVL statement, the time constraint statement $(t_1, t_2)tp$ means that tp is executed over the time duration from t_1 to t_2. Two possible interpretations of formula $(t_1, t_2)tp$ are shown in Fig. 2. The black dots are states and are represented by $s_0, s_1, \ldots, s_k, \ldots, s_{l_2}$, respectively. We specify s_k as the current state here, thus $s_0, \ldots, s_{k-1}$ are the previous states and $s_{k+1}, \ldots, s_{l_2}$ the future ones. s_{l_2} is the terminal state. An interval is a sequence of states, for example $s_0, s_1, \ldots, s_{l_2}$ constitute an interval. Figure 2(a) shows the case $t_1 > T$ and Fig. 2(b) the case $t_1 = T$. The formula tp in $(t_1, t_2)tp$ must terminate just when $T = t_2$, otherwise $(t_1, t_2)tp$ is $false$. $tp_1 \wedge tp_2$ means that tp_1 and tp_2 are executed concurrently, and terminate at the same time. Selection statement $tp_1 \vee tp_2$ means tp_1 or tp_2 is executed. $tp_1; tp_2$ means that tp_2 is executed after tp_1 finishes. Parallel statement $tp \parallel tq$ means that tp and tq are executed in parallel, while they are not required to terminate at the same time. Conditional and while constructs are consistent with that in general programming languages such as C and $Java$. Projection statement $(tp_1, \ldots, tp_m)$ prj tp means that tp is executed in parallel with $tp_1; tp_2; \ldots; tp_m$ over an interval obtained by taking the endpoints of the intervals over which $tp_1, \ldots, tp_m$ are executed. An endpoint denotes the first or the last state of an interval. Taken (tp_1, tp_2, tp_3) prj tp as an example. We assume tp_3 terminates before tp. The semantics of (tp_1, tp_2, tp_3) prj tp is intuitively depicted in Fig. 3.

2.2 Normal Form and Normal Form Graph for TMSVL

Execution of TMSVL programs depends on the transformation of TMSVL programs into normal forms. A TMSVL program p is in its normal form if p is

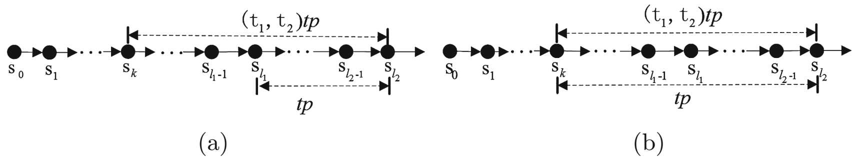


Fig. 2. Semantics of time constraint statement

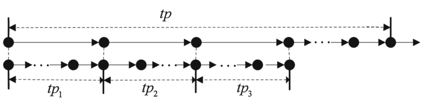


Fig. 3. An example of projection structure

written as:

$$p \equiv \bigvee_{i=1}^{l_1} p_{ei} \wedge \varepsilon \vee \bigvee_{j=1}^{l_2} p_{cj} \wedge \bigcirc p_{fj}$$

where l_1, l_2, i, and $j \in N_0$, $l_1 + l_2 \geq 1$, and p_{fj} is a TMSVL program; p_{ei} and p_{cj} are formulas of the form: $x_1 = e_1 \wedge \ldots \wedge x_m = e_m$. ε means the termination of a program. That is there does not exist a next state. $\bigcirc p_{fj}$ means that p_{fj} will be executed at the next state. It has been proved that any TMSVL program can be transformed into normal form.

Given a TMSVL program p, we can construct a graph named Normal Form Graph (NFG) [6,14,16] that explicitly illustrates the state space of the program. An NFG is a directed graph, denoted as $G =< V, A >$, with a node in the set V of nodes representing a program in TMSVL and an arc in the set A of arcs representing a state. In fact, NFG determines the models that satisfy the corresponding TMSVL program.

Suppose that the sets V and A are empty initially, NFG $G =< V, A >$ of a TMSVL program p can be constructed by determining the set of nodes V and the set of arcs A inductively as follows:

1. $V = V \cup \{p\}$;
2. for any node $q \in V \backslash \{\varepsilon, false\}$, if $q \equiv \bigvee_{i=1}^{l_1} q_{ei} \wedge \varepsilon \vee \bigvee_{j=1}^{l_2} q_{cj} \wedge \bigcirc q_{fj}$, then $V = V \cup \{\varepsilon,\ q_{fj}\}$ and $A = A \cup \{(q, q_{ei}, \varepsilon),\ (q, q_{cj}, q_{fj})\}$ for each i and j with $1 \leq i \leq l_1$ and $1 \leq j \leq l_2$.

