Model Checking Rate-Monotonic Scheduler with TMSVL

Jin Cui
ICTT and ISN Lab
Xidian University
Xi'an, 710071, P.R.China
Email: cuijin_xd@126.com

Zhenhua Duan*
ICTT and ISN Lab
Xidian University
Xi'an, 710071, P.R.China
Email: zhhduan@mail.xidian.edu.cn

Cong Tian
ICTT and ISN Lab
Xidian University
Xi'an, 710071, P.R.China
Email: ctian@mail.xidian.edu.cn

***Abstract*—This paper presents a model checking-based schedulability checking approach for Rate-Monotonic Scheduling (RMS) algorithm. To do so, RMS algorithm is modelled with TMSVL, and the desired property, i.e. schedulability, is specified with the property specification language in TMSVL. Next, whether RMS algorithm is schedulable on a set of tasks is verified by checking whether the desired property is valid on the TMSVL model. A significant advantage of TMSVL is the mechanism of adjustable time intervals which makes an effective reduction on the state space.**

***Keywords*-model checking; Rate-Monotonic Scheduler; TMSVL; real-time systems**

I. Introduction

RMS is a classical scheduling algorithm for periodic tasks. It was proposed by Liu and Layland in 1973 and shown to be optimum among all fixed priority scheduling algorithms [1]. Let τ be a set containing n periodic tasks with P_i and C_i denoting the period and execution time of task $\tau_i \in \tau$, here $1 \leq i \leq n$. A schedulability condition for RMS has been derived under the assumptions that (1) all tasks start simultaneously, (2) deadlines of tasks are equal to their periods, and (3) tasks are independent, i.e. an arrival of a task does not depend on other tasks in τ. Under these assumptions, Liu and Layland proved that the tasks in τ are schedulable with RMS if

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leqslant n(2^{1/n} - 1);$$

that is, the utilization factor U is not more than $n(2^{1/n} - 1)$. However, this is a sufficient but not necessary condition. There may exist a set of tasks where $U > n(2^{1/n}-1)$ but the tasks are still schedulable with RMS. Thus, lots of researches have been carried out to improve the schedulability bound of RMS.

Bini et al. [2] derived a tighter sufficient condition: RMS is feasible for a set of n periodic tasks if $\prod_{i=1}^{n}(\frac{C_i}{T_i}+1) \leq 2$. Exact tests algorithms [3] based on response time analysis can provide sufficient and necessary schedulability conditions for RMS. However, they are too complex to be executed on large task sets. Further, the analysis process is complicated and slight modifications of the scheduling policies will make exact tests algorithms unavailable.

TMSVL[4] is a parallel programming language useful in modeling, simulation, and verification of real-time systems. With systems modeled in TMSVL and properties specified with property specification language in TMSVL, a unified model checking approach can be applied to verify whether the property is valid on the model.

In this paper, we verify schedulability of RMS algorithm on a set of tasks with a unified model checking approach of TMSVL. To do so, RMS algorithm is modelled with TMSVL, and the desired property, i.e. schedulability, is specified with the property specification language in TMSVL. Next, whether RMS algorithm is schedulable on a set of tasks is verified by checking whether the desired property is valid on the TMSVL model. A significant advantage of TMSVL is the mechanism of adjustable time intervals which makes an effective reduction on the state space. The method is straightforward and can be extended to solve a series of relevant problems. For instance, when scheduling policies change, the schedulability still can be verified by making adjustment on the model of scheduler.

The paper is organized as follows. The next section introduces TMSVL language. Section 3 presents the unified model checking approach with TMSVL. In Section 4, how RMS algorithm is modeled and verified with TMSVL is discussed in details. Section 5 presents the related work and Section 6 concludes the paper.

II. TMSVL

TMSVL is a real-time extension of MSVL [5] by making time explicit. In TMSVL, variables T and Ts are used to describe time and time increment, respectively.

Let $Prop$ be a countable set of atomic propositions, N_0 the set of non-negative integers, and V the set of variables. A state s is a pair of assignments (I_{var}, I_{prop}), where for each variable $v \in V$ defines $s[v] = I_{var}[v]$, and for each $\pi \in Prop$ defines $s[\pi] = I_{prop}[\pi]$. $I_{var}[v]$ is a value in the data domain D or nil (undefined), and $I_{prop}[\pi]$ is $true$ or $false$. An interval $\sigma = (s_0, s_1, \ldots)$ is a non-empty (even infinite) sequence of states with its length denoted by $|\sigma|$. Generally, a model of a TMSVL program is an interval.