An element in the set of arcs A is a triple. For instance, (q, q_{ei}, ε) denotes a directed arc from nodes q to ε with the arc labeled with q_{ei}.

2.3 Timeout in TMSVL

It is necessary to confine the time for waiting for a particular condition to become true such that the waiting is terminated when the time expires. Timeout on waiting is a practical method usually adopted in real-time systems and protocols. A maximum waiting time is given in advance, so the waiting process stops finally in one of the following cases: (1) the events waited occur; (2) the event does not occur but the waiting time expires. The two cases are formalized separatively in *time delay* and *timeout* constraints first. Then the constraint named *timeout after time delay* which combines *time delay* and *timeout* constraints is introduced to express the scenario of timeout on waiting or on other process.

Time delay constraint $\{d_1, d_m\}p$ (d_1 and d_m are non-negative reals and $d_1 \leq d_m$) represents that the statement p starts at the current time and terminates after at least d_1 time units and at most d_m time units. d_1 and d_m provides the upper and lower limits of the time that is taken for p to execute. The statement p is the TMSVL formalism of the waiting process or other real-time process. The constraint for *time delay* is expressed as follows:

$$\{d_1, d_m\}p \overset{\text{def}}{=} (T, T+d_1)p \vee \ldots \vee (T, T+d_i)p \vee \ldots \vee (T, T+d_m)p$$

where $d_i = d_{i-1} + Ts$ and $1 < i \leq m$. It is a disjunction of time constraint statements starting at T and ending at any time within $T + d_1$ and $T + d_m$.

Timeout constraint $(t_1@t_m)p$ means that p starting at $T = t_1$ terminates when $T = t_m$ naturally or forcibly. If p is not finished when $T = t_m$, it is terminated forcibly. Otherwise, p finishes just when $T = t_1$ naturally. Its definition is given as follows:

$$(t_1@t_m)p \overset{\text{def}}{=} (t_1, t_m)\ p_c^1 \wedge (t_2, t_m)p_c^2 \wedge \ldots \wedge (t_m, t_m)(p_e^m \vee p_c^m)$$

where $t_1, \ldots, t_m$ are the time values of m consecutive states respectively. p_c^i represents a state formula obtained by the state reduction on p when $T = t_i$ ($1 \leq i \leq m$). p_e^m represents a terminal state formula indicating that p finishes naturally when $T = t_m$.

Combining the two constraints above, we derive the *timeout after time delay* constraint, denoted as $\{d_1@d_m\}p$. The definition is given as follows:

$$\{d_1@d_m\}p \overset{\text{def}}{=} (T, T+d_1)p \vee \ldots \vee (T, T+d_i)p \vee \ldots \vee (T, T+d_{m-1})p \vee (T@T+d_m)p$$

where $d_i = d_{i-1} + Ts$ and $1 < i \leq m$. In the *timeout after time delay* constraint, when the time delay reaches the upper bound d_m but p still does not finish, p will be terminated forcibly.

3 μC/OS-III Overview

μC/OS-III is different from μC/OS-II mainly in two aspects: (1) task management; (2) OS kernel service.

3.1 Task Management

μC/OS-III supports multitasking and allows the applications to have any number of tasks. The maximum number of tasks available only limited by and depends on the configurations of hardware systems. Tasks of embedded systems typically take the form of an infinite loop.

In order to implement a specific functionality, tasks are usually not completely independent in realistic applications. They need to synchronize and communicate. μC/OS-III uses semaphores, task semaphores, event flags, messages and message queues to synchronize and communicate between tasks. Compared with μCOS-II, task semaphore is a newly introduced synchronous mechanism. It can be directly signaled by a task to another one without creation.

3.2 OS Kernel Services

The kernel is an important part of OS and its primary duty is tasks scheduling. μC/OS-III kernel is preemptive and it uses priority-based scheduling. Tasks priority is specified by users when tasks are created. Different μC/OS-III tasks may have the same priority. For this reason, round robin scheduling [15] is adopted in the kernel scheduler. Each task is assigned a duration of time (namely time quantum) to perform. The task is blocked when the time quantum runs out and the following task which is ready gets the turn to execute. A task finishes or being blocked before the quantum running out also yields the processor to other ready tasks. A list is needed for recording the ready tasks and arranges them in order of the earliest ready time. When a task runs out of quantum, it is moved to the end of the list. μC/OS-III scheduler differs from that of μC/OS-II for it utilizes round robin to priority-based scheduling to deal with tasks with the same priority. Task scheduling is triggered in the following situations: (1) a task is added or deleted, or the priority of a task is changed; (2) a task delays itself, or the delay ends; (3) the event a task requests becomes available.

4 Modeling and Verification of a μC/OS-III Multi-task Application

In this section, an abstract μC/OS-III multi-task application is given. In order to verify schedulability, the μC/OS-III kernel is formalized. Then, the TMSVL formalism of different kinds of tasks including dependent (periodic and non-periodic) tasks and tasks with synchronizations is given. Meanwhile, the property to be verified is expressed in TMSVL. Finally, The toolkit MSV is used to verify the schedulability of the tasks in the application.

4.1 A Multi-task Application

The application consists of five user tasks: task_0, task_1, task_2, task_3, and task_4 with the priorities being 5,6,7,7,8. Larger number represents lower priority.

Thus, task$_0$ has the highest priority, followed by task$_1$, then task$_2$ and task$_3$ which have the same priority, and finally task$_4$. The five tasks are responsible for different functionalities. The relationship of tasks is given in Fig. 4. task$_0$ and task$_2$ are synchronized through task semaphore se_0. task$_0$ sends out se_0 to activate task$_2$. If task$_2$ cannot receive se_0, it waits infinitely. Similarly, task$_1$ and task$_3$ are synchronized through se_1. But task$_3$ waits no more than *to* time units for se_1. task$_4$ is an independent task.

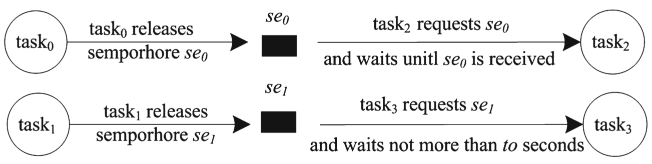

task$_0$
task$_0$ releases semporhore se_0
se_0
task$_2$ requests se_0 and waits unitl se_0 is received
task$_2$
task$_1$
task$_1$ releases semporhore se_1
se_1
task$_3$ requests se_1 and waits not more than *to* seconds
task$_3$

Fig. 4. The relationship between task$_0$ and task$_2$, task$_1$ and task$_3$

In Fig. 5, task$_0$ executes `Computation`$_1$ first. Then it releases the semaphore se_0 after `Computation`$_0$. Finally, task$_0$ delays for t_0 time units. The structure of task$_1$ is the same as task$_0$. task$_2$ and task$_3$ share the same priority. They also have similar structures, so we just give the pseudo-code of task$_2$. It requests se_0 first. `pend(se`$_0$`,` *to*`)` means that the waiting time for se_0 is at most *to* time unites, specially, *to* < 0 means there is no time limit on waiting se_0. The first argument se_0 is an integer variable representing the semaphore being requested and the second argument is the time limit for waiting for the signal. For task$_2$, since *to* < 0, it executes `Computation`$_2$ only after receiving the signal se_0. When the execution of `Computation`$_2$ is finished, it goes on the requesting for se_0 for the next execution. task$_4$ performs computation and delays for t_4 time units when the computation finishes.

```
task$_0$()
{ while(1)
  { Computation$_0$;
    post(se$_0$);
    delay(t$_0$);
  }
}

task$_2$()
{ while(1)
  { pend(se$_0$, $to$);
    Computation$_2$;
  }
}

task$_4$()
{ while(1)
  { Computation$_4$;
    delay(t$_4$);
  }
}
```

Fig. 5. Tasks pseudo-code

4.2 TMSVL Model of OS Kernel

The OS kernel model consists of the variables which represent the OS objects (e.g. the ready tasks, the highest priority ready task) and the TMSVL model of the OS scheduler.

Kernel Variables. We use an array *rd* to represent whether each task in it is ready. The index of *rd* is the task number and smaller index corresponds to higher priority. For index i, if task i is ready, $rd[i] = 1$; otherwise, $rd[i] = 0$. *rd* is initialized to zero. The variable *runTaskID* stores the task number of the currently running task. The float variable *Quan* stores the time quantum for scheduling the same priority tasks in round-robin manner.