TMSVL consists of arithmetic expressions, boolean expressions, and basic statements. The arithmetic expression e and boolean expression b are defined as follows.

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

where n is a constant, x is a variable; $\bigcirc x$ and $\ominus x$ denote x concerning the next state and previous state over an interval, respectively. Elementary statements of TMSVL are defined as follows.

978-1-4799-5482-7/14 $31.00 © 2014 IEEE
DOI 10.1109/ICECCS.2014.37

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. Projection statement	$\{tp_1, \ldots, tp_m\}$ prj $\{tp\}$
8. Conditional statement	`if` b `then` tp `else` tq
9. While statement	`while` (b) { tp }

MSVL statements are included in TMSVL since TMSVL is an extension of MSVL. tp is supposed to be a TMSVL statement, t_1 and t_2 are arithmetic expressions, and their values can be determined at each state. The time constraint statement $(t_1, t_2)tp$ means that tp is executed over the time duration from t_1 to t_2. $tp_1 \wedge tp_2$ means that tp_1 and tp_2 are executed concurrently, and can terminate at the same time. Selection statement $tp_1 \vee tp_2$ means tp_1 or tp_2 are executed. $tp_1; tp_2$ means that tp_2 is executed when 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.

Projection statement $\{tp_1, \ldots, tp_m\}$ prj $\{tp\}$ means tp is executed in parallel with $tp_1; tp_2; \ldots; tp_m$ over an interval obtained by taking 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. If $tp_1, \ldots, tp_m$ are identical, we usually use $\{(tp_1)^m\}$ prj $\{tp\}$ to represent $\{tp_1, \ldots, tp_m\}$ prj $\{tp\}$ for simplicity.

Conditional statement `if` b `then` tp `else` tq means that if condition b is evaluated to $true$, then tp is executed, otherwise tq is executed. Statement `while` (b) { tp } allows statement tp to be executed repeatedly for a number of times as long as condition b can be evaluated to $true$. It terminates in case condition b becomes $false$.

TMSVL programs have the form $clock(e_T, e_{Ts}) \wedge q$, where $clock(e_T, e_{Ts})$ is a clock generator, and q denotes TMSVL statements. $clock(e_T, e_{Ts})$ initializes T and Ts, the time and time increment, with the evaluations of arithmetic expressions e_T and e_{Ts}, and enables T to increase with the increment Ts. Meanwhile, Ts can be set in q.

Execution of TMSVL programs depends on transformation of TMSVL programs into normal forms. A TMSVL program p is in its normal form if p is 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 $l_1, l_2, i, j \in N_0$ and $l_1 + l_2 \geq 1$, 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 programs can be transformed into normal forms.

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

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

1) $V = V \cup \{p\}$;
2) for all nodes $q \in V \backslash \{\varepsilon, false\}$, 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 $E = E \cup \{(q, q_{ei}, \varepsilon),\ (q, q_{cj}, q_{fj})\}$ for each i and j with $1 \leq i \leq l_1$ and $1 \leq j \leq l_2$.

An element of the edges set E is a triple. For instance, (q, q_{ei}, ε) denotes the directed edge from nodes q to ε with the edge labeled with q_{ei}.

III. Model checking TMSVL Programs

Suppose the TMSVL model of a system is p and the property to be verified is ϕ. To check whether or not ϕ is valid on p amounts to deciding whether $p \rightarrow \phi$ is valid. Further, whether $p \rightarrow \phi$ is valid is equivalent to whether $p \wedge \neg\phi$ is unsatisfiable. This can be achieved by constructing NFG of $p \wedge \neg\phi$ and then checking whether no paths in the NFG are acceptable. Otherwise, an acceptable path presents a counterexample in the program that violates the property.

We have developed a prototyping tool named TMSVL for supporting verification of real-time systems with TMSVL. An overview of TMSVL is shown in Fig.1. The input of TMSVL can be a TMSVL program or a TMSVL program with a desired property. There are three modes included in TMSVL: simulation, modeling, and verification. In the simulation mode, a sequence of states is generated by executing the program; in the modeling mode, NFG of the input program is constructed; and in the verification mode, TMSVL will verify whether or not the input property is valid to the program. If it is not valid to the program, a witness (counterexample) is provided.

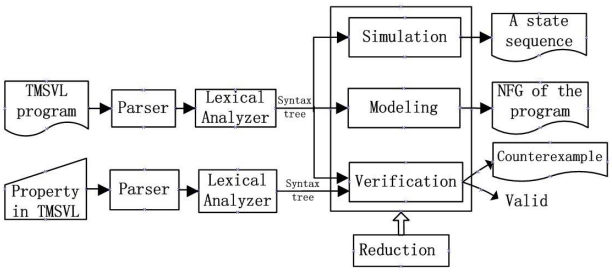

Figure 1. Overview of TMSVL

Reduction is the main technology for implementation of the three modes mentioned above. It includes state reduction and interval reduction. State reduction is to transform the program into its normal form. Interval reduction makes the execution of a TMSVL program stop or move to the next state according to its normal form.