A *List* variable l is used to store the ready tasks for each priority when round robin scheduling is enabled. The definition of *List* is as follows:

```
struct{ int taskID; List *nextEL; } List l;
```

The first member in *List* stores the identifier of a task, and the second member is a *List* pointer pointing to the next *List* element. When a task is ready, it is added to the end of l.

Kernel Services. In the TMSVL model of the kernel service, we use Q and *M_Robin* to represent the OS scheduler and the round robin scheduling modules. Q is given in Fig. 6 and it finds the ready task with the highest priority by conjunctions of the *if* statements and stores the task's number in *runTaskID*. The number of *if* statements is the number of priorities used by tasks. For a priority which corresponds to more than one task, an *if* statement is enough and round robin scheduling *M_Robin* is invoked in that case. In Fig. 6, task$_2$ and task$_3$ have the same priority.

M_Robin is given in Fig. 7. We use several functions to express the operation on the *List* l. In Line 1, `size(l)` returns the number of elements in l.

```
Q ≝
 1.  while(true)
 2.  {  if(rd[0]=1) then{runTaskID=0}
 3.     and
 4.     if(rd[0]=0 and rd[1]=1)
 5.     then{runTaskID=1}
 6.     and
 7.     if(rd[0]=0 and rd[1]=0 and (rd[2]=1 or rd[3]=1))
 8.     then{M_Robin }
 9.     and
10.     if( rd[0]=0 and rd[1]=0 and rd[2]=0 and rd[3]=0 and rd[4]=1)
11.     then{runTaskID=4 }
12.     and
13.     ...
14.     if( rd[0]=0 and rd[1]=0 and ...)
15.     then{runTaskID=IDEL }
16.     and skip
17. }
```

Fig. 6. TMSVL model of the scheduler

```
M_Robin ≝
     1.  if(size(l)>0)
     2.  then{ runTaskID=head(l) and
     3.      if( ac[head(l)]+Ts<C[head(l)] and (ac[head(l)]+Ts)%Quan!=0 )
     4.      then{runTaskID:=head(l)} }
     5.      and
     6.      if(ac[head(l)]+Ts=C[head(l)])
     7.      then{next popHead(l) and
     8.          if(size(l)>1) then{runTaskID:=head(l)} }
     9.      and
     10.     if(ac[head(l])]+Ts<C[head(l)] and (ac[head(l)]+Ts)%Quan=0)
     11.     then{ next MoveHead2Tail(l) and
     12.         runTaskID:=head(l) }
     13.     }
```

Fig. 7. TMSVL model of the round robin scheduling

If `size(l)`>0, l is not empty, the statements in Lines 2–13 are executed. The head of l is running first (Line 2). The function `head()` is used to obtain the *taskID* of the first element in l. It takes only one *List* type argument and returns an integer representing the first element's *taskID* of the *List* l. Lines 3–4 show the case where neither does the first element of l run out of the quantum nor does it finish at the next state and the head element goes on running at the next state. Lines 6–8 show the second case where the task corresponding to `head(l)` finishes. In this case, the task is removed from l. Here we use the function `popHead(l)` to represent this operation. Next, we need to test whether l is empty after `popHead(l)` and if l is not empty, the task corresponding to the head of l gets the processor by setting *runTaskID* to the value of `head(l)` at the next state. Lines 10–12 show the case where the task is not finished but runs out of the time quantum at the next state. In this case, the task is moved from the head to the tail of l and the task corresponding to the new head gets the chance to run at the next state. The function `MoveHeadToTail()` moves the head of l to the tail and makes the head of l change (Line 10).

4.3 TMSVL Model of a Multi-task System

The multi-task system consists of five parallel tasks. Let $\texttt{M\_task}_i$ represent the user task$_i$ (i=0,1,2,3,4). We denote the model of the multi-task system as `M`. Thus $\texttt{M} \equiv ||_{i=0}^{4}\texttt{M\_task}_i$.

We use float array elements $C[i]$, $ac[i]$ and $acD[i]$ to represent the required computation time, the accumulated running time and the accumulated delay time of $task_i$ in the current period, respectively. Boolean array elements $wait[i]$ and $ex[i]$ are used to indicate whether task$_i$ is at the waiting and the executing state, respectively.