IV. Modeling And Verification of RMS

Within RMS algorithm, priorities of tasks to be scheduled are fixed and tasks with lower priorities can be preempted by those with higher priorities. Further, the priorities of the tasks are inversely proportional to their periods [6]. As a result, tasks scheduled under RMS are assumed to be periodic.

Let τ which consists of n tasks: $\tau_1, \tau_2, \ldots,$ and τ_n be the set of tasks to be scheduled under RMS. Note that n here can be any positive integer. For each task τ_i, its period is P_i and execution time is C_i $(1 \leq i \leq n)$.

A. Modeling RMS with TMSVL

Useful notations are given below.

- r_i: a boolean variable. If the current request for task τ_i is standing, $r_i = 1$; otherwise, $r_i = 0$.
- ex_i: a boolean variable. $ex_i = 1$ represents that task τ_i is being executed, and $ex_i = 0$ is the contrary.
- ac_i: a real variable, with a non-negative value denoting the accumulated running time of task τ_i in the current period.
- run_i: a boolean variable. If task τ_i is being executed, $run_i = i$; otherwise, $run_i = 0$.
- $runTaskNum$: an integer variable, with a non-negative value indicating the running task's subscript. That is, $runTaskNum = i$ if task τ_i is running; otherwise, $runTaskNum = 0$.
- d: a non-negative real variable indicating the time required for finishing the remaining part of a running task.
- d_i: a non-negative real variable representing the time needed for the arrival of the next request of task τ_i .
- ST_i: a non-negative real constant indicating the release time of task τ_i.
- ET_i: a non-negative real constant denoting the end time of the last period of τ_i.
- N_i: a non-negative integer constant representing the number of execution circles for task τ_i.

In the following, RMS is formalized as a TMSVL program defined as RMS_Sch.

$RMS\_Sch \stackrel{\text{def}}{=} clock(e_T, e_{Ts}) \wedge TsSet \wedge ||_{i=1}^{n}(ST_i, ET_i)\tau_i$

$TsSet \stackrel{\text{def}}{=}$

```
1.  keep(
2.  if(runTaskNum! = 0)
3.  then{i = runTaskNum
4.      and if((T − STi)%Pi = 0)
5.          then {d = Ci}
6.          else{d = Ci − aci}
7.      and Ts = min(d, d1, ..., dn) }
8.  else {Ts = min(d1, ..., dn) }
9.  )
```

The module $TsSet$ is a $keep()$ *statement* [5] of MSVL, it means the lines of statements labeled from 2 to 8 which are in $keep()$ are executed at every state expect for the last one over an interval. $TsSet$ sets the time step Ts to eliminate states without events and maintain states concerning schedulability verification. Events here mean the arrivals of tasks' requests or deadlines, as well as the ends of tasks' execution. $min(d, d_1, \ldots, d_n)$ in $TsSet$ returns the minimum in $d, d_1, \ldots,$ and d_n.

Formula $||_{i=1}^{n}(ST_i, ET_i)\tau_i$ means parallel execution of n tasks. It specifies the RMS scheduling policies internally and initiates all the auxiliary variables. τ_i in $||_{i=1}^{n}(ST_i, ET_i)\tau_i$ denotes the TMSVL model of task τ_i as defined below.

$$\tau_i \stackrel{\text{def}}{=} total_i = 0 \wedge ac_i = 0 \wedge \{((T, T + P_i)true)^{N_i}\}\text{prj}\{(T, ET_i)\Box(r_i = 1)\} \wedge keep(Q_i)$$

Projection statement is used here to express periodic requests of each task. To maintain the consistency of time in formula $||_{i=1}^{n}(ST_i, ET_i)\tau_i$, $N_i * P_i = ET_i - ST_i$ is required. The scheduler is described by program Q_i as follows.