Figure 8 shows the TMSVL model of task$_0$, task$_1$ and task$_4$. A new computation circle starts in Line 2. Then the task waits its turn to run (Line 3).

```
M_task$_i$ $\overset{\text{def}}{=}$ //i=0,1,4
    1.  while(true)
    2.  { ac[i]=0 and
    3.      await(runTaskID=i);
    4.      while(ac[i]<C[i])
    5.      { if(runTaskID=i)
    6.      then{ac[i]:=ac[i]+Ts and ex[i]=1 and wait[i]=0 and
    7.              if(ac[i]+Ts=C[i] and $i! = 4$) then{$se_i$:=$se_i$+1} }
    8.      else {ex[i]=0 and skip} };
    9.      (T,T+dly[i])keep( next acD[i]=acD[i]+Ts and
    10.                 rd[i]=0 and ex[i]=0 and wait[i]=1);
    11.             acD[i]=0 and rd[i]=1 and empty
    12. }
```

Fig. 8. TMSVL model of tasks 0, 1, 4

```
M_task$_i$ $\overset{\text{def}}{=}$ //i=2,3
    1.  while(true)
    2.  { if($se_{i-2} \leq 0$)
    3.      then{rd[i]=0 and ex[i]=0 and wait[i]=1 and
    4.              {0@$to$}await($se_{i-2} > 0$);
    5.              rd[i]=1 and wait[i]=0 and empty };
    6.      ac[i]=0 and
    7.      await(runTaskID=i);
    8.      while(ac[i]<C[i])
    9.      { if(runTaskID=i)
    10.       then{ac[i]:=ac[i]+Ts and ex[i]=1 and wait[i]=0 and
    11.             if(ac[i]+Ts=C[i] and $se_{i-2} > 0$)
    12.             then{$se_{i-2}$:=$se_{i-2}$-1 and
    13.                 if($se_{i-2} - 1 > 0$)
    14.                then {rd[i]:=1 } } }
    15.     else {ex[i]=0 and skip} }
    16. }
```

Fig. 9. TMSVL model of tasks signaled by other tasks

Lines 4–8 shows the task starts running in a new circle, during this period, it can be preempted by tasks with higher priorities. Lines 5–7 corresponds to the case where the task is running and Line 8 stands for the situation where the task is preempted. task$_0$ and task$_1$ signal to other tasks and delay themselves upon finishing the computations. task$_4$ just delays after finishing its computation. In Line 7, se_i is increased by 1 at the next state when task$_i$ finishes at the next state, the value of i is 0 or 1. When task$_i$ finishes a computation, namely, `ac[i]=C[i]`, it delays for `dly[i]` time units. This is represented by the time constraint statement in Lines 9–10. During this period, `rd[i]` is 0. When the delay ends, task$_i$ becomes ready by setting `rd[i]` to 1.

The models for task$_2$ and task$_3$ are given in Fig. 9. Before a new computation starts, first, the task needs to test whether the requested task semaphore has been sent out (Line 2). $se_{i-2} \leq 0$ (i=2,3) means that the task semaphore has not been sent out and the task has to wait and be at the waiting state (Lines 3–4). $\{$`0@to`$\}$`await(`$se_{i-2} > 0$`)` means the waiting on se$_{i-2} > 0$ is no more than *to* time units. When the semaphore is received or not received in *to* time units, the task stops waiting and turns to the ready state by setting $rd[i]$ to 1 and wait[i] to 0 (Line 5). Then the task waits its turn to run (Line 6–7). Lines 8–15 shows the task starts running in a new circle, during this period, it can be preempted by tasks with higher priorities. Lines 10–14 corresponds to the case where the task is running and Line 15 stands for the situation where the task is preempted. When $ac[i] = C[i]$, namely, task$_i$ finishes the computation of the current period, se_{i-2} is decreased. When the computation of the current circle completes, the program goes to Line 2 to repeat the process above.

4.4 Verification of Schedulability

In the previous section, we model the OS scheduler and different kinds of μC/OS-III tasks (independent tasks, synchronous tasks) with TMSVL. Based on the TMSVL model, the property to be verified is formalized.

Schedulability is an important property for real-time multi-task systems. It means that all the tasks scheduled can finish within the given deadline. In other words, each task can finish in a given time duration from the moment it is ready. Schedulability of N tasks is expressed in TMSVL denoted as $PSch$ below.

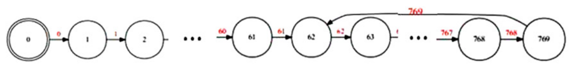
$$PSch \overset{\text{def}}{=} \wedge_{i=0}^{N}(rd[i] = 1 \wedge ac[i] = 0 \rightarrow (\{Ts, D[i]\}true; ac[i] = C[i]))^{+}$$

Here, $D[i]$ is the deadline for task$_i$. $C[i]$ and $ac[i]$ are the computation time and accumulated running time which are given in the previous section. In $PSch$, '+' is derived from the sequential operator '**;** '. Suppose p is a TMSVL statement, p^{+} means that the number of p in $p; p; \ldots ; p$ can be any positive integer.

With the TMSVL model of a μC/OS-III multi-task application and the property described in TMSVL. Whether the property is valid on the application can be automatically checked by the toolkit MSV. In this section, we verify schedulability for the multi-task application given in the previous subsection.

The deadlines for the 5 tasks are stored in the array D where $D[5]$ =$\{$0.03, 0.04, 0.13, 0.23, 0.25$\}$. The computation time for each task is stored in the array C where $C[5]$ =$\{$0.03, 0.04, 0.06, 0.09, 0.12$\}$. The delay time of the five tasks are stored in the array dly where $dly[5] = \{0.3, 0.2, 0, 0, 0.3\}$. The waiting time *to* on request for se_1 is set to 0.03. we assume the tasks are started at the same time $T = 0$.

The verification result for the application is shown in Fig. 10. There are 770 nodes and 770 arcs on the counterexample. Each node represents a program while each arc represents a state which shows the executing of the application at different time. The root node is a double circle, it represents the TMSVL model of the application. Other node represents the future part produced by



```
Arc 0:   T=0.000000 and Ts=0.010000 and ac=[0,0,0,0,0]      and  rd=[1,1,0,0,1] and runTaskID=0 and se0=0 and se1=0
Arc 1:   T=0.010000 and Ts=0.010000 and ac=[0.01,0,0,0,0] and  rd=[1,1,0,0,1] and runTaskID=0 and se0=0 and se1=0
Arc 2:   T=0.020000 and Ts=0.010000 and ac=[0.02,0,0,0,0] and  rd=[1,1,0,0,1] and runTaskID=0 and se0=0 and se1=0
Arc 3:   T=0.030000 and Ts=0.010000 and ac=[0.03,0,0,0,0] and  rd=[0,1,1,0,1] and runTaskID=1 and se0=1 and se1=0
  ⋮                                   ⋮
Arc 7:  T=0.150000 and Ts=0.020000 and ac=[0.03,0.04,0.04,0.04,0] and rd=[0,0,1,1,1] and runTaskID=2 and se0=1 and se1=1
Arc 8:  T=0.170000 and Ts=0.040000 and ac=[0.03,0.04,0.06,0.04,0] and rd=[0,0,0,1,1] and runTaskID=3 and se0=0 and se1=1
  ⋮
                                       ⋮
Arc 769: T=15.100077 and Ts=0.010000 and ac=[0.03,0.04,0.06,0.08,0.12] and rd=[0,0,0,1,1] and
       runTaskID=3 and se0=0 and se1=0
```

Fig. 10. Verification result

executing the program that the precursor node represents, and the current part is represented by an arc. For example, arc 0 and node 1 are the executing results of node 0. The arcs are state formulas while the nodes are TMSVL programs which are required to be further executed.

In Fig. 10, we can see that arc 0 represents the state that $T = 0$, task$_0$, task$_1$ and task$_4$ are ready since the first, second and fifth elements in rd is 1, and task$_0$ starts executing since $runTaskID = 0$. After 0.03 s, task$_0$ finishes and activates task$_2$, that is, task$_2$ is ready at $T = 0.03$ for the first time. When $T = 0.17$, task$_2$ finishes, for the accumulated time $ac[2]$ is equal to $C[2]$. We can see that task$_2$ finishes after 0.14 s from the time it is activated which is greater than the give deadline. So the schedulability for task$_2$ is violated which leads to the violation of $PSch$.

Fig. 11. Verification result

The property $PSch$ is too strict for it requires all the tasks should finish in the given deadlines. We can relax the schedulability requirements by ignoring the deadline for task$_2$ and task$_3$. Thus the property can be represented as follows:

$$PSch_{0,1,4} \overset{\text{def}}{=} \wedge_{i=0,1,4}(rd[i] = 1 \wedge ac[i] = 0 \rightarrow (\{Ts, D[i]\}true; ac[i] = C[i]))^{+}$$

$PSch_{0,1,4}$ just requires that task$_0$, task$_1$ and task$_4$ always finish in the given deadline. The verification result is shown in Fig. 11, there is no counterexample, so we can conclude that task$_0$, task$_1$ and task$_4$ are always finished in their deadline. When the schedulability of a set of tasks is violated, we need to determine which tasks violated the property. In this case, verifying schedulability of a single task at one time instead of the whole is efficient.