$Q_i \stackrel{\text{def}}{=}$

```
1.  if(run_i)
2.  then{ ex_i=1
3.      and if((T − STi)%Pi = 0)
4.      then{◯aci = Ts
5.          and if(Ci > 1)
6               then{◯ri=1}
7.              else{◯ri=0}}
8.      else{◯aci = aci + Ts
9.          and if((T + Ts − STi)%Pi > 0)
10.             then{if(aci + Ts < Ci)
11.                 then{◯ri = 1 }
12.                 else{◯ri = 0 and
13.                     ◯exi = 0}}}}
14. else{ ex_i=0
15.     and if((T − STi)%Pi = 0)
16.     then{◯aci=0
17.         and ◯ri = 1}
18.     else{◯aci = aci
19.         and if((T + Ts − STi)%Pi > 0)
20.             then{◯ri = ri}}}
```

If task τ_i can be processed, ex_i is set as 1. If the current state is a start of a new period, the value of ac_i at the next state is set as the value of Ts at the current state. That is, $\bigcirc ac_i = Ts$. Nevertheless, the value of ac_i at the next state is the sum of ac_i and Ts at the current state, i.e. $\bigcirc ac_i = ac_i + Ts$. It is essential to determine the value of r_i at the next state to decide whether task τ_i will be finished. The value of ex_i is set as 0 if task τ_i has been finished or can't be executed. Meanwhile, if the current state is the start of a new circle of τ_i, r_i is set as 1 and ac_i is set as 0.

B. Verification of schedulability

Following the assumptions in Section I, for a set τ of periodic tasks, schedulability means each task in τ can always be finished before or just at the start of its new circle. The execution of τ will be repeated after $\prod_{j=1}^{n} P_j$ time units at most from $T = min(ST_i)$. So we only need to consider the schedulability in this finite time interval. Particularly, the least common multiple of $P_1, P_2, \ldots, P_n$ can play the same role as $\prod_{j=1}^{n} P_j$.

Based on model RMS_Sch, $ac_i = C_i$ means task τ_i is finished in a period. Thus, schedulability can be is described by $\phi_{schedulable}$ below.

$$\phi_{schedulable} \stackrel{\text{def}}{=} ||_{i=1}^{n}(ST_i, ET_i)(\{((T, T + P_i)true)^{N_i}\}\text{prj}$$

$$\{(T + P_i, ET_i)\ \square(ac_i = C_i)\})$$

This is the parallel of n formulas where n represents the number of tasks. Sub-formula $(ST_i, ET_i)(\{((T, T + P_i)true)^{N_i}\}\mathsf{prj}\{(T + P_i, ET_i)\square(ac_i = C_i)\})$ means that task τ_i always completes at the start of a new circle. Projection construct is used to ensure $ac_i = C_i$ for every P_i time units over the duration $(ST_i + P_i, ET_i)$.

In order to execute and verify RMS with TMSVL, we take a concrete task set $\tau = \{\tau_1, \tau_2, \tau_3\}$ as an example. For each task, the period is P, the time required for execution is C, and the time ST when the first request occurs are given in the table below. All tasks are supposed to request for execution simultaneously at $T = 0$.

Task	τ_1	τ_2	τ_3
P(ms)	6	8	12
C(ms)	2	3	2
ST	T=0	T=0	T=0

By executing the established RMS model in TMSVL, the values of the notions mentioned above at each state are obtained and shown in Fig.2. Since the scheduler starts at $T = 0$, at the first state (State0) the time is 0, and $Ts = 2$ leads to $T = 2$ at the second state (State1).

```
State0: T=0 Ts=2 ac1=0 ac2=0 ac3=0 d=2 ex1=1 ex2=0 ex3=0 r1=1 r2=1 r3=1 runTaskNum=1
State1: T=2 Ts=3 ac1=2 ac2=0 ac3=0 d=3 ex1=0 ex2=1 ex3=0 r1=0 r2=1 r3=1 runTaskNum=2
State2: T=5 Ts=1 ac1=2 ac2=3 ac3=0 d=2 ex1=0 ex2=0 ex3=1 r1=0 r2=0 r3=1 runTaskNum=3
State3: T=6 Ts=2 ac1=2 ac2=3 ac3=1 d=2 ex1=1 ex2=0 ex3=0 r1=1 r2=0 r3=1 runTaskNum=1
State4: T=8 Ts=3 ac1=2 ac2=3 ac3=1 d=3 ex1=0 ex2=1 ex3=0 r1=0 r2=1 r3=1 runTaskNum=2
State5: T=11 Ts=1 ac1=2 ac2=3 ac3=1 d=1 ex1=0 ex2=0 ex3=1 r1=0 r2=0 r3=1 runTaskNum=3
State6: T=12 Ts=2 ac1=2 ac2=3 ac3=2 d=2 ex1=1 ex2=0 ex3=0 r1=1 r2=0 r3=1 runTaskNum=1
State7: T=14 Ts=2 ac1=2 ac2=3 ac3=0 d=2 ex1=0 ex2=0 ex3=1 r1=0 r2=0 r3=1 runTaskNum=3
State8: T=16 Ts=2 ac1=2 ac2=3 ac3=2 d=3 ex1=0 ex2=1 ex3=0 r1=0 r2=1 r3=0 runTaskNum=2
State9: T=18 Ts=2 ac1=2 ac2=2 ac3=2 d=2 ex1=1 ex2=0 ex3=0 r1=1 r2=1 r3=0 runTaskNum=1
State10: T=20 Ts=1 ac1=2 ac2=2 ac3=2 d=1 ex1=0 ex2=1 ex3=0 r1=0 r2=1 r3=0 runTaskNum=2
State11: T=21 Ts=3 ac1=2 ac2=3 ac3=2 d=0 ex1=0 ex2=0 ex3=0 r1=0 r2=0 r3=0 runTaskNum=0
State12: T=24 Ts=2 ac1=2 ac2=3 ac3=2 d=2 ex1=1 ex2=0 ex3=0 r1=1 r2=1 r3=1 runTaskNum=1
```