5 Conclusion

We present a unified model checking approach to verify schedulability of multi-task application running under μC/OS-III. The OS scheduler which combines priority based scheduling and round-robin scheduling is modeled in TMSVL. Tasks synchronization with timeout and delay mechanism are also formalized in TMSVL. With the toolkit MSV, a multi-task system running under μC/OS-III is formalized and verified. The mechanism that time intervals are adjustable for modeling improves the efficiency of verification. In the near future, we will put TMSVL into practise and verify more realistic industrial applications.

References

1. Abdeddaım, Y., Asarin, E., Maler, O.: Scheduling with timed automata. Theoret. Comput. Sci. **354**(4), 272–300 (2006)
2. Mokadem, H.B., Berard, B., Gourcuff, V., De Smet, O., Roussel, J.-M.: Verification of a timed multitask system with uppaal. IEEE Trans. Autom. Sci. Eng. **7**(4), 921–932 (2010)
3. Bini, E., Buttazzo, G.C.: Schedulability analysis of periodic fixed priority systems. IEEE Trans. Comput. **53**(11), 1462–1473 (2004)
4. Bini, E., Buttazzo, G.C., Buttazzo, G.M.: Rate monotonic analysis: the hyperbolic bound. IEEE Trans. Comput. **52**(7), 933–942 (2003)
5. Bucci, G., Fedeli, A., Sassoli, L., Vicario, E.: Timed state space analysis of real-time preemptive systems. IEEE Trans. Softw. Eng. **30**(2), 97–111 (2004)
6. Duan, Z., Tian, C.: A unified model checking approach with projection temporal logic. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 167–186. Springer, Heidelberg (2008)
7. Duan, Z., Yang, X., Koutny, M.: Framed temporal logic programming. Sci. Comput. Program. **70**(1), 31–61 (2008)
8. Han, M., Duan, Z., Wang, X.: Time constraints with temporal logic programming. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 266–282. Springer, Heidelberg (2012)
9. Huang, Y., Ferreira, J.F., He, G., Qin, S., He, J.: Deadline analysis of AUTOSAR OS periodic tasks in the presence of interrupts. In: Groves, L., Sun, J. (eds.) ICFEM 2013. LNCS, vol. 8144, pp. 165–181. Springer, Heidelberg (2013)
10. Labrosse, J.J.: uC/OS-III: The Real-Time Kernel. Micrium Press, Weston (2009)
11. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM (JACM) **20**(1), 46–61 (1973)
12. Madl, G., Dutt, N., Abdelwahed, S.: A conservative approximation method for the verification of preemptive scheduling using timed automata. In: 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2009, pp. 255–264 (2009)
13. Miku00ionis, M., Larsen, K.G., Rasmussen, J.I., Nielsen, B., Skou, A., Palm, S.U., Pedersen, J.S., Hougaard, P.: Schedulability analysis using uppaal: Herschel-planck case study. In: Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part II (2010)

14. Pang, T., Duan, Z., Tian, C.: Symbolic model checking for propositional projection temporal logic. In: 2012 Sixth International Symposium on Theoretical Aspects of Software Engineering (TASE), pp. 9–16. IEEE (2012)
15. Rasmus, R.V., Trick, M.A.: Round robin scheduling-a survey. Eur. J. Oper. Res. **188**(3), 617–636 (2008)
16. Tian, C., Duan, Z.: Propositional Projection Temporal Logic, B*ü*chi Automata and ω-Regular Expressions. In: Agrawal, M., Du, D.-Z., Duan, Z., Li, A. (eds.) TAMC 2008. LNCS, vol. 4978, pp. 47–58. Springer, Heidelberg (2008)
17. Wasziwoski, L., Hanzalek, Z.: Model checking of multitasking real-time applications based on the timed automata model using one clock. Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation: Applications for Design and Implementation, p. 194 (2009)
18. Waszniowski, L., Krákora, J., Hanzálek, Z.: Case study on distributed and fault tolerant system modeling based on timed automata. J. Syst. Softw. **82**(10), 1678–1694 (2009)