Figure 2. Result of simulation

We also verify the schedulability of RMS on τ. The verification result shown in Fig.3 indicates that $RMS\_Sch \rightarrow \phi_{schedulable}$ is valid. That is, the task set τ is schedulable under RMS.

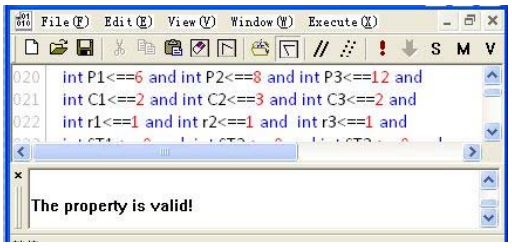

Figure 3. Verification result

V. Related work

A timed state space analysis for real-time scheduling systems via Timed Petri Nets based on the tool ORIS is discussed in [7]. In [8], schedulability checking problem was solved by reachability analysis on standard timed automata with the tool TIMES. Both of the two work relate to our work since they analyze the schedulability of the scheduler with the help of formal models. However, compared with them, TMSVL is expressive enough to describe systems and the desired properties. Thus, a unified model checking approach can be applied to verify the schedulability, which makes a transformation from schedulability analysis to property verification. Moreover, when a property is specified, we can model the system with adjustable time interval Ts to reduce the number of states as many as possible. This largely mitigates the state explosion problem in verification of the real-time systems.

VI. Conclusion

We studied model checking-based schedulability checking approach for RMS with TMSVL. With our tool TMSVL, the whole state space of a TMSVL program can be achieved and the desired property of the model can also be verified automatically. The mechanism that time intervals are adjustable for modeling improves the efficiency of verification. This case study convinces us that TMSVL is quite practical for modeling and verification of real-time systems. In the near future, we will investigate modeling and verification techniques for multiprocessor systems and asynchronous real-time systems on the basis of TMSVL.

Acknowledgment

This research is supported by NSFC Grant Nos. 61133001, 61272117, 61272118, 61202038, and 61322202, National Program on Key Basic Research Project of China (973 Program) Grant No. 2010CB328102. * Corresponding author: Zhenhua Duan.

References

[1] Y. Zou, M.-S. Li, and Q. Wang, "Analysis for scheduling theory and approach of open real-time system," *Journal of Software*, vol. 14, no. 1, pp. 83–90, 2003.

[2] E. Bini, G. C. Buttazzo, and G. M. Buttazzo, "Rate monotonic analysis: the hyperbolic bound," *Computers, IEEE Transactions on*, vol. 52, no. 7, pp. 933–942, 2003.

[3] Y. Manabe and S. Aoyagi, "A feasibility decision algorithm for rate monotonic and deadline monotonic scheduling," *Real-Time Systems*, vol. 14, no. 2, pp. 171–181, 1998.

[4] M. Han, Z. Duan, and X. Wang, "Time constraints with temporal logic programming," in *Formal Methods and Software Engineering*. Springer, 2012, pp. 266–282.

[5] Z.-H. Duan and M. Koutny, "A framed temporal logic programming language," *Journal of Computer Science and Technology*, vol. 19, no. 3, pp. 341–351, 2004.

[6] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[7] G. Bucci, A. Fedeli, L. Sassoli, and E. Vicario, "Timed state space analysis of real-time preemptive systems," *Software Engineering, IEEE Transactions on*, vol. 30, no. 2, pp. 97–111, 2004.

[8] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Schedulability analysis using two clocks," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2003, pp. 224–239